软件分析与架构设计

程序语义

何冬杰

重庆大学

语法（Syntax）

- **Grammar G： which programs are syntactically correct?**
  - Terminals Σ， Non-terminals N, Initial symbol $s \in N$, Productions P
  - Language $L(G)$: All sentences derived from s
- **Example：**

Arithmetic Expression

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

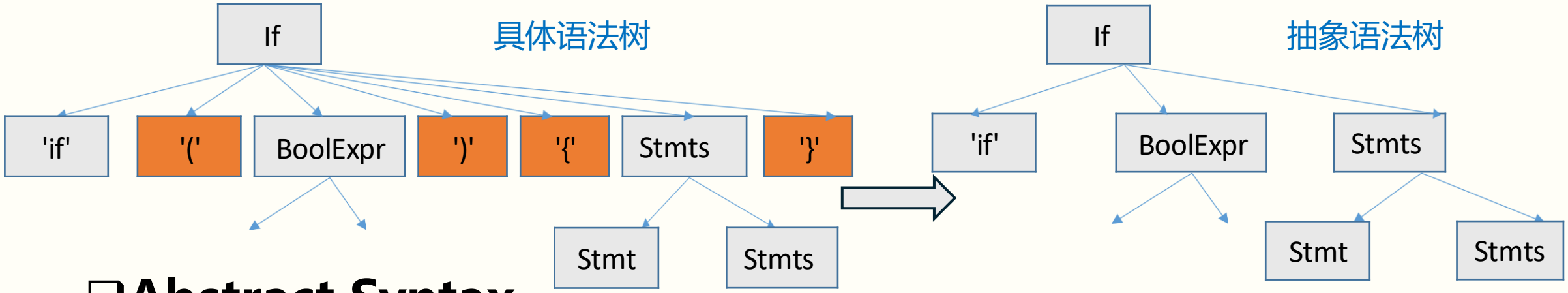
B. 2+(12-4)

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.

Abstract Syntax of SIMP

❑SIMP：simple imperative PL

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

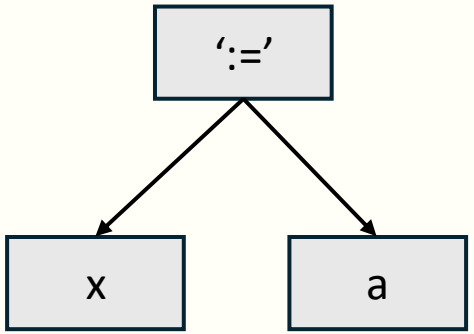
❑Meta-variables frequently used for easy of notation

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?

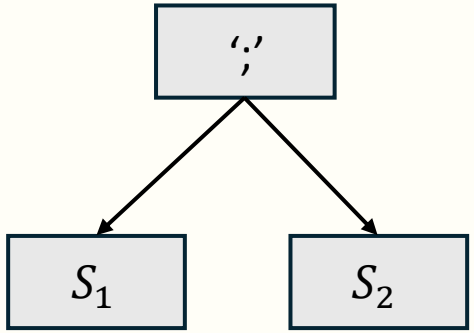
(1) x:=a



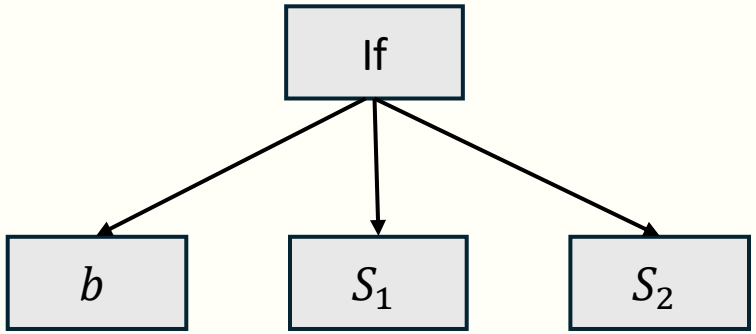
(2) skip

(do nothing)

