

Lecture 1 – Syntax and Semantics

SWS128: Software Analysis

Jihyeok Park



2026 Fall

- Testing shows the presence of faults, not their absence.
- An **analyzer** decides whether a program satisfies a property — by running it (dynamic) or without (static).
- No analyzer is sound, complete, and terminating at once (Rice's theorem). Every analysis gives something up.
- Today: before analyzing programs, we must say **precisely** what a program is and what it does.

1. Syntax

- Concrete Syntax

- Abstract Syntax

2. Semantics

- Big-Step Semantics

- Small-Step Semantics

- The Interpreter

- Adding a Heap

3. Riding on Python

- Grammar and Parsing

- Normalization

- Running It

- A Caution

1. Syntax

Concrete Syntax

Abstract Syntax

2. Semantics

Big-Step Semantics

Small-Step Semantics

The Interpreter

Adding a Heap

3. Riding on Python

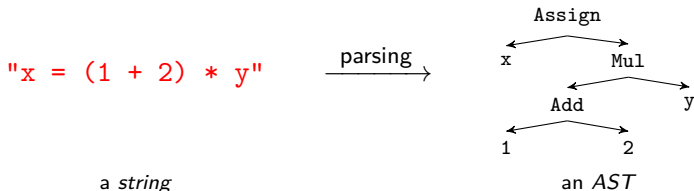
Grammar and Parsing

Normalization

Running It

A Caution

- **Concrete syntax:** the characters a programmer types.
 - $x = (1 + 2) * y$
 - Parentheses, whitespace, keywords, operator precedence.
- **Abstract syntax:** the tree structure that remains once all of that is stripped away.
- **Parsing** takes you from one to the other:



- Every analysis in this course works on the AST.

Let us define the **concrete syntax** of the **core** of MiniPy in BNF:

```
<expr> ::= <num> | <id>
          | <expr> "+" <expr>
          | <expr> "*" <expr>
          | <expr> "<" <expr>
          | "(" <expr> ")"

<stmt> ::= "pass"
          | <id> "=" <expr>
          | "if" <expr> ":" <block> "else" ":" <block>
          | "while" <expr> ":" <block>

<block> ::= <stmt>+
```

- **Nonterminals:** <expr>, <stmt>, <block>, <num>, <id>
- **Terminals:** "+", "*", "<", "(", "if", "while", "pass", ":", "="
- **Ambiguous** as written (1 + 2 * 3 has two parse trees), so we fix **precedence** (* > + > <) and **associativity** (all left).

And its **abstract syntax**:

Expressions $e ::= n \mid x \mid e + e \mid e * e \mid e < e$

Statements $s ::=$ pass
 $\mid x := e$
 $\mid \text{if } e \text{ then } s^+ \text{ else } s^+$
 $\mid \text{while } e \text{ do } s^+$

where $n \in \mathbb{Z}$ and $x \in \mathbb{X}$.

- No parentheses, no precedence, no ambiguity: the **tree** already says how the program is grouped.
- We write $:=$ for assignment, to keep it apart from mathematical equality in the rules that follow.

```
@dataclass(frozen=True)
class Expr: pass
```

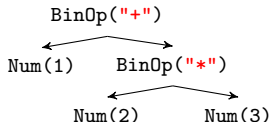
```
@dataclass(frozen=True)
class Num(Expr):
    n: int
```

```
@dataclass(frozen=True)
class Var(Expr):
    x: str
```

```
@dataclass(frozen=True)
class BinOp(Expr):
    op: str
    left: Expr
    right: Expr
```

So $1 + 2 * 3$ is built as:

```
BinOp("+",
      Num(1),
      BinOp("*",
            Num(2),
            Num(3)))
```



Precedence is gone — the shape carries it.

```
@dataclass(frozen=True)
class Stmt: pass
```

```
@dataclass(frozen=True)
class Pass(Stmt): pass
```

```
@dataclass(frozen=True)
class Assign(Stmt):
    x: str
    e: Expr
```

```
@dataclass(frozen=True)
class If(Stmt):
    cond: Expr
    then: Block
    els: Block
```

```
@dataclass(frozen=True)
class While(Stmt):
    cond: Expr
    body: Block
```

```
Block = tuple["Stmt", ...]
```

So `while i < n: i = i + 1` is built as:

```
While(
    BinOp("<", Var("i"), Var("n")),
    (Assign("i",
            BinOp("+", Var("i"), Num(1))),),
)
```

Indentation is gone — the tuple carries it.

