软件分析与架构设计

操作语义

何冬杰
重庆大学

语法（Syntax)

- ❑Grammar：which programs are syntactically correct?
  - ➢Terminals Σ， Non-terminals N, Initial symbol $s \in N$, Productions P
- ❑Example：

Arithmetic Exmpression

Σ	$=$	$\{0,1,\ldots,9,+,-\}$
N	$=$	$\{Exp, Num, Op, Digit\}$
s	$=$	Exp
Productions		
Exp	$\rightarrow$	$Num \mid Exp\ Op\ Exp$
Op	$\rightarrow$	$+ \mid -$
Num	$\rightarrow$	$Digit \mid Digit\ Num$
$Digit$	$\rightarrow$	$0 \mid 1 \mid \ldots \mid 9$

What is not part of the language?

A. 12+2

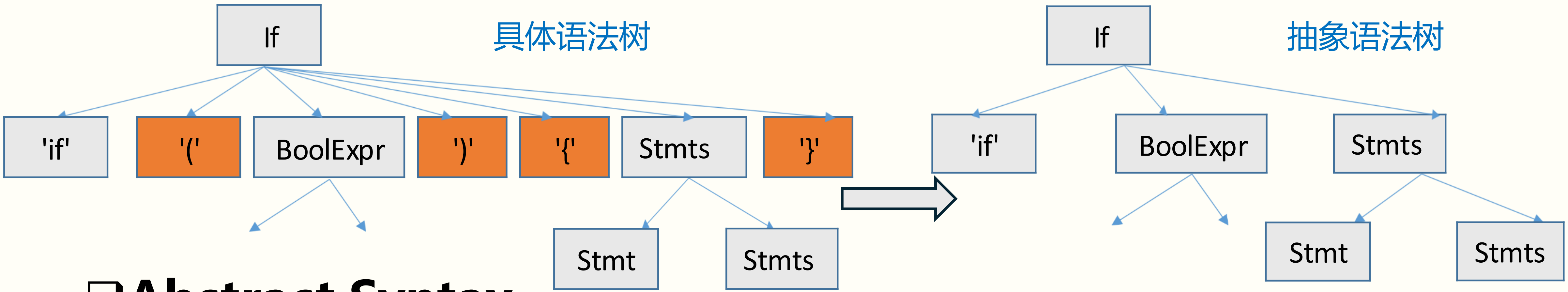
B. 2+(12-4)

C. 11*4

D. 12345609

Concrete vs. Abstract Syntax

- **Syntax Tree:**
  - 基于编程语言文法的源代码的树表示
- **Concrete Syntax**
  - The rules by which programs can be expressed as strings



- **Abstract Syntax**
  - Concerns only statements, expressions, and their operands
  - Don’t care about parentheses, semicolons, keywords, etc.

Learning Goals

- Recognize the basic WHILE demonstration language and define its abstract syntax.
- Describe the function of an AST and outline the principles behind AST walkers for simple bug-finding analyses
- Define the meaning of programs using operational semantics
- Read and write inference rules and derivation trees
- Use big- and small-step semantics to show how WHILE programs evaluate
- Use structural induction to prove things about program semantics

Abstract Syntax of SIMP

- SIMP：simple imperative PL

$$
\begin{array}{rcl}
S & ::= & x := a \\
& | & \text{skip} \\
& | & S_1;\ S_2 \\
& | & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
& | & \text{while } b \text{ do } S
\end{array}
\qquad
\begin{array}{rcl}
b & ::= & \text{true} \\
& | & \text{false} \\
& | & \text{not } b \\
& | & b_1\ op_b\ b_2 \\
& | & a_1\ op_r\ a_2
\end{array}
\qquad
\begin{array}{rcl}
a & ::= & x \\
& | & n \\
& | & a_1\ op_a\ a_2
\end{array}
\qquad
\begin{array}{rcl}
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \ \mid\ \leqslant\ \mid\ = \\
& | & > \ \mid\ \geqslant \\
op_a & ::= & + \mid - \mid * \mid /
\end{array}
$$

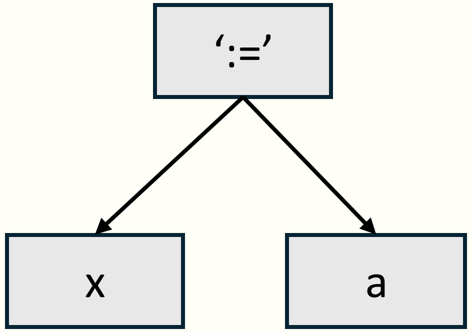
- Meta-variables frequently used for easy of notation

S	statements
a	arithmetic expressions (AExp)
x, y	program variables (Vars)
n	number literals
b	boolean expressions (BExp)

根据需要，后面会进一步添加产生式

如何根据文法构建AST?

