TLA$^+$ Video Course – Lecture 2

Leslie Lamport

STATE MACHINES IN TLA$^+$

This video should be viewed in conjunction with a Web page.
To find that page, search the Web for *TLA+ Video Course* .

The TLA$^+$ Video Course
Lecture 2
STATE MACHINES IN MATH

In the first lecture, I introduced state machines as a simple abstraction of digital systems.

You saw how a tiny C program can be viewed as a state machine.

In this lecture, you will see how that state machine can be described mathematically, and you will get your first glimpse of TLA^{+}.

[slide 2]

WHAT LANGUAGE SHOULD WE USE ?

What language should we use to describe state machines?

State machines are a simple and powerful abstraction.

State machines are a simple and powerful abstraction.

State machines are a simple and powerful abstraction.

We need a precise, practical way to describe them.

State machines are a simple and powerful abstraction.

We need a precise, practical way to describe them.

State machines are a simple and powerful abstraction.

We need a precise, practical way to describe them.

This is neither precise nor practical:

if current value of pc equals "$start$"
  then next value of i in $\{0, 1, \ldots, 1000\}$
    next value of pc equals "$middle$"
  else **if** current value of pc equals "$middle$"
    then next value of i equals
      current value of $i + 1$
    next value of pc equals "$done$"
    else no next values

State machines are a simple and powerful abstraction.

We need a precise, practical way to describe them.

The way we described the next state for the simple program
is neither precise nor is it practical for real systems .

[slide 6]

We need a language for describing state machines.

We need a precise language for describing state machines.

Asked what such a language should look like,

We need a language for describing state machines.

Most software engineers want one like their favorite programming language.

We need a precise language for describing state machines.

Asked what such a language should look like, most programmers and software engineers want one that's a lot like their favorite programming language.

We need a language for describing state machines.

Most software engineers want one like their favorite programming language.

TLA^{+}

We need a precise language for describing state machines.

Asked what such a language should look like, most programmers and software engineers want one that's a lot like their favorite programming language.

TLA+ takes a different approach.

We need a language for describing state machines.

Most software engineers want one like their favorite programming language.

TLA$^+$ uses ordinary, simple math.

We need a precise language for describing state machines.

Asked what such a language should look like, most programmers and software engineers want one that's a lot like their favorite programming language.

TLA+ takes a different approach. It uses ordinary, simple math.

We need a language for describing state machines.

Most software engineers want one like their favorite programming language.

TLA$^+$ uses ordinary, simple math.

Most software engineers find that a terrible and terrifying idea.

We need a precise language for describing state machines.

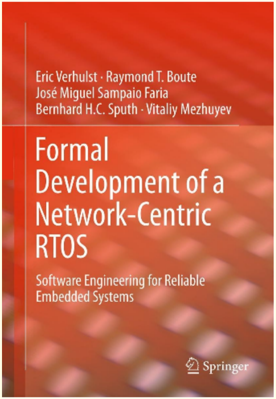
Asked what such a language should look like, most programmers and software engineers want one that's a lot like their favorite programming language.

TLA+ takes a different approach. It uses ordinary, simple math.

This strikes most programmers and software engineers as a terrible idea–and probably a terrifying one.

[slide 11]

Here’s what the designers of this real-time operating system



Here’s what the designers of this real-time operating system

Here's what the designers of this
real-time operating system said
in this paper:

An industrial Case: Pitfalls and Benefits of Applying Formal Methods to the Development of a Network-Centric RTOS

Eric Verhulst, Gjalt de Jong, and Vitaliy Mezhuyev

Formal Methods 2008, pages 411 – 418

Here's what the designers of this real-time operating system

said in this paper:

While we had an initial bias toward using language X,

While we had an initial bias toward using language X,

I’m not going to tell you what that language was

While we had an initial bias toward using language X,

in the end it was decided to use TLA$^+$.

While we had an initial bias toward using language X,

I'm not going to tell you what that language was

in the end it was decided to use TLA$^+$.

