Formal Syntax and Semantics of Programming Languages

Mooly Sagiv

Reference: Semantics with Applications

Chapter 2

H. Nielson and F. Nielson

http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html

The **While** Programming Language

- Abstract syntax
  S::= x := a | **skip** | S_1 ; S_2 | **if** b **then** S_1 **else** S_2 | **while** b do S
- Use parenthesizes for precedence
- Informal Semantics
  - **skip** behaves like no-operation
  - Import meaning of arithmetic and Boolean operations

Natural Semantics for While

axioms

$[ass_{ns}]\ \langle x := a, s\rangle \rightarrow s[x \mapsto \mathbf{A}[\![a]\!]s]$

$[skip_{ns}]\ \langle \mathbf{skip}, s\rangle \rightarrow s$

$[while^{ff}_{ns}]\ \langle \text{while } b \text{ do } S, s\rangle \rightarrow s \quad \text{if } \mathbf{B}[\![b]\!]s=ff$

rules

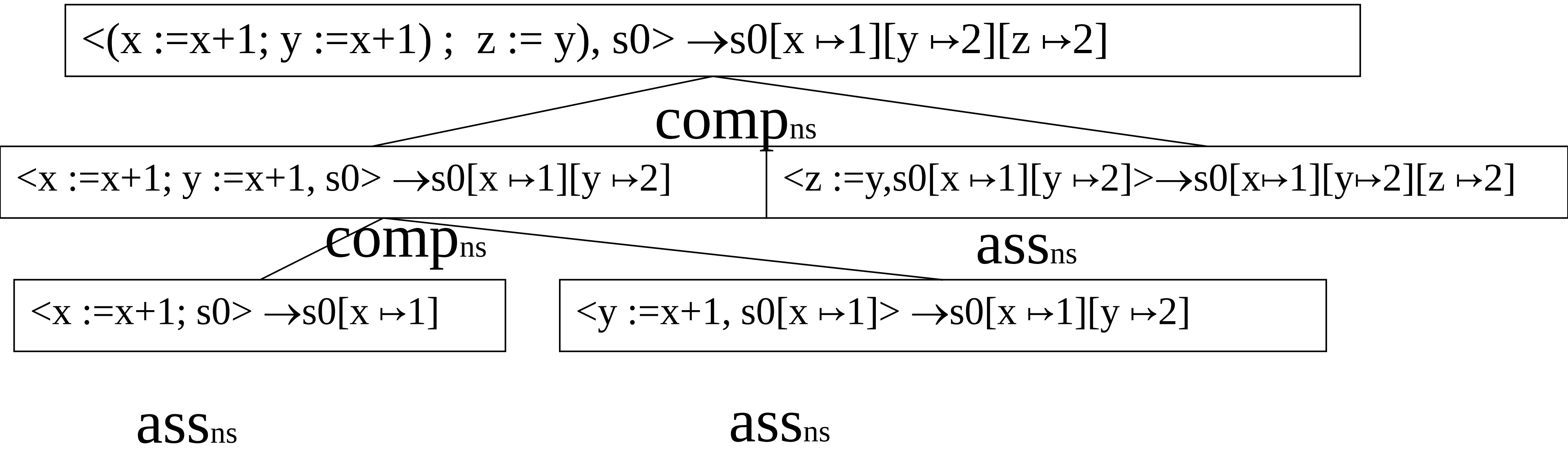
$$[comp_{ns}]\ \frac{\langle S_1 , s\rangle \rightarrow s',\ \langle S_2, s'\rangle \rightarrow s''}{\langle S_1; S_2, s\rangle \rightarrow s''}$$

$$[if^{tt}_{ns}]\ \frac{\langle S_1 , s\rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \rightarrow s'} \quad \text{if } \mathbf{B}[\![b]\!]s=tt$$

$$[if^{ff}_{ns}]\ \frac{\langle S_2 , s\rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \rightarrow s'} \quad \text{if } \mathbf{B}[\![b]\!]s=ff$$

$$[while^{tt}_{ns}]\ \frac{\langle S , s\rangle \rightarrow s',\ \langle \text{while } b \text{ do } S, s'\rangle \rightarrow s''}{\langle \text{while } b \text{ do } S, s\rangle \rightarrow s''} \quad \text{if } \mathbf{B}[\![b]\!]s=tt$$