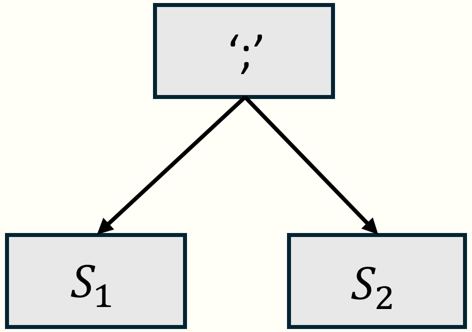
(1) x:=a



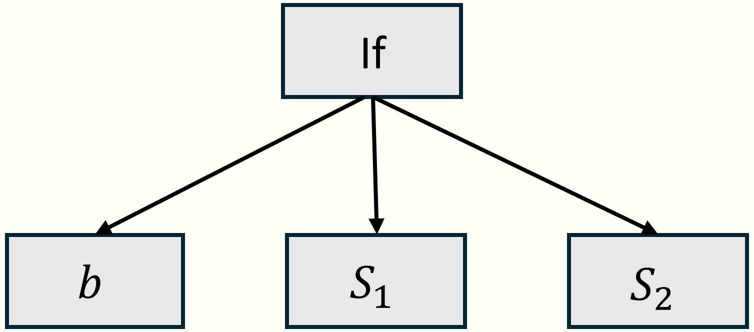
(2) skip

(do nothing)

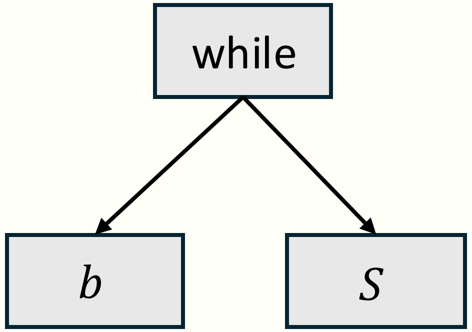
(3) $S_1; S_2$



(4) if b then S_1 else S_2



(5) while b do S



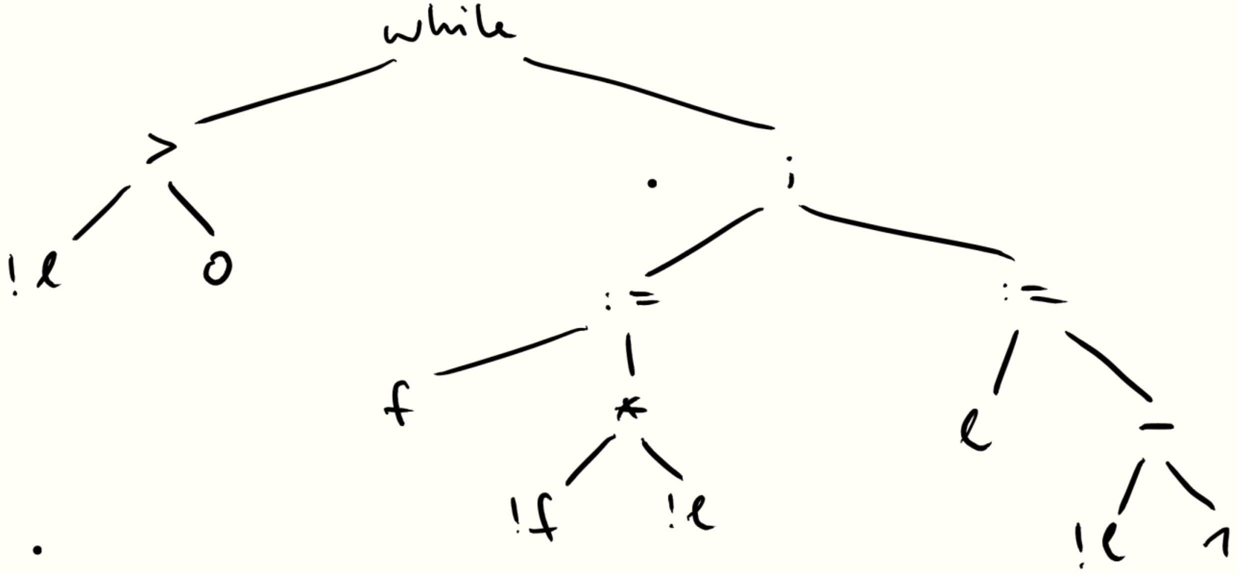
$$
\begin{array}{lcl}
S & ::= & x := a \\
 & | & \text{skip} \\
 & | & S_1;\ S_2 \\
 & | & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & | & \text{while } b \text{ do } S
\end{array}
$$