While we had an initial bias toward using language X, in the end it was decided to use TLA$^+$. Although the mathematical notation of the TLA$^+$ language was first considered a hindrance versus the C-like language X,

While we had an initial bias toward using language X,

I'm not going to tell you what that language was

in the end it was decided to use TLA$^+$.

Although the mathematical notation of the TLA$^+$ language was first considered a hindrance versus the C-like language X,

[slide 16]

While we had an initial bias toward using language X, in the end it was decided to use TLA+. Although the mathematical notation of the TLA+ language was first considered a hindrance versus the C-like language X, in the end it has proven to be a major benefit

in the end it has proven to be a major benefit

not a hindrance, a major benefit

While we had an initial bias toward using language X, in the end it was decided to use TLA+. Although the mathematical notation of the TLA+ language was first considered a hindrance versus the C-like language X, in the end it has proven to be a major benefit as it forced us to reason in a much more abstract way about the system.

in the end it has proven to be a major benefit

not a hindrance, a major benefit

as it forced us to reason in a much more abstract way about the system.

While we had an initial bias toward using language X, in the end it was decided to use TLA$^+$. Although the mathematical notation of the TLA$^+$ language was first considered a hindrance versus the C-like language X, in the end it has proven to be a major benefit as it forced us to reason in a much more abstract way about the system.

in the end it has proven to be a major benefit

not a hindrance, a major benefit

as it forced us to reason in a much more abstract way about the system.

A more **abstract** way. *And remember*. . .

Remember what Brannon Batson said:

what Brannon Batson said.

Remember what Brannon Batson said:

The hard part of learning to write TLA$^+$ specs is learning to think abstractly about the system.

what Brannon Batson said.

The hard part of learning to write TLA$^+$ specs is learning to *think abstractly* about the system.

Remember what Brannon Batson said:

The hard part of learning to write TLA$^+$ specs is learning to think abstractly about the system.

Being able to think abstractly improves our design process.

what Brannon Batson said.

The hard part of learning to write TLA$^+$ specs is learning to *think abstractly* about the system.

Being able to *think abstractly* improves our design process.

Remember what Verhulst said:

And remember what Eric Verhulst, the leader of that real-time operating system project, said:

Remember what Verhulst said:

> We witnessed first hand the brain washing done by years of C programming.

And remember what Eric Verhulst, the leader of that real-time operating system project, said:

We witnessed first hand the brain washing done by years of C programming.

DESCRIBING A STATE MACHINE WITH MATH

Describing a state machine with math.

Our example C program:

$$
\begin{array}{l}
\texttt{int}\ i\ ;\\
\texttt{void}\ main()\\
\quad \{\ i = someNumber()\ ;\\
\qquad i = i + 1\ ;\\
\quad \}
\end{array}
$$

Remember our example C program.

Our example C program:

$$\texttt{int}\ i\ ;$$
$$\texttt{void}\ main()$$
$$\{\ i = someNumber()\ ;$$
$$i = i + 1\ ;$$
$$\}$$

We introduced pc to describe the control state.

Remember our example C program.

Recall that we introduced the variable pc to describe the control state.

Our example C program:

$$\begin{array}{ll} & \texttt{int}\ i\ ; \\ & \texttt{void}\ main() \\ pc = \text{“}start\text{”} & \{\ \boxed{i = someNumber()\ ;} \\ & \quad i = i + 1\ ; \\ & \} \end{array}$$

We introduced pc to describe the control state.

Remember our example C program.

Recall that we introduced the variable pc to describe the control state.

pc equals the string *start* means this is the next statement to be executed.

Our example C program:

$$
\begin{array}{ll}
 & \texttt{int}\ i\ ; \\
 & \texttt{void}\ main() \\
 & \quad \{\ i = someNumber()\ ; \\
pc = \text{“}middle\text{”} & \quad \boxed{i = i + 1\ ;} \\
 & \quad \}
\end{array}
$$

We introduced pc to describe the control state.