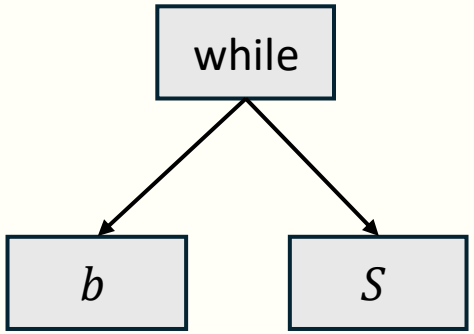
(3) $S_1; S_2$



(4) if b then S_1 else S_2



(5) while b do S

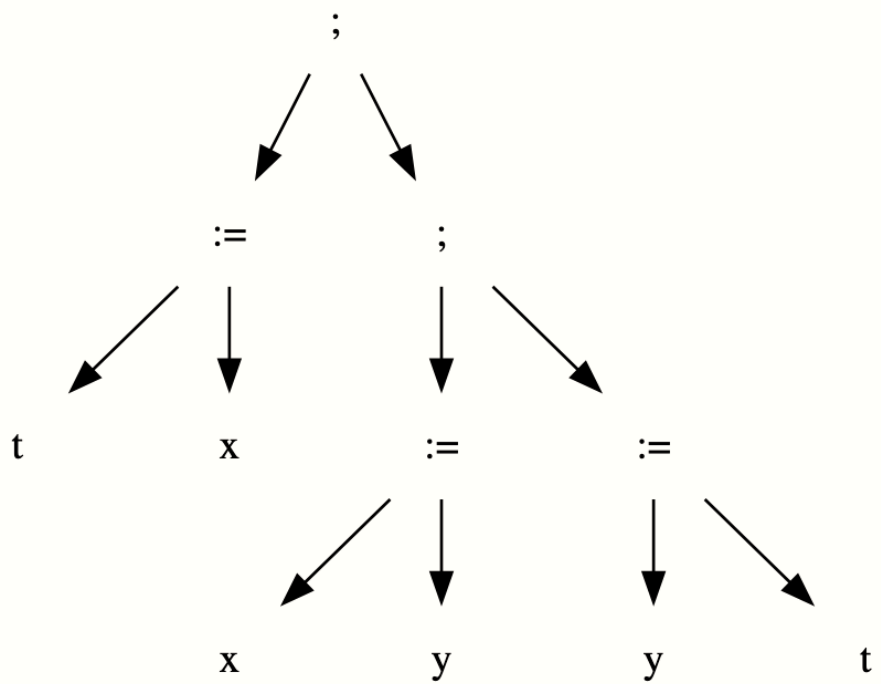


$$
\begin{array}{lcl}
S & ::= & x := a \\
 & | & \text{skip} \\
 & | & S_1;\ S_2 \\
 & | & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & | & \text{while } b \text{ do } S
\end{array}
$$