An Example Derivation Tree



Top Down Evaluation of Derivation Trees

- Given a program S and an input state s
- Find an output state s' such that

  $\langle S, s \rangle \rightarrow s'$
- Start with the root and repeatedly apply rules until the axioms are reached
- Inspect different alternatives in order
- In While s' and the derivation tree is unique

Semantic Equivalence

- S_1 and S_2 are semantically equivalent if for all s and s'
  $<S_1, s> \to s'$ if and only if $<S_2, s> \to s'$
- Simple example
  "while b do S"
  is semantically equivalent to:
  "if b then (S ; while b do S) else skip"

Deterministic Semantics for While

- If $<S, s> \to s_1$ and $<S, s> \to s_2$ then $s_1=s_2$
- The proof uses induction on the shape of derivation trees
  - Prove that the property holds for all simple derivation trees by showing it holds for axioms
  - Prove that the property holds for all composite trees:
    - For each rule assume that the property holds for its premises (induction hypothesis) and prove it holds for the conclusion of the rule

The Semantic Function S_{ns}

- The meaning of a statement S is defined as a partial function from **State** to **State**
- S_{ns}: **Stm** → **(State** ↪ **State)**
- S_{ns} ⟦S⟧s = s' if <S, s> → s' and otherwise S_{ns} ⟦S⟧s is undefined
- Examples
  - S_{ns} ⟦skip⟧s =s
  - S_{ns} ⟦x :=1⟧s = s [x ↦1]
  - S_{ns} ⟦while true do skip⟧s = undefined

Structural Operational Semantics

- Emphasizes the individual execution steps
- $<S, i> \Rightarrow \gamma$
  - If the "**first**" step of executing the statement S on an input state *i* leads to γ
- Two possibilities for γ
  - $\gamma = <S', s'>$
    - The execution of S is not completed, S' is the remaining computation which need to be performed on s'
  - $\gamma = o$
    - The execution of S has terminated with a final state o
- γ is a stuck configuration when there are no transitions
- The meaning of a program P on an input state s is the set of final states that can be executed in arbitrary finite steps

Structural Semantics for While

axioms

$[\text{ass}_{sos}]\ \langle x := a, s\rangle \Rightarrow s[x \mapsto \mathbf{A}[\![a]\!]s]$

$[\text{skip}_{sos}]\ \langle \mathbf{skip}, s\rangle \Rightarrow s$

rules

$$[\text{comp}^1_{sos}]\ \frac{\langle S_1 , s\rangle \Rightarrow \langle S'_1, s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S'_1; S_2, s'\rangle}$$

$$[\text{comp}^2_{sos}]\ \frac{\langle S_1 , s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

Structural Semantics for While
if construct

$[\text{if}^{tt}_{sos}]$ $\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \Rightarrow \langle S_1, s\rangle$ if $\mathbf{B}[\![b]\!]s = tt$

$[\text{if}^{ff}_{os}]$ $\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \Rightarrow \langle S_2, s\rangle$ if $\mathbf{B}[\![b]\!]s = ff$

Structural Semantics for While
while construct

[while$_{sos}$] <while b do S, s> $\Rightarrow$
<if b then (S; while b do S) else skip, s>

Structural Semantics for While (Summary)

axioms

$[\text{ass}_{\text{sos}}]\ \langle x := a, s\rangle \Rightarrow s[x \mapsto \mathbf{A}[\![a]\!]s]$