Remember our example C program.

Recall that we introduced the variable pc to describe the control state.

pc equals the string *start* means this is the next statement to be executed.

pc equals *middle* means control is here.

Our example C program:

$$
\begin{array}{l}
\texttt{int}\ i\ ; \\
\texttt{void}\ main() \\
\quad \{\ i = someNumber()\ ; \\
\quad\ \ i = i + 1\ ; \\
\quad \}\ \ pc = \text{“}done\text{”}
\end{array}
$$

We introduced pc to describe the control state.

Remember our example C program.

Recall that we introduced the variable pc to describe the control state.

pc equals the string *start* means this is the next statement to be executed.

pc equals *middle* means control is here.

and pc equals *done* when execution has terminated.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

To describe this program,

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

We must describe:

To describe this program, we must describe two things:

$$
\begin{array}{l}
\texttt{int}\ i\ ;\\
\texttt{void}\ main()\\
\quad \{\ i = someNumber()\ ;\\
\quad\ \ i = i + 1\ ;\\
\quad \}
\end{array}
$$

We must describe:

1. Possible initial values of variables.

To describe this program, we must describe two things:

The possible initial values of the variables.

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

We must describe:

1. Possible initial values of variables.
2. The relation between their values in the current state and their possible values in the next state.

To describe this program, we must describe two things:

The possible initial values of the variables.

And what the relation is between the values of the variables in the current state and their possible values in the next state.

$$\begin{array}{l} \texttt{int } i \,; \\ \texttt{void } main() \\ \quad \{\, i = someNumber() \,; \\ \quad\;\; i = i + 1 \,; \\ \quad \} \end{array}$$

We must describe:

1. Possible initial values of variables.
2. The relation between their values in the current state and their possible values in the next state.

To describe this program, we must describe two things:

The possible initial values of the variables.

And what the relation is between the values of the variables in the current state and their possible values in the next state.

Let's start with the initial values.

Possible initial values of variables.

Possible initial values of variables.

$i = 0$ and $pc = \text{“}start\text{”}$

These are the initial values. But we want a mathematical formula, so

Possible initial values of variables.

$$i = 0 \text{ and } pc = \text{“}start\text{”}$$

Must replace “and” by a mathematical operator.

These are the initial values. But we want a mathematical formula, so

we must replace *and* by a mathematical operator.

Possible initial values of variables.

$$i = 0 \text{ and } pc = \text{“}start\text{”}$$

Must replace “and” by a mathematical operator.

Written && in some programming languages.

That operator is written *ampersand ampersand* in some programming languages.

Possible initial values of variables.

$$i = 0 \ \wedge \ pc = \text{“}start\text{”}$$

Must replace “and” by a mathematical operator.

Written && in some programming languages.

Written $\wedge$ in mathematics.

That operator is written *ampersand ampersand* in some programming languages.

It’s written with this symbol in mathematics.

Possible initial values of variables.

$$(i = 0) \land (pc = \text{“}start\text{”})$$

Must replace “and” by a mathematical operator.

Written && in some programming languages.

Written $\land$ in mathematics.

Some unnecessary parentheses make it easier to read.

That operator is written *ampersand ampersand* in some programming languages.

It’s written with this symbol in mathematics.

Let’s add some unnecessary parentheses to make it easier to read.

2. The relation between their values in the current state and their possible values in the next state.

$$
\begin{array}{l}
\texttt{int}\ i\ ; \\
\texttt{void}\ main() \\
\quad \{\ i = someNumber()\ ; \\
\quad\ \ i = i + 1\ ; \\
\quad \}
\end{array}
$$

Now, let's describe the relation between the values of the variables in the current state and their possible values in the next state.