构建AST举例

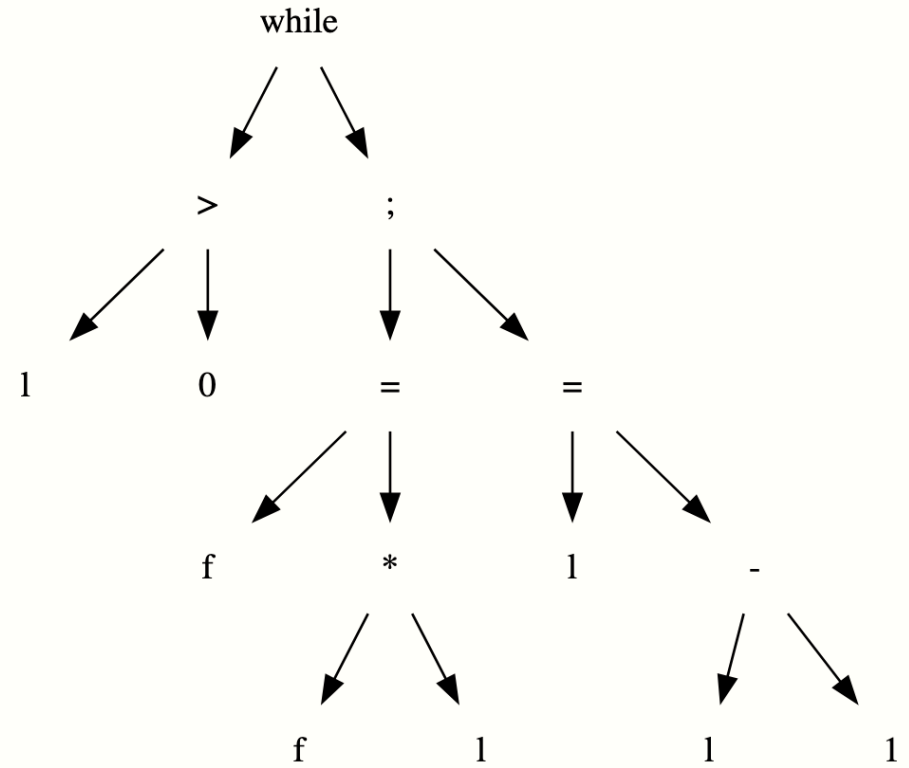
❑例1: 交换x和y的值

```
t = x; x = y; y = t
```



❑例2:计算阶乘

```
while 1 > 0 do f = f * 1 ; 1 = 1 -1
```



(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
```

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

(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

软件分析与架构设计

操作语义

何冬杰

重庆大学

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?**

```
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
    - 然后将表达式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

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

❑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** ⟹

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

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

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

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

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

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}?**

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

- ⊢ ⟨E1, x * 2 - 6⟩ ⇓ 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

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

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

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

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

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

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

small-assign处理类似 big-assign

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

$$\frac{\langle E, b\rangle \to_b b'}{\langle E, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rangle \to \langle E, \texttt{if}\ b'\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rangle}\ \textit{small-if-congruence}$$

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

small-iffalse处理类似

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

Small-Step Semantics: Example

- **Program p:** `t = x; x = y; y = t`
  - Environment E= $\{t \longmapsto 0, x \longmapsto 1, y \longmapsto 2\}$
- **Small-Step Semantics:**
  - $\langle E, p\rangle \to \cdots \to \langle\{t \longmapsto 1, x \longmapsto 2, y \longmapsto 1\}, skip\rangle$
  - Each step applies an axiom or an inference rule, form a **proof tree**
  - Excerpt of the proof tree:

$$\dfrac{\dfrac{\dfrac{}{\langle E, x\rangle \to \langle E, 1\rangle}\,(small-var)}{\langle E, t = x\rangle \to \langle E, t = 1\rangle}\,(small-assign)}{\langle E, p\rangle \to \langle E, t = 1;\ x = y;\ y = t\rangle}\,(small-seq)$$

Evaluation Sequence

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

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

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

❑3 possible outputs:

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

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 $\forall\, n\colon P(n)$ by induction on natural numbers
  - Base case: show that $P(0)$ holds
  - Inductive case: show that $\forall\, m\colon P(m) \rightarrow P(m+1)$

Proofs by Structural Induction

- **Prove $\forall\, a \in Aexp: P(a)$ by induction on structure of syntax**
  - Base cases: show that $P(x)$ and $P(n)$ holds
  - Inductive cases: show that
    - $\mathrm{P}(a_1) \wedge \mathrm{P}(a_2) \Rightarrow \mathrm{P}(a_1 + a_2)$
    - $\mathrm{P}(a_1) \wedge \mathrm{P}(a_2) \Rightarrow \mathrm{P}(a_1 - a_2)$
    - $\mathrm{P}(a_1) \wedge \mathrm{P}(a_2) \Rightarrow \mathrm{P}(a_1 * a_2)$
    - $\mathrm{P}(a_1) \wedge \mathrm{P}(a_2) \Rightarrow \mathrm{P}(a_1 / a_2)$