$[\text{skip}_{\text{sos}}]\ \langle \mathbf{skip}, s\rangle \Rightarrow s$

$[\text{if}^{tt}_{\text{sos}}]\ \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \Rightarrow \langle S_1, s\rangle \quad \text{if } \mathbf{B}[\![b]\!]s = tt$

$[\text{if}^{ff}_{\text{sos}}]\ \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \Rightarrow \langle S_2, s\rangle \quad \text{if } \mathbf{B}[\![b]\!]s = ff$

rules

$[\text{while}_{\text{sos}}]\ \langle \text{while } b \text{ do } S, s\rangle \Rightarrow$
$\langle \text{if } b \text{ then } (S; \text{ while } b \text{ do } S) \text{ else skip}, s\rangle$

$$[\text{comp}^1_{\text{sos}}]\ \frac{\langle S_1 , s\rangle \Rightarrow \langle S'_1, s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S'_1; S_2, s'\rangle}$$

$$[\text{comp}^2_{\text{sos}}]\ \frac{\langle S_1 , s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

Example

- $S=[x \mapsto 5 , y \mapsto 7]$
- $S = (z:=x; x := y); y := z$

$$\dfrac{\dfrac{z:= x , [x\mapsto 5, y \mapsto 7] \Rightarrow [x\mapsto 5, y \mapsto 7, z \mapsto 5] \quad ass_{sos}}{z:= x; x := y , [x\mapsto 5, y \mapsto 7] \Rightarrow x := y ; y := z, [x\mapsto 5, y \mapsto 7, z \mapsto 5]} comp^{2}_{sos}}{(z:=x; x := y); y := z, [x\mapsto 5, y \mapsto 7] \Rightarrow x := y ; y := z, [x\mapsto 5, y \mapsto 7, z \mapsto 5]} comp^{1}_{sos}$$

Example (2^{nd} step)

- S=[x $\mapsto$ 5 , y $\mapsto$ 7]
- S = (z:=x; x := y); y := z

$$\text{comp}^2_{\text{sos}}\ \frac{x := y;\ y := z,\ [x\mapsto 5,\ y \mapsto 7,\ z \mapsto 5] \Rightarrow\ y := z,\ [x\mapsto 7,\ y \mapsto 7,\ z \mapsto 5]}{x := y\ ,\ [x\mapsto 5,\ y \mapsto 7,\ z \mapsto 5] \Rightarrow x := y\ ;\ y := z,\ [x\mapsto 7,\ y \mapsto 7,\ z \mapsto 5]}$$

ass_{sos}

Example (3^{rd} step)

- S=[x $\mapsto$ 5 , y $\mapsto$ 7]
- S = (z:=x; x := y); y := z

y := z [x$\mapsto$7, y $\mapsto$7, z $\mapsto$ 5] => y := z, [x$\mapsto$7, y $\mapsto$5, z $\mapsto$ 5]

ass_{sos}

Factorial Program

- Input state s such that s x = 3

$y := 1;\ \text{while}\ \neg(x=1)\ \text{do}\ (y := y * x;\ x := x - 1)$

$<y := 1 ; W, s>$

$\Rightarrow <W, s[y \mapsto 1]>$

$\Rightarrow <\text{if}\ \neg\ (x = 1)\ \text{then}\ (y := y * x ; x := x - 1\ \text{else skip}); W), s[y \mapsto 1]>$

$\Rightarrow <((y := y * x ; x := x - 1); W), s[y \mapsto 1]>$

$\Rightarrow <(x := x - 1 ; W), s[y \mapsto 3]>$

$\Rightarrow < W , s[y \mapsto 3][x \mapsto 2]>$

$\Rightarrow <\text{if}\ \neg\ (x = 1)\ \text{then}\ ((y := y * x ; x := x - 1); W)\ \text{else skip}, s[y \mapsto 3][x \mapsto 2]>$