两个构建AST的例子

Ex. 2

```
while (!l > 0) do (
    f := !f * !l ;
    l := !l - 1
)
```

AST:

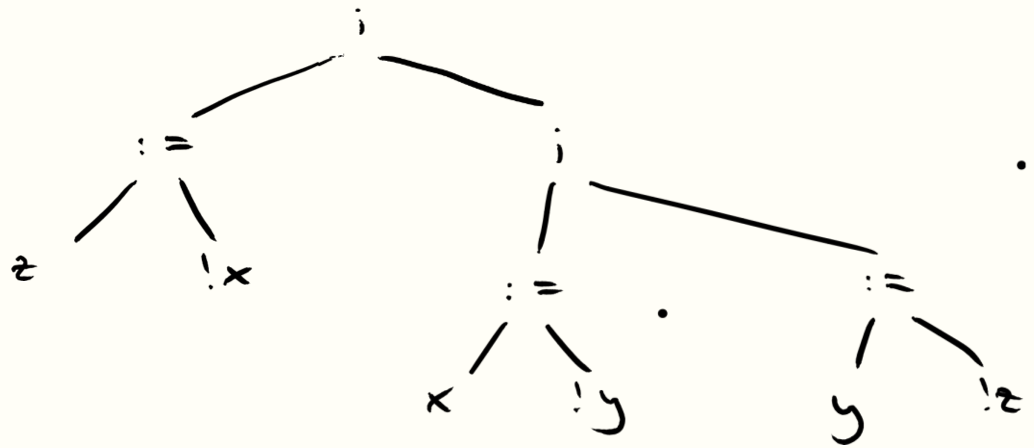


Ex. 1

```
z := !x ; (x := !y ; y := !z)
```

... swap values in x and y

AST:



(optional) 作业: 手动构建一个AST

- 以y为临时变量计算z = x!
- 包含了复合语句、赋值语句、条件和循环语句
- 没有IO语句，输入输出是隐式的
- 所有变量是整数

```
y  := x;
z  := 1;
if y > 0 then
  while y > 1 do
    z := z * y;
    y := y - 1
else
  skip
```

$$
\begin{array}{lll}
S & ::= & x := a \\
  & \mid & \text{skip} \\
  & \mid & S_1;\ S_2 \\
  & \mid & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
  & \mid & \text{while } b \text{ do } S
\end{array}
\qquad
\begin{array}{lll}
b & ::= & \text{true} \\
  & \mid & \text{false} \\
  & \mid & \text{not } b \\
  & \mid & b_1\ op_b\ b_2 \\
  & \mid & a_1\ op_r\ a_2
\end{array}
\qquad
\begin{array}{lll}
a & ::= & x \\
  & \mid & n \\
  & \mid & a_1\ op_a\ a_2
\end{array}
\qquad
\begin{array}{lll}
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \mid \leqslant \mid = \\
     & \mid & > \mid \geqslant \\
op_a & ::= & + \mid - \mid * \mid /
\end{array}
$$