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

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

- **Let** $\mathrm{D} :: \langle \mathrm{E}, \mathrm{S}\rangle \Downarrow \mathrm{E}'$**, and** $\mathrm{D}' :: \langle \mathrm{E}, \mathrm{S}\rangle \Downarrow \mathrm{E}''$
- **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

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

    - $\mathrm{D}' :: \langle \mathrm{E}, \text{while b do S}\rangle \Downarrow \mathrm{E}''$ must have sub-derivations:
      - $D_1' :: \langle \mathrm{E}, \mathrm{b}\rangle \Downarrow \text{true}$, $D_2' :: \langle \mathrm{E}, \mathrm{S}\rangle \Downarrow E_1'$ and $D_3' :: \langle E_1', \text{while b do S}\rangle \Downarrow \mathrm{E}''$
    - By induction hypothesis, $E_1 = E_1'$ and $\mathrm{E}'' = \mathrm{E}'$
  - 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.

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

- Can be proved via structural induction over syntax

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

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

软件分析与架构设计

收集语义

何冬杰

重庆大学

三地址码（Three-Address）

❑Review SIMP Syntax

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

❑三地址码：一种编译器常用中间表示（IR）形式

- ➢3-address Syntax:
- ➢解决AST难以追踪程序执行时的数据流和控制流问题

Inst ::= $x := n \mid x := y \mid x := y\ op\ z \mid \text{goto } n \mid \text{if } x\ \text{op}_\text{r}\ 0\ \text{goto } n$

op_a ::= $+ \mid - \mid * \mid / \mid \ldots$

op_r ::= $< \mid \leq \mid = \mid > \mid \geq \mid \ldots$

P ∈ ***Num*** → ***Inst***

Inst: instructions
Num: number literals
x, y ∈ ***Var***: variables

更多3地址指令，后续会用

Inst ::= $\cdots \mid x := f(y) \mid \text{return } x \mid x := y.m(z) \mid \text{read } x \mid \text{print } x \mid x := \&p \mid x :=* p \mid *p := x \mid \text{x} := \text{y.f} \mid \text{x.f} := \text{y} \mid \text{halt}$

三地址码（Three-Address）

`w = x * y + z`

3地址码

```
1. t = x * y
2. w = t + z
```

`if b then S1 else S2`

3地址码

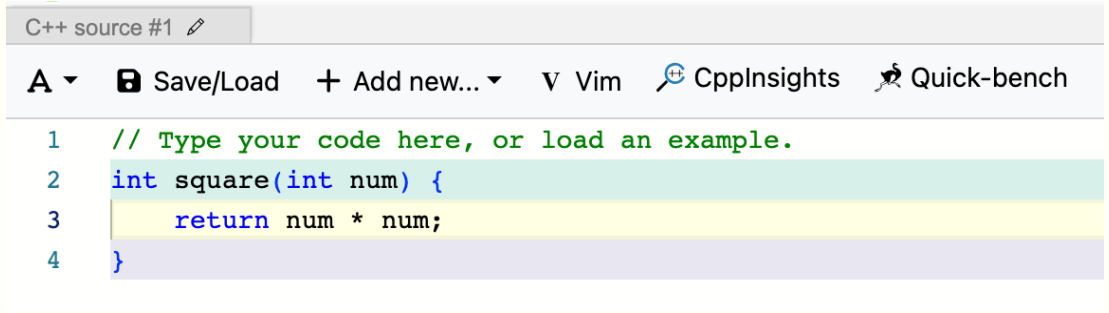
```
1. if b goto 4
2. S2
3. goto 5
4. S1
5. ...
```