2. The relation between their values in the current state and their possible values in the next state.

```
int i ;
void main()
  { i = someNumber() ;
    i = i + 1 ;
  }
```

if current value of pc equals "$start$"
  then next value of i in $\{0, 1, \ldots, 1000\}$
    next value of pc equals "$middle$"
  else **if** current value of pc equals "$middle$"
    then next value of i equals
      current value of $i + 1$
    next value of pc equals "$done$"
    else no next values

Now, let's describe the relation between the values of the variables in the current state and their possible values in the next state.

Here's how I did it in the previous lecture.

if current value of pc equals "$start$"
  then next value of i in $\{0, 1, \ldots, 1000\}$
    next value of pc equals "$middle$"
  else **if** current value of pc equals "$middle$"
    then next value of i equals current value of $i + 1$
      next value of pc equals "$done$"
    else no next values

OK. Let's now write this in math.

if current value of pc equals "$start$"
  then next value of i in $\{0, 1, \ldots, 1000\}$
    next value of pc equals "$middle$"
  else **if** current value of pc equals "$middle$"
    then next value of i equals current value of $i + 1$
      next value of pc equals "$done$"
    else no next values

Let's write this in mathematics.

OK. Let's now write this in math.

if current value of pc equals "$start$"
then next value of i in $\{0, 1, \ldots, 1000\}$
next value of pc equals "$middle$"
else **if** current value of pc equals "$middle$"
then next value of i equals current value of $i + 1$
next value of pc equals "$done$"
else no next values

Let's write this in mathematics.

This requires some notation.

OK. Let's now write this in math.

This requires replacing words with some notation.

if current value of pc equals "$start$"
  then next value of i in $\{0, 1, \ldots, 1000\}$
    next value of pc equals "$middle$"
  else **if** current value of pc equals "$middle$"
    then next value of i equals current value of $i + 1$
      next value of pc equals "$done$"
    else no next values

OK. Let's now write this in math.

This requires replacing words with some notation.

First, let's get rid of "current value of"

if ~~current value of~~ pc equals "$start$"
  then next value of i in $\{0, 1, \ldots, 1000\}$
    next value of pc equals "$middle$"
  else **if** ~~current value of~~ pc equals "$middle$"
    then next value of i equals current value of $i + 1$
      next value of pc equals "$done$"
    else no next values

pc means *current value of* pc

OK. Let's now write this in math.

This requires replacing words with some notation.

First, let's get rid of "current value of"

We simply let pc mean the current value of pc
and let i mean the current value of i

[slide 48]

if ~~current value of~~ pc equals "$start$"

then next value of i in $\{0, 1, \ldots, 1000\}$

next value of pc equals "$middle$"

else **if** ~~current value of~~ pc equals "$middle$"

then next value of i equals ~~current value of~~ $i + 1$

next value of pc equals "$done$"

else no next values

pc means *current value of* pc

i means *current value of* i

OK. Let's now write this in math.

This requires replacing words with some notation.

First, let's get rid of "current value of"

We simply let pc mean the current value of pc
and let i mean the current value of i

if pc equals "$start$"

  then next value of i in $\{0, 1, \ldots, 1000\}$

    next value of pc equals "$middle$"

  else **if** pc equals "$middle$"

    then next value of i equals $i + 1$

      next value of pc equals "$done$"

    else no next values

Next, we get rid of "next value of"

if pc equals "$start$"
  then next value of i in $\{0, 1, \ldots, 1000\}$
    next value of pc equals "$middle$"
  else **if** pc equals "$middle$"
    then next value of i equals $i + 1$
      next value of pc equals "$done$"
    else no next values

Next, we get rid of "next value of"

if pc equals "$start$"

then ~~next value of~~ i in $\{0, 1, \ldots, 1000\}$

~~next value of~~ pc equals "$middle$"

else **if** pc equals "$middle$"

then ~~next value of~~ i equals $i + 1$

~~next value of~~ pc equals "$done$"

else no next values

pc' means *next value of* pc

i' means *next value of* i

Next, we get rid of "next value of"

by letting pc prime and i prime mean the next values of pc and i

if pc equals "$start$"
then i' in $\{0, 1, \ldots, 1000\}$
pc' equals "$middle$"
else **if** pc equals "$middle$"
then i' equals $i + 1$
pc' equals "$done$"
else no next values