$\Rightarrow <((y := y * x ; x := x - 1); W), s[y \mapsto 3] [x \mapsto 2] >$

$\Rightarrow <(x := x - 1 ; W) , s[y \mapsto 6] [x \mapsto 2] >$

$\Rightarrow < W, s[y \mapsto 6][x \mapsto 1]>$

$\Rightarrow <\text{if}\ \neg\ (x = 1)\ \text{then}\ (y := y * x ; x := x - 1); W)\ \text{else skip}, s[y \mapsto 6][x \mapsto 1]>$

$\Rightarrow <\text{skip}, s[y \mapsto 6][x \mapsto 1]> \Rightarrow s[y \mapsto 6][x \mapsto 1]$

Finite Derivation Sequences

- finite derivation sequence starting at <S, i>

  $\gamma_0, \gamma_1, \gamma_2 \ldots, \gamma_k$ such that

  - γ_0=<S, i>
  - $\gamma_i \Rightarrow \gamma_{i+1}$
  - γ_k is either stuck configuration or a final state
- For each step there is a derivation tree
- $\gamma_0 \Rightarrow^k \gamma_k$ in k steps
- $\gamma_0 \Rightarrow^* \gamma$ in finite number of steps

Infinite Derivation Sequences

- An infinite derivation sequence starting at <S, i>

  $\gamma_0, \gamma_1, \gamma_2$ … such that
  - γ_0=<S, i>
  - $\gamma_i \Rightarrow \gamma_{i+1}$
- Example
  - S = while true do skip
  - s_0 x = 0

Program Termination

- Given a statement S and input s
  - S terminates on s if there exists a finite derivation sequence starting at <S, s>
  - S terminates successfully on s if there exists a finite derivation sequence starting at <S, s> leading to a final state
  - S loops on s if there exists an infinite derivation sequence starting at <S, s>

Properties of the Semantics

- S_1 and S_2 are semantically equivalent if:
  - for all s and γ which is either final or stuck $<S_1, s> \Rightarrow^* \gamma$ if and only if $<S_2, s> \Rightarrow^* \gamma$
  - there is an infinite derivation sequence starting at $<S_1, s>$ if and only if there is an infinite derivation sequence starting at $<S_2, s>$
- Deterministic
  - If $<S, s> \Rightarrow^* s_1$ and $<S, s> \Rightarrow^* s_2$ then $s_1=s_2$
- The execution of $S_1; S_2$ on an input can be split into two parts:
  - execute S_1 on s yielding a state s'
  - execute S_2 on s'

Sequential Composition

- If $<S_1; S_2, s> \Rightarrow^k s''$ then there exists a state s' and numbers k_1 and k_2 such that
  - $<S_1, s> \Rightarrow^{k1} s'$
  - $<S_2, s'> \Rightarrow^{k2} s''$
  - and $k = k_1 + k_2$
- The proof uses induction on the length of derivation sequences
  - Prove that the property holds for all derivation sequences of length 0
  - Prove that the property holds for all other derivation sequences:
    - Show that the property holds for sequences of length k+1 using the fact it holds on all sequences of length k (induction hypothesis)

The Semantic Function S_{sos}

- The meaning of a statement S is defined as a partial function from **State** to **State**
- S_{sos}: **Stm** $\rightarrow$ **(State** $\hookrightarrow$ **State)**
- $S_{sos}[\![S]\!]s = s'$ if $<S, s> \Rightarrow^* s'$ and otherwise $S_{sos}[\![S]\!]s$ is undefined

An Equivalence Result

- For every statement S of the While language
  - $S_{nat}[\![S]\!] = S_{sos}[\![S]\!]$

Extensions to While

- Abort statement (like C exit w/o return value)
- Non determinism
- Parallelism
- Local Variables
- Procedures
  - Static Scope
  - Dynamic scope

The **While** Programming Language with Abort

- Abstract syntax
  S::= x := a | **skip** | S_1 ; S_2 | **if** b **then** S_1 **else** S_2 |
  while b do S| **abort**
- Abort terminates the execution
- No new rules are needed in natural and structural operational semantics
- Statements
  - if x = 0 then abort else y := y / x
  - skip
  - abort
  - while true do skip