(optional)项目：AST walking

- One way to find “bugs” is to walk the AST
  - Traverse the AST, look for nodes of a particular type
  - Check the neighborhood of the node for particular patterns
- 检测：“shifting by more than 31 bits”
  - e.g. ”x << -3”, ”z >> 35”, 对于32 位整数变量，这些操作可能表示意外的拼写错误，因为将数字移出范围 (0, 32) 是没有意义的
- 提醒：基于现有框架去探索
  - Python’s ”astor” package designed for Python ASTs. Clean API; highly specific.
  - LLVM/Clang：基于visitor pattern
    - class Visitor has a visitX method for each type of AST node X
    - Default Visitor code just descends the AST, visiting each node
    - To do something interesting for AST element of type X, override visitX

操作语义（Operational Semantics)

- 描述程序如何执行
  - How would I execute this?
- 为什么需要操作语义?
- 举例： 下面C代码中f()函数的参数是什么?

```c
int i = 5;
f(i++, --i);
```

- Option 1 (left-to-right)： 5， 5
- Option 2 (right-to-left)： 4， 4
- Both options are possible in C!
  - Unspecified Semantics
  - Compiler decides

- Want: (almost) all behavior should be clearly specified

操作语义（Operational Semantics）

- Specifies how expressions and statements should be evaluated depending on the form of the expression
  - 0，1, 2, . . . 已经是值，无需进一步计算
  - 4 + 2：整数相加得结果，可推广值只包含数值的任意表达式： n1 + n2
  - a1+ a2的计算方法如下（按从左到右的AST后序遍历）：
    - 首先将表达式 a1 的值计算为 n1
    - 然后讲表达式a2 的值计算为n2
    - 将计算结果用n1 + n2表示
- 操作语义抽象了具体解释器的执行过程

推理规则（Inference Rules)

❑一般性推理规则

$$\frac{premise_1 \quad premise_2 \quad \ldots \quad premise_n}{conclusion}$$

- ➢If ALL of the premises above the line can be proved true, then the conclusion holds as well.
- ➢用于定义语义

❑公理（ Axiom ）： no premises

$$\frac{}{conclusion} \qquad \text{或} \qquad conclusion$$

大步语义（Big-Step Semantics）

- **Uses down-arrow ⇓ notation to denote evaluation to normal form**
- **a ⇓ n is a judgment that expression a is evaluated to value n**
  - For example: 4 + 2 + 9 ⇓ 15
- **You can think of this as a logical proposition.**
  - The semantics of a language determines what judgments are provable.

大步语义举例

- ❑ **Big-step semantics for ADD**

$$\frac{}{n \Downarrow n}\ \textit{big-int} \qquad\qquad \frac{a_1 \Downarrow n_1 \quad a_2 \Downarrow n_2}{a_1 + a_2 \Downarrow n_1 + n_2}\ \textit{big-add}$$

- ❑ **Derive (4 + 2) + 9 ⇓ 15 from the rules**
  - ➢ The derivation provides a proof of (4 + 2) + 9 ⇓ 15 using only axioms and inference rules.

forms a **derivation tree** ⇨

$$\dfrac{\dfrac{4 \Downarrow 4 \quad 2 \Downarrow 2}{4 + 2 \Downarrow 6} \qquad 9 \Downarrow 9}{(4 + 2) + 9 \Downarrow 15}$$

Big-Step Semantics for SIMP

其他算术和布尔
运算处理类似

Expression

$$\frac{}{\langle E, n\rangle \Downarrow n}\ \textit{big-int} \qquad \frac{}{\langle E, x\rangle \Downarrow E(x)}\ \textit{big-var} \qquad \frac{\langle E, a_1\rangle \Downarrow n_1 \quad \langle E, a_2\rangle \Downarrow n_2}{\langle E, a_1 + a_2\rangle \Downarrow n_1 + n_2}\ \textit{big-add}$$

No side effects to the environment

Statement

$$\frac{}{\langle E, \texttt{skip}\rangle \Downarrow E}\ \textit{big-skip} \qquad \frac{\langle E, a\rangle \Downarrow n}{\langle E, x := a\rangle \Downarrow E[x \mapsto n]}\ \textit{big-assign} \qquad \frac{\langle E, S_1\rangle \Downarrow E' \quad \langle E', S_2\rangle \Downarrow E''}{\langle E, S_1; S_2\rangle \Downarrow E''}\ \textit{big-seq}$$