1. Syntax

Concrete Syntax

Abstract Syntax

2. Semantics

Big-Step Semantics

Small-Step Semantics

The Interpreter

Adding a Heap

3. Riding on Python

Grammar and Parsing

Normalization

Running It

A Caution

- Syntax says what programs **look like**. Semantics says what they **do**.
- The tool is **operational semantics**: rules that say how a program steps from one **state** to the next.
- A state $\sigma \in \mathbb{S} = \mathbb{X} \rightarrow \mathbb{V}$ maps variables to values.

$$\boxed{\sigma \vdash e \Rightarrow v} \quad \text{where} \quad v ::= n \mid \text{true} \mid \text{false}$$

“in state σ , expression e evaluates to value v .”

$$\frac{}{\sigma \vdash n \Rightarrow n} \qquad \frac{}{\sigma \vdash x \Rightarrow \sigma(x)}$$

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2}$$

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 * e_2 \Rightarrow n_1 \times n_2}$$

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 < e_2 \Rightarrow n_1 < n_2}$$

$$\begin{array}{c}
 \frac{}{\sigma \vdash n \Rightarrow n} \qquad \frac{}{\sigma \vdash x \Rightarrow \sigma(x)} \qquad \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \\
 \\
 \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 * e_2 \Rightarrow n_1 \times n_2} \qquad \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 < e_2 \Rightarrow n_1 < n_2}
 \end{array}$$

A **derivation tree** is a proof that a given judgment holds.

With $\sigma = [x \mapsto 2]$:

$$\frac{
 \frac{
 \frac{}{\sigma \vdash x \Rightarrow 2} \quad \frac{}{\sigma \vdash 3 \Rightarrow 3}
 }{\sigma \vdash x * 3 \Rightarrow 6}
 \quad \frac{}{\sigma \vdash 10 \Rightarrow 10}
 }{\sigma \vdash x * 3 < 10 \Rightarrow \text{true}}$$

$\boxed{\sigma \vdash s \Rightarrow \sigma'}$ “running s in σ ends in σ' .”

$$\frac{\sigma \vdash e \Rightarrow v}{\sigma \vdash x := e \Rightarrow \sigma[x \mapsto v]}$$

$$\frac{}{\sigma \vdash \text{pass} \Rightarrow \sigma}$$

$$\frac{\sigma \vdash e \Rightarrow \text{true} \quad \sigma \vdash s_1 \Rightarrow \sigma'}{\sigma \vdash \text{if } e \text{ then } s_1 \text{ else } s_2 \Rightarrow \sigma'} \quad \frac{\sigma \vdash e \Rightarrow \text{false} \quad \sigma \vdash s_2 \Rightarrow \sigma'}{\sigma \vdash \text{if } e \text{ then } s_1 \text{ else } s_2 \Rightarrow \sigma'}$$

$$\frac{\sigma \vdash e \Rightarrow \text{false}}{\sigma \vdash \text{while } e \text{ do } s \Rightarrow \sigma}$$

$$\frac{\sigma \vdash e \Rightarrow \text{true} \quad \sigma \vdash s \Rightarrow \sigma_1 \quad \sigma_1 \vdash \text{while } e \text{ do } s \Rightarrow \sigma'}{\sigma \vdash \text{while } e \text{ do } s \Rightarrow \sigma'}$$

Big-step jumps from start to finish. Sometimes we want the **path**:

$$\langle \sigma, s \rangle \rightarrow \langle \sigma', s' \rangle$$

$$\frac{\sigma \vdash e \Rightarrow v}{\langle \sigma, x = e \rangle \rightarrow \langle \sigma[x \mapsto v], \text{pass} \rangle} \qquad \frac{\langle \sigma, s_1 \rangle \rightarrow \langle \sigma', s'_1 \rangle}{\langle \sigma, s_1; s_2 \rangle \rightarrow \langle \sigma', s'_1; s_2 \rangle}$$

$$\langle \sigma, \text{while } e : s \rangle \rightarrow \langle \sigma, \text{if } e : (s; \text{while } e : s) \text{ else: pass} \rangle$$