Next, we get rid of "next value of"

by letting pc prime and i prime mean the next values of pc and i

And finally, we replace the word "equals" by an equal sign.

if pc equals "$start$"

  then i' in $\{0, 1, \ldots, 1000\}$

    pc' equals "$middle$"

  else **if** pc equals "$middle$"

    then i' equals $i + 1$

      pc' equals "$done$"

    else no next values

equals $\rightarrow$ =

Next, we get rid of "next value of"

by letting pc prime and i prime mean the next values of pc and i

And finally, we replace the word "equals" by an equal sign.

if pc = "$start$"

  then i' in $\{0, 1, \ldots, 1000\}$

    pc' = "$middle$"

  else **if** pc = "$middle$"

    then i' = $i + 1$

      pc' = "$done$"

    else no next values

Next, we get rid of "next value of"

by letting pc prime and i prime mean the next values of pc and i

And finally, we replace the word "equals" by an equal sign.

Whew!

[slide 55]

if $pc = \text{“}start\text{”}$
    then i' in $\{0, 1, \ldots, 1000\}$
        $pc' = \text{“}middle\text{”}$
    else **if** $pc = \text{“}middle\text{”}$
        then $i' = i + 1$
            $pc' = \text{“}done\text{”}$
        else no next values

It’s now easier to read.

It’s now a lot easier to read.

if pc = "$start$"
  then i' in $\{0, 1, \ldots, 1000\}$
    pc' = "$middle$"
  else **if** pc = "$middle$"
    then i' = $i + 1$
      pc' = "$done$"
    else no next values

It's now easier to read.

But it's not yet mathematics.

It's now a lot easier to read.

But it's not yet a mathematical formula.

if pc = "$start$"

    then i' in $\{0, 1, \ldots, 1000\}$

            pc' = "$middle$"

    else **if** pc = "$middle$"

            then i' = $i + 1$

                    pc' = "$done$"

            else no next values

in here means is an element of the set of integers from 0 to 1000.

is an element of the set

if pc = "$start$"
  then i' in $\{0, 1, \ldots, 1000\}$
    pc' = "$middle$"
  else **if** pc = "$middle$"
    then i' = $i + 1$
      pc' = "$done$"
    else no next values

in here means is an element of the set of integers from 0 to 1000.

is an element of the set

if pc = "$start$"
  then $i' \in \{0, 1, \ldots, 1000\}$
    pc' = "$middle$"
  else **if** pc = "$middle$"
    then $i' = i + 1$
      pc' = "$done$"
    else no next values

Written in math as $\in$.

in here means is an element of the set of integers from 0 to 1000.

In is written in mathematics as this symbol.

if $pc = \text{"}start\text{"}$

  then $i' \in \{0, 1, \ldots, 1000\}$

    $pc' = \text{"}middle\text{"}$

  else **if** $pc = \text{"}middle\text{"}$

    then $i' = i + 1$

      $pc' = \text{"}done\text{"}$

    else no next values

in here means is an element of the set of integers from 0 to 1000.

In is written in mathematics as this symbol.

if $pc = \text{“}start\text{”}$

then $i' \in \{0, 1, \boxed{\ldots}, 1000\}$

$pc' = \text{“}middle\text{”}$

else **if** $pc = \text{“}middle\text{”}$

then $i' = i + 1$

$pc' = \text{“}done\text{”}$

else no next values

“. . .” is informal math.

in here means is an element of the set of integers from 0 to 1000.

In is written in mathematics as this symbol.