$$\frac{\langle E, b\rangle \Downarrow \texttt{true} \quad \langle E, S_1\rangle \Downarrow E'}{\langle E, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rangle \Downarrow E'}\ \textit{big-iftrue} \qquad \frac{\langle E, b\rangle \Downarrow \texttt{false} \quad \langle E, S_2\rangle \Downarrow E'}{\langle E, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rangle \Downarrow E'}\ \textit{big-iffalse}$$

$$\frac{\langle E, b\rangle \Downarrow \texttt{false}}{\langle E, \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E}\ \textit{big-whilefalse} \qquad \frac{\langle E, b\rangle \Downarrow \texttt{true} \quad \langle E, S; \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E'}{\langle E, \texttt{while}\ b\ \texttt{then}\ S\rangle \Downarrow E'}\ \textit{big-whiletrue}$$

Statements can have side effects

Environment E : var → Z General form: <E, S> ⇓ E′

举例：States propagate in derivations

- What will **x * 2 - 6** evaluate to in state **E1 = {x ↦ 4}**?

$$\dfrac{\dfrac{\langle E_1, x\rangle \Downarrow 4 \quad \langle E_1, 2\rangle \Downarrow 2}{\langle E_1, x * 2\rangle \Downarrow 8} \qquad \langle E_1, 6\rangle \Downarrow 6}{\langle E_1, (x * 2) - 6\rangle \Downarrow 2}$$

- $\vdash \langle E1, x * 2 - 6\rangle \Downarrow 2$
  - ➢this evaluation is provable via a well-formed derivation

Big-Step Semantics: Discussion

- **Inference rules suggest an AST interpreter**
  - Recursively evaluate operands, then current node
  - post-order traversal
- **Disadvantages:**
  - Cannot reason about non-terminating loops
    - e.g. while true do skip
  - Does not model intermediate states
  - Needed for semantics of concurrent execution models
    - e.g. Java threads

小步语义（Small-Step Semantics)

- ❑Each step is an atomic rewrite of the program
- ❑Execution is a sequence of (possibly infinite) steps
  - ➢ ⟨E1, (x * 2) - 6⟩ → ⟨E1, 4 * 2 - 6⟩ → ⟨E1, 8 - 6⟩ → 2
- ❑Small arrow notation for a single step:

$$\langle E, a\rangle \rightarrow_a a' \qquad \langle E, b\rangle \rightarrow_b b' \qquad \langle E, S\rangle \rightarrow \langle E', S'\rangle$$

  - ➢subscripts on the arrows can be omitted when context is clear

Small-Step Semantics for SIMP

$$\frac{}{\langle E, x\rangle \rightarrow_a E(x)}\ \textit{small-var} \qquad \frac{}{\langle E, n\rangle \rightarrow_a n}\ \textit{small-int}$$

$$\frac{\langle E, S_1\rangle \rightarrow \langle E', S_1'\rangle}{\langle E, S_1; S_2\rangle \rightarrow \langle E', S_1'; S_2\rangle}\ \textit{small-seq-congruence}$$

$$\frac{}{\langle E, \mathtt{skip}; S_2\rangle \rightarrow \langle E, S_2\rangle}\ \textit{small-seq}$$

$$\frac{\langle E, a_1\rangle \rightarrow_a a_1'}{\langle E, a_1 + a_2\rangle \rightarrow_a a_1' + a_2}\ \textit{small-add-left}$$

$$\frac{\langle E, a_2\rangle \rightarrow_a a_2'}{\langle E, n_1 + a_2\rangle \rightarrow_a n_1 + a_2'}\ \textit{small-add-right}$$

$$\frac{}{\langle E, n_1 + n_2\rangle \rightarrow_a n_1 + n_2}\ \textit{small-add}$$

*small-assign*处理类似 *big-assign*

$$\frac{\langle E, a\rangle \Downarrow n}{\langle E, x := a\rangle \Downarrow E[x \mapsto n]}\ \textit{big-assign}$$

$$\frac{\langle E, b\rangle \rightarrow_b b'}{\langle E, \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2\rangle \rightarrow \langle E, \mathtt{if}\ b'\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2\rangle}\ \textit{small-if-congruence}$$

$$\frac{}{\langle E, \mathtt{if}\ \mathtt{true}\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2\rangle \rightarrow \langle E, S_1\rangle}\ \textit{small-iftrue}$$

small-iffalse 处理类似

$$\frac{}{\langle E, \mathtt{while}\ b\ \mathtt{do}\ S\rangle \rightarrow \langle \mathtt{if}\ b\ \mathtt{then}\ S; \mathtt{while}\ b\ \mathtt{do}\ S\ \mathtt{else}\ \mathtt{skip}\rangle}\ \textit{small-while}$$