Examples

- <if x = 0 then abort else y := y / x, s> s →
  if s x = 0 then undefined else s [y ↦ s y / sx]
- <skip, s> → s
- For no s: <abort, s>→ s
- For no s: <while b do skip, s>→ s

Undefined semantics in C

- Pointer dereferences

```
x = *p; “≈” if (p !=NULL)  x = *p; else abort;
```

- Pointer arithmetic

```
x = a[i]; “≈” if (i <alloc(a)) x = *(a+i); else abort;
```

- Structure boundaries

Undefined semantics in Java?

- What about exceptions?

Pros and Cons of PLs with Undefined Semantics

Benefits

- Performance
- Expressive power
- Simplicity of the programming language

Disadvantages

- Security
- Portability
- Predictability
- Programmer productivity

Formulating Undefined semantics

- A programming language is type safe if correct programs cannot go wrong
- No undefined semantics
  - But runtime exceptions are fine
- For every program P
  - For every input state s one of the following holds:
    - $<P, s> \Rightarrow^{*} s'$ for some final state s'
    - $<P, s> \Rightarrow^{i} \gamma$ for all i
- While is type safe and while+abort is not

Conclusion

- The natural semantics cannot distinguish between looping and abnormal termination (unless the states are modified)
- In the structural operational semantics looping is reflected by infinite derivations and abnormal termination is reflected by stuck configuration

The **While** Programming Language with Non-Determinism

- Abstract syntax
  S::= x := a | **skip** | S_1 ; S_2 | **if** b **then** S_1 **else** S_2 | **while** b do S| S_1 or S_2
- Either S_1 or S_2 is executed
- Example
  - x := 1 or (x :=2 ; x := x+2)

The While Programming Language with Non-Determinism Natural Semantics

$$[or^1_{ns}]\ \frac{\langle S_1, s\rangle \rightarrow s'}{\langle S_1 \text{ or } S_2, s\rangle \rightarrow s'}$$

$$[or^2_{ns}]\ \frac{\langle S_2, s\rangle \rightarrow s'}{\langle S_1 \text{ or } S_2, s\rangle \rightarrow s'}$$

The While Programming Language with Non-Determinism Structural Semantics

The While Programming Language with Non-Determinism Examples

- x := 1 or (x :=2 ; x := x+2)
- (while true do skip) or (x :=2 ; x := x+2)

Conclusion

- In the natural semantics non-determinism will suppress looping if possible (mnemonic)
- In the structural operational semantics non-determinism does not suppress not termination configuration

The **While** Programming Language with Parallel Constructs

- Abstract syntax
  S::= x := a | **skip** | S_1 ; S_2 | **if** b **then** S_1 **else** S_2 |
  while b do S| S_1 par S_2
- All the interleaving of S_1 or S_2 are executed
- Examples
  - x := 1 par (x :=2 ; x := x+2)
  - (x := 1; a :=y) par (y := 1; b :=x)

The **While** Programming Language with Parallel Constructs Structural Semantics

$$[\text{par}^1_{\text{sos}}]\ \frac{\langle S_1, s\rangle \Rightarrow \langle S'_1, s'\rangle}{\langle S_1 \text{ par } S_2, s\rangle \Rightarrow \langle S'_1 \text{par } S_2, s'\rangle}$$

$$[\text{par}^2_{\text{sos}}]\ \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1 \text{ par } S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

$$[\text{par}^3_{\text{sos}}]\ \frac{\langle S_2, s\rangle \Rightarrow \langle S'_2, s'\rangle}{\langle S_1 \text{ par } S_2, s\rangle \Rightarrow \langle S_1 \text{par } S'_2, s'\rangle}$$

$$[\text{par}^4_{\text{sos}}]\ \frac{\langle S_2, s\rangle \Rightarrow s'}{\langle S_1 \text{ par } S_2, s\rangle \Rightarrow \langle S_1, s'\rangle}$$