- A non-terminating loop is now an **infinite sequence** of steps, not a missing derivation.
- We will use big-step for the interpreter, and small-step thinking when an analysis needs to reason about a single step.

```
Value = int | bool                                # v ::= n | true | false
State = dict[str, Value]                          # s in S = X -> V

def eval_expr(e: Expr, s: State) -> Value:
    match e:
        case Num(n):                               return n
        case Var(x):                               return s[x]
        case BinOp("+", l, r):                     return eval_expr(l, s) + eval_expr(r, s)
        case BinOp("*", l, r):                     return eval_expr(l, s) * eval_expr(r, s)
        case BinOp("<", l, r):                       return eval_expr(l, s) < eval_expr(r, s)
        case _:                                     raise NotImplementedError(e)

def exec_stmt(st: Stmt, s: State) -> State:
    match st:
        case Assign(x, e):
            return {**s, x: eval_expr(e, s)}
        case If(c, then, els):
            return exec_block(then if eval_expr(c, s) else els, s)
        case While(c, body):
            while eval_expr(c, s):
                s = exec_block(body, s)
            return s
        case _:                                     raise NotImplementedError(st)
```

The core has no lists. Add them, and something new happens:

```
xs = [1, 2]
ys = xs
ys[0] = 7
r = xs[0]    # 7, not 1
```

`xs` and `ys` are **aliases**: two names for one list.

A state $\sigma : \mathbb{X} \rightarrow \mathbb{V}$ cannot say that — it has no notion of “the same list.”

So a list is a **location** ℓ , and a **heap** says what is stored there:

$$v ::= n \mid \text{true} \mid \text{false} \mid \ell \qquad H \in \mathbb{H} = \mathbb{L} \xrightarrow{\text{fin}} \mathbb{V}^*$$

Every judgment now takes a heap in and gives a heap out:

$$\langle H, \sigma \rangle \vdash e \Rightarrow \langle v, H' \rangle$$

$$\langle H, \sigma \rangle \vdash s \Rightarrow \langle H', \sigma' \rangle$$

The old rules only **thread** H through, unchanged.

$$\frac{\langle H_{i-1}, \sigma \rangle \vdash e_i \Rightarrow \langle v_i, H_i \rangle \quad (1 \leq i \leq k) \quad \ell \notin \text{dom}(H_k)}{\langle H_0, \sigma \rangle \vdash [e_1, \dots, e_k] \Rightarrow \langle \ell, H_k[\ell \mapsto (v_1, \dots, v_k)] \rangle}$$

*“a list literal **allocates** a fresh location.”*

$$\frac{\langle H, \sigma \rangle \vdash e_1 \Rightarrow \langle \ell, H_1 \rangle \quad \langle H_1, \sigma \rangle \vdash e_2 \Rightarrow \langle n, H_2 \rangle \quad H_2(\ell) = (v_0, \dots, v_{k-1}) \quad 0 \leq n < k}{\langle H, \sigma \rangle \vdash e_1 [e_2] \Rightarrow \langle v_n, H_2 \rangle}$$

*“indexing **loads** from the heap.”*

$$\frac{\sigma(x) = \ell \quad \langle H, \sigma \rangle \vdash e_1 \Rightarrow \langle n, H_1 \rangle \quad \langle H_1, \sigma \rangle \vdash e_2 \Rightarrow \langle v, H_2 \rangle}{\langle H, \sigma \rangle \vdash x [e_1] := e_2 \Rightarrow \langle H_2[\ell \mapsto H_2(\ell)[n \mapsto v]], \sigma \rangle}$$