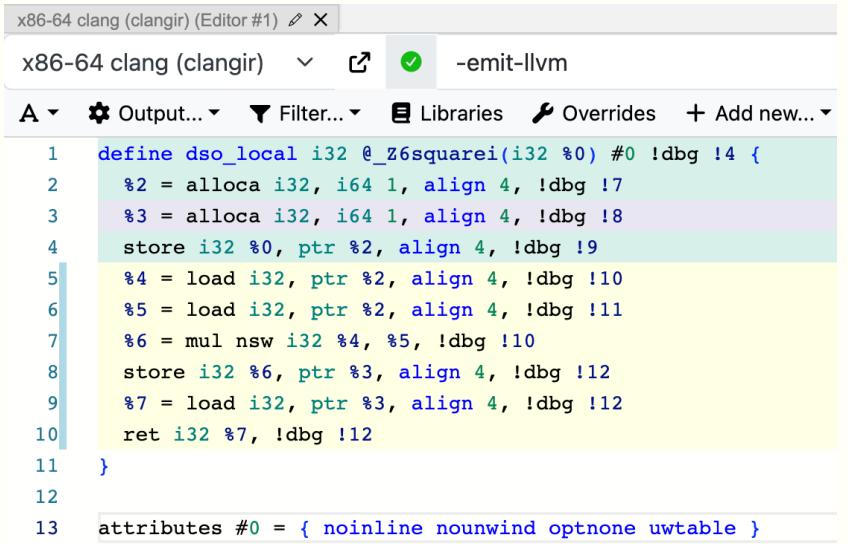
`while b do S`

3地址码

```
1. if !b then goto 3
2. S
3. ...
```

- **AST到3地址的转换是直接的、简单的**
  - Compiler Explorer：https://godbolt.org
  - (可选)作业：clang如何将AST转成LLVM-IR?





An Abstract Machine of 3-Address IR

- **Configuration (state):**
  - Environment
  - Program counter

$$c \in E \times \mathbb{N}$$

- **Program:**
  - Maps labels to instructions

$$\mathrm{P} \in \mathbb{N} \to \boldsymbol{Inst}$$

- **Transition or Execution:**
  - Small-step Semantics

$$P \vdash \langle E, n \rangle \rightsquigarrow \langle E', n' \rangle$$

$$\boldsymbol{Inst} ::= x := n \mid x := y \mid x := y\ op\ z \mid \text{goto } n \mid \text{if } x\ \text{op}_\text{r}\ 0 \text{ goto } n$$

$$\frac{P(n) = x := m}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E[x \mapsto m], n+1 \rangle}\ \textit{step-const}$$

$$\frac{P[n] = x := y}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E[x \mapsto E(y)], n+1 \rangle}\ \textit{step-copy}$$

$$\frac{P(n) = x := y\ op\ z \quad E(y)\ \mathbf{op}\ E(z) = m}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E[x \mapsto m], n+1 \rangle}\ \textit{step-arith}$$

$$\frac{P(n) = \text{goto } m}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E, m \rangle}\ \textit{step-goto}$$

$$\frac{P(n) = \text{if } x\ op_r\ 0 \text{ goto } m \quad E(x)\ \mathbf{op_r}\ 0 = true}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E, m \rangle}\ \textit{step-iftrue}$$

$$\frac{P(n) = \text{if } x\ op_r\ 0 \text{ goto } m \quad E(x)\ \mathbf{op_r}\ 0 = false}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E, n+1 \rangle}\ \textit{step-iffalse}$$

收集语义（Collecting Semantics）

The **collecting semantics** is used to define a collection of all the states (defined by operational semantics) of the program at a given point.