The **While** Programming Language with Parallel Constructs Natural Semantics

Conclusion

- In the natural semantics immediate constituent is an atomic entity so we cannot express interleaving of computations
- In the structural operational semantics we concentrate on small steps so interleaving of computations can be easily expressed

The **While** Programming Language with local variables

- Abstract syntax

  S::= x := a | **skip** | S_1 ; S_2 | **if** b **then** S_1 **else** S_2 |
  while b do S| **{** L S }

  L ::= **var** x := a ; L | ε

Simple Example

```
{
  var y := 1;
  (var x := 2 ;
      {
          var x := 3 ;
          y :=  x + y // 4
      }
   x := y + x   // 6
)
}
```

Another Example

```
while (y > 0) (
    {
       var x := y ;
        y :=  x + y;
        y : = y -1
     }
  x := y + x
```

Natural Semantics

$$\mathrm{LHS} : L \to 2^{\mathrm{Var}}$$

$$\mathrm{LHS}(\varepsilon) = \varnothing$$

$$\mathrm{LHS}(\textbf{var}\ x := a\ ;\ L) = \{x\} \cup \mathrm{LHS}(L)$$

$$s[X \mapsto s'] = \begin{cases} s\ x & \text{if } x \notin X \\ s'\ x & \text{if } x \in X \end{cases}$$

$$[\mathrm{none}_{ns}]\ \langle \varepsilon, s\rangle \to s$$

$$[\mathrm{var}_{ns}]\ \frac{\langle L\ ,\ s[x \mapsto \mathbf{A}[\![a]\!]s]\rangle \to s'}{\langle \mathrm{var}\ x := a;\ L,\ s\rangle \to s'}$$

$$[\mathrm{block}_{ns}]\ \frac{\langle L\ ,\ s\rangle \to s',\ \langle S,\ s'\rangle \to s''}{\langle \{\ L\ S\ \},\ s\rangle \to s''[\mathrm{LHS}(L) \mapsto s]}$$

Simple Example

if (y > 0)

then

{

var x := y +1;

y := x + y

}

else skip ;

y := y +x

<if (y > 0) …; y := y +x), [x ↦8, y ↦5] > →[y ↦17, x ↦6]

comp$_{ns}$

<if (y>0) …, [x ↦8, y ↦5] > → [y ↦11, x ↦6]

<y := y +x, [y ↦11, x ↦6] → [y ↦17, x ↦6]

[if$^{tt}_{ns}$]

<{ var x := y +1; y := x+ y } , [x ↦8, y ↦5]>→[y ↦11, x ↦6]

[block$_{ns}$]

<var x := y +1;, [x ↦8, y ↦5] >→ [y ↦5, x ↦6]

<y := x+y,[y ↦5, x ↦6]>→ [y ↦11, x ↦6]

[var$_{ns}$]

<var x := y +1; , [x ↦8, y ↦5] >→[y ↦5, x ↦6]

[none$_{ns}$]

Structural Semantics

$$[\text{block}_{sos}]\ \frac{?}{\langle \text{begin } D_v \text{ S end, s} \rangle \Rightarrow s'}$$

Conclusions Local Variables

- The natural semantics can “remember” local states
- Need to introduce stack or heap into state of the structural semantics

The **While** Programming Language with local variables and procedures

- Abstract syntax

  S::= x := a | **skip** | S_1 ; S_2 | **if** b **then** S_1 **else** S_2 |
  while b do S|
  { L P S } **| call p**

  L ::= **var** x := a ; L | ε

  P ::= **proc** p **is** S ; P | ε

Summary

- SOS is powerful enough to describe imperative programs
  - Can define the set of traces
  - Can represent program counter implicitly
  - Handle gotos
- Natural operational semantics is an abstraction
- Different semantics may be used to justify different behaviors
- Thinking in concrete semantics is essential for language designer/compiler writer/…