Example: $P = z := !x ; (x := !y ; y := !z)$

$s = \{ z \mapsto 0, x \mapsto 1, y \mapsto 2 \}$

$$\langle P, s \rangle \to \dots \to \langle \mathrm{skip}, s[z \mapsto 1, x \mapsto 2, y \mapsto 1] \rangle$$

each step: axiom or rule $\hookrightarrow$ proof tree

Excerpt of proof tree:

$$\dfrac{\dfrac{\dfrac{}{\langle !x, s \rangle \to \langle 1, s \rangle}\ (\text{var})}{\langle z := !x, s \rangle \to \langle z := 1, s \rangle}\ (:=_R)}{\langle P, s \rangle \to \langle z := 1 ; (x := !y ; y := !z), s \rangle}\ (\text{seq})$$

Evaluation Sequence

For $\langle \mathrm{E}, \mathrm{S} \rangle$, the **evaluation sequence** is a uniquely defined sequence of transitions that starts with $\langle \mathrm{E}, \mathrm{S} \rangle$ and has maximal length.

- **Multi-step notation:** $\langle E, S \rangle \to^* \langle E', S' \rangle$

$$\frac{}{\langle E, S \rangle \to^* \langle E, S \rangle} \textit{ multi-reflexive} \qquad \frac{\langle E, S \rangle \to \langle E', S' \rangle \quad \langle E', S' \rangle \to^* \langle E'', S'' \rangle}{\langle E, S \rangle \to^* \langle E'', S'' \rangle} \textit{ multi-inductive}$$

- **3 possible outputs:**
  - Infinite sequences: $\langle E, \mathit{while}\ \mathrm{True}\ \mathit{do}\ \mathit{skip} \rangle \to \cdots \to \langle E, \mathit{while}\ \mathrm{True}\ \mathit{do}\ \mathit{skip} \rangle$
  - Evaluation terminates: $\langle E_{in}, S \rangle \to^* \langle E_{out}, \mathit{skip} \rangle$
  - Evaluation blocked:
    - $\langle E, \mathit{if}\ x > 0\ \mathit{then}\ S\ \mathit{else}\ \mathit{skip} \rangle \to^* ?$, where $x \notin \mathit{dom}(E)$

Proofs over semantics

- **Given some operational semantics, $\langle E, a\rangle \Downarrow n$ is provable if there exists a well-formed derivation with $\langle E, a\rangle \Downarrow n$ as its conclusion**
  - "well-formed" = "every step in the derivation is a valid instance of one of the inference rules"
  - $\vdash \langle E, a\rangle \Downarrow n$: "it is provable that $\langle E, a\rangle \Downarrow n$"

- **Once semantics is defined clearly, we can reason about programs rigorously via proofs by structural induction**
- **Recall *mathematical induction* :**
  - To prove $\forall\, n{:}\, P(n)$ by induction on natural numbers
  - Base case: show that $P(0)$ holds
  - Inductive case: show that $\forall\, m{:}\, P(m) \rightarrow P(m+1)$

Proofs by Structural Induction

- **Prove $\forall\, \boldsymbol{a} \in \boldsymbol{Aexp}: \boldsymbol{P(a)}$ by induction on structure of syntax**
  - Base cases: show that $P(x)$ and $P(n)$ holds
  - Inductive cases: show that

$$
\begin{array}{lll}
a & ::= & x \\
 & | & n \\
 & | & a_1 \; op_a \; a_2
\end{array}
\qquad\qquad
op_a \quad ::= \quad + \mid - \mid * \mid /
$$

Proofs by Structural Induction

Example. Let $L(a)$ be the number of literals and variable occurrences in some expression a and $O(a)$ be the number of operators in a. Prove by induction on the structure of a that $\forall a \in$ `Aexp` $.\ L(a) = O(a) + 1$:

Base cases:

- Case $a = n$. $L(a) = 1$ and $O(a) = 0$
- Case $a = x$. $L(a) = 1$ and $O(a) = 0$

Inductive case 1: Case $a = a_1 + a_2$

- By definition, $L(a) = L(a_1) + L(a_2)$ and $O(a) = O(a_1) + O(a_2) + 1$.
- By the induction hypothesis, $L(a_1) = O(a_1) + 1$ and $L(a_2) = O(a_2) + 1$.
- Thus, $L(a) = O(a_1) + O(a_2) + 2 = O(a) + 1$.