*“index assignment **stores** into the heap — σ does not change.”*

Nothing is shared behind the scenes: the interpreter threads H the same way.

```
Value = int | bool | Loc                                # v ::= n | true | false | l
Heap  = dict[Loc, tuple[Value, ...]]                    # H in L -> V*
def eval_expr(e: Expr, h: Heap, s: State) -> tuple[Value, Heap]:
    match e:
        case ListLit(items):
            vals = []
            for i in items:
                v, h = eval_expr(i, h, s)
                vals.append(v)
            loc = Loc(len(h))                            # l not in dom(h)
            return loc, {**h, loc: tuple(vals)}
        case Index(base, idx):
            loc, h = eval_expr(base, h, s)
            n, h = eval_expr(idx, h, s)
            return h[loc][n], h

def exec_stmt(st: Stmt, h: Heap, s: State) -> tuple[Heap, State]:
    match st:
        case IndexAssign(x, idx, e):
            loc = s[x]
            n, h = eval_expr(idx, h, s)
            v, h = eval_expr(e, h, s)
            return {**h, loc: h[loc][:n] + (v,) + h[loc][n + 1:]}, s
```

A MiniPy program is **one function**, and return is its **last line**:

```
def main(n, uid):          # parameters: the input
    ...
    return r                # last line: the output
```

$$\langle \emptyset, [x_i \mapsto v_i] \rangle \vdash s \Rightarrow \langle H', \sigma' \rangle \quad \langle H', \sigma' \rangle \vdash e \Rightarrow \langle v, H'' \rangle$$

“run the body from the arguments, then evaluate e .”

- No calls — not even `input()` or `print()`. Input comes in through the parameters and goes out through `return`.
- Also there: `str` with `+`, `<=`, `==`, `!=`, and `or` `not`, unary `-`, and `None`.
- Every rule is written out in the spec on the course page:
[minipy-spec.pdf](#)

1. Syntax

Concrete Syntax

Abstract Syntax

2. Semantics

Big-Step Semantics

Small-Step Semantics

The Interpreter

Adding a Heap

3. Riding on Python

Grammar and Parsing

Normalization

Running It

A Caution

This course targets MiniPy, a small subset of Python. Python publishes both of the things we just wrote down:

- concrete syntax, as a grammar

docs.python.org/3/reference/grammar.html

- abstract syntax, as a list of node types

docs.python.org/3/library/ast.html

And the ast module **parses for us**. We never write a parser.

```
>>> import ast
>>> print(ast.dump(ast.parse("x = 1 + 2"), indent=2))
Module(
  body=[
    Assign(
      targets=[Name(id='x', ctx=Store())],
      value=BinOp(
        left=Constant(value=1),
        op=Add(),
        right=Constant(value=2))))]
```

- Same shape as our AST, with more detail than we need (ctx, positions, and a hundred node types we do not support).

So we **normalize** Python's AST into ours, and **reject** everything else.

```
def expr(node: ast.expr) -> Expr:
    match node:
        case ast.Constant(value=int(n)):
            return Num(n)
        case ast.Name(id=x):
            return Var(x)
        case ast.BinOp(left=l, op=op, right=r) if type(op) in BINOPS:
            return BinOp(BINOPS[type(op)], expr(l), expr(r))
        case ast.Compare(left=l, ops=[op], comparators=[r]):
            return BinOp(CMPOPS[type(op)], expr(l), expr(r))
        case _:
            raise Unsupported(node)
```

Download `syntax.py` and `interp.py` from the course page:

plrg.korea.ac.kr/courses/sws128

```
>>> from syntax import parse
>>> from interp import run
>>> src = """
... def sum(n):
...     i = 0
...     s = 0
...     while i <= n:
...         s = s + i
...         i = i + 1
...     return s
... """
>>> parse(src)
Prog(name='sum', params=('n',), body=(Assign(x='i', e=Num(n=0)),
Assign(x='s', e=Num(n=0)), While(...)), ret=Var(x='s'))
>>> run(parse(src), [5])
15
```

We borrow Python's **syntax**, not its **semantics**. They mostly agree — but not always.

```
x = 1 < 2 < 3
```

- Python **chains**: $1 < 2$ and $2 < 3$, so `x` is `True`.
- MiniPy has one binary `<`: $(1 < 2) < 3$, that is `true < 3`.

1. Syntax

- Concrete Syntax

- Abstract Syntax

2. Semantics

- Big-Step Semantics

- Small-Step Semantics

- The Interpreter

- Adding a Heap

3. Riding on Python

- Grammar and Parsing

- Normalization

- Running It

- A Caution

- Control Flow Graphs

Jihyeok Park

`jihyeok_park@korea.ac.kr`

`https://plrg.korea.ac.kr`