More information is available at here

❑零分析 (zero analysis):

- ➢Determine whether the value of a variable is 0
- ➢Reachable States: $\mathbb{E} = \{\mathrm{E} \mid \langle \mathrm{E}, n \rangle \text{ is a state}\}$, n is a node
- ➢Collecting semantics: $C_{\mathrm{ZA}}(\mathrm{x}) = \{\mathrm{i} \mid \exists \mathrm{E} \in \mathbb{E}, \text{such that } \mathrm{E}(\mathrm{x}) = \mathrm{i}\}$
- ➢Given a variable x : if $C_{\mathrm{ZA}}(\mathrm{x})$ = {0}, the value of x is zero

```
1. x = 10
2. y = 2 * x
3. z = 0
4. if y > 0 goto 6
5. z = 2 * x - y
6. ...
```

$x \mapsto \{10\}$
$\mathrm{x} \mapsto \{10\}, \mathrm{y} \mapsto \{20\}$
$\mathrm{x} \mapsto \{10\}, \mathrm{y} \mapsto \{20\}, \mathrm{z} \mapsto \{0\}$
$\mathrm{x} \mapsto \{10\}, \mathrm{y} \mapsto \{20\}, \mathrm{z} \mapsto \{0\}$
$\mathrm{x} \mapsto \{10\}, \mathrm{y} \mapsto \{20\}, \mathrm{z} \mapsto \{0\}$
$\mathrm{x} \mapsto \{10\}, \mathrm{y} \mapsto \{20\}, \mathrm{z} \mapsto \{0\}$

收集语义（Collecting Semantics）

The **collecting semantics** is used to define a collection of all the states (defined by operational semantics) of the program at a given point.

❑达到定值 (reaching definitions)

- Extending environment E (called E_{RD}): $E_{RD} \in Var \to \mathbb{Z} \times \mathbb{N}$
- $\mathrm{x} \mapsto \mathrm{v}, \mathrm{n}$: indicating that x was last defined as v at the location n

Collecting Semantics for Reaching definitions

$$\frac{P[n] = x := m}{P \vdash E, n \rightsquigarrow E[x \mapsto m, n], n+1} \textit{ step-const}$$

$$\frac{P[n] = x := y}{P \vdash E, n \rightsquigarrow E[x \mapsto E[y], n], n+1} \textit{ step-copy}$$

New rules

$$\frac{P[n] = x := y \; op \; z \quad E[y] \; \mathbf{op} \; E[z] = m}{P \vdash E, n \rightsquigarrow E[x \mapsto m, n], n+1} \textit{ step-arith}$$

$$\frac{P(n) = \text{goto } m}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E, m \rangle} \textit{ step-goto}$$

$$\frac{P(n) = \text{if } x \; op_r \; 0 \text{ goto } m \quad E(x) \; \mathbf{op_r} \; 0 = true}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E, m \rangle} \textit{ step-iftrue}$$

$$\frac{P(n) = \text{if } x \; op_r \; 0 \text{ goto } m \quad E(x) \; \mathbf{op_r} \; 0 = false}{P \vdash \langle E, n \rangle \rightsquigarrow \langle E, n+1 \rangle} \textit{ step-iffalse}$$

```
1. x = 3
2. t = x - 2
3. if t > 0 goto 5
4. x = 4
5. y = x + 5
```

$\alpha(E, 4) = \{2,4\}$
$\alpha(E, 5) = \{1,2,4,5\}$

- Abstract function: $\alpha_{RD}(E_{RD}, n) = \{m \mid \exists x \in domain(E_{RD}) \text{ such that } E_{RD}(x) = i, m\}$

m处定义的变量可到达节点n

(optional)作业：收集语义练习

❑计算变量(x, y)在每个程序点的可能收集状态

```
(1) x := 0
(2) y := 0
(3) if (x = 0) then
(4)     y := 1
(5) else
(6)     y := 2
(7) x := y
```

Program Point	说明
entry	程序入口
after (1)	执行 x := 0
after (2)	执行 y := 0
then-branch (4)	then 分支
else-branch (6)	else 分支
after if	分支合并
after (7)	程序结束