The other arithmetic operators follow the same logic.

Prove that SIMP is deterministic

- **Deterministic:**
  - if the program terminates, it evaluates to a unique value.

$$
\begin{array}{lll}
\forall a \in \texttt{Aexp}\,. & \forall E\,.\,\forall n, n' \in \mathbb{N}\,. & \langle E, a\rangle \Downarrow n \wedge \langle E, a\rangle \Downarrow n' \Rightarrow n = n' \\
\forall P \in \texttt{Bexp}\,. & \forall E\,.\,\forall b, b' \in \mathcal{B}\,. & \langle E, P\rangle \Downarrow b \wedge \langle E, P\rangle \Downarrow b' \Rightarrow b = b' \\
\forall S\,. & \forall E, E', E''\,. & \langle E, S\rangle \Downarrow E' \wedge \langle E, S\rangle \Downarrow E'' \Rightarrow E' = E''
\end{array}
$$

- **Expressions are easier to prove**
- **But statements are not**
  - Rule for while is recursive; doesn’t depend only on sub-expressions

$$
\frac{\langle E, b\rangle \Downarrow \texttt{true} \quad \langle E, S; \texttt{while}\; b\; \texttt{do}\; S\rangle \Downarrow E'}{\langle E, \texttt{while}\; b\; \texttt{then}\; S\rangle \Downarrow E'}\; \textit{big-whiletrue}
$$

Prove that SIMP is deterministic

证明： $\boxed{\forall S . \quad \forall E, E', E'' . \quad \langle E, S\rangle \Downarrow E' \wedge \langle E, S\rangle \Downarrow E'' \Rightarrow E' = E''}$

- **Let** $D :: \langle E, S\rangle \Downarrow E'$, **and** $D' :: \langle E, S\rangle \Downarrow E''$
- **Base case: skip**

$$\frac{}{\langle E, \texttt{skip}\rangle \Downarrow E}\ \textit{big-skip}$$

- **Inductive cases:**
  - Need to show that the property hold when the last rule used in D was each of the possible non-skip statements
  - Suppose the last rule used was while-true

$$D ::= \frac{D_1 :: \langle E, b\rangle \Downarrow \texttt{true} \quad D_2 :: \langle E, S\rangle \Downarrow E_1 \quad D_3 :: \langle E_1, \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E'}{\langle E, \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E'}$$

    - $D' :: \langle E, \text{while b do S}\rangle \Downarrow E''$ must have sub-derivations:
      - $D_1' :: \langle E, b\rangle \Downarrow \text{true}$, $D_2' :: \langle E, S\rangle \Downarrow E_1'$ and $D_3' :: \langle E_1', \text{while b do S}\rangle \Downarrow E''$
    - By induction hypothesis, $E_1 = E_1'$ and $E'' = E'$
  - other cases are left as for an exercise

(optional)作业：练习结构归纳证明

- Prove that small-step and big-step semantics of expressions produce equivalent results.

$$\forall a \in \texttt{AExp} \,.\, \langle E, a \rangle \to_a^* n \Leftrightarrow \langle E, a \rangle \Downarrow n$$

- Can be proved via structural induction over syntax

$$\begin{array}{lcl} a & ::= & x \\ & | & n \\ & | & a_1 \; op_a \; a_2 \end{array} \qquad \begin{array}{lcl} op_b & ::= & \text{and} \mid \text{or} \\ op_r & ::= & < \;\mid\; \leqslant \;\mid\; = \\ & | & > \;\mid\; \geqslant \\ op_a & ::= & + \mid - \mid * \mid / \end{array}$$

S	statements
a	arithmetic expressions (AExp)
x, y	program variables (Vars)
n	number literals
b	boolean expressions (BExp)