Dot-dot-dot is informal math. We want to write this whole formula in a precisely defined language.

if $pc = \text{“}start\text{”}$

then $i' \in \{0, 1, \ldots, 1000\}$

$pc' = \text{“}middle\text{”}$

else **if** $pc = \text{“}middle\text{”}$

then $i' = i + 1$

$pc' = \text{“}done\text{”}$

else no next values

This set

in here means is an element of the set of integers from 0 to 1000.

In is written in mathematics as this symbol.

Dot-dot-dot is informal math. We want to write this whole formula in a precisely defined language.

The set of integers from 0 to 1000 is written in TLA+ like this.

if $pc = \text{"}start\text{"}$

then $i' \in 0\,..\,1000$

$pc' = \text{"}middle\text{"}$

else **if** $pc = \text{"}middle\text{"}$

then $i' = i + 1$

$pc' = \text{"}done\text{"}$

else no next values

This set is written in TLA⁺ as $0\,..\,1000$.

in here means is an element of the set of integers from 0 to 1000.

In is written in mathematics as this symbol.

Dot-dot-dot is informal math. We want to write this whole formula in a precisely defined language.

The set of integers from 0 to 1000 is written in TLA+ like this.

if pc = "$start$"
  then $i' \in$ 0 . . 1000
    pc' = "$middle$"
  else **if** pc = "$middle$"
    then i' = $i + 1$
      pc' = "$done$"
    else no next values

This set is written in TLA$^+$ as 0 . . 1000 .

The operator . . is precisely defined.

in here means is an element of the set of integers from 0 to 1000.

In is written in mathematics as this symbol.

Dot-dot-dot is informal math. We want to write this whole formula in a precisely defined language.

The set of integers from 0 to 1000 is written in TLA+ like this.

Where the operator dot-dot is precisely defined.

if $pc = \text{"}start\text{"}$
then $i' \in 0 \,..\, 1000$
$pc' = \text{"}middle\text{"}$
else **if** $pc = \text{"}middle\text{"}$
then $i' = i + 1$
$pc' = \text{"}done\text{"}$
else no next values

This **then** clause consists of two separate formulas.

if $pc = \text{“}start\text{”}$

then $\boxed{\begin{array}{l} i' \in 0\,..\,1000 \\ pc' = \text{“}middle\text{”} \end{array}}$

else **if** $pc = \text{“}middle\text{”}$

then $i' = i + 1$

$pc' = \text{“}done\text{”}$

else no next values

This **then** clause is two formulas.

This **then** clause consists of two separate formulas.

if pc = "$start$"

then $\boxed{\begin{array}{l} i' \in 0 \,..\, 1000 \\ pc' = \text{"}middle\text{"} \end{array}}$

else **if** pc = "$middle$"

then $i' = i + 1$

$pc' =$ "$done$"

else no next values

It should be a single formula

This **then** clause consists of two separate formulas.

It should be a single formula asserting that both formulas are true.

if pc = "$start$"

then $\boxed{\begin{array}{l} i' \in 0 \,..\, 1000 \\ pc' = \text{“}middle\text{”} \end{array}}$

else **if** pc = "$middle$"

then $i' = i + 1$

$pc' =$ "$done$"

else no next values

It should be a single formula asserting that both formulas are true.

This **then** clause consists of two separate formulas.

It should be a single formula asserting that both formulas are true.

if $pc = \text{"}start\text{"}$

  then $i' \in 0 \,..\, 1000$ and

    $pc' = \text{"}middle\text{"}$

  else **if** $pc = \text{"}middle\text{"}$

    then $i' = i + 1$

      $pc' = \text{"}done\text{"}$

    else no next values

There's an implicit "and" here

This **then** clause consists of two separate formulas.

It should be a single formula asserting that both formulas are true.

There's an implicit "and" here, and we know how to write **and** in math:

if pc = "$start$"

then $i' \in 0\,..\,1000 \;\wedge$

pc' = "$middle$"

else **if** pc = "$middle$"

then $i' = i + 1$

pc' = "$done$"

else no next values

There's an implicit "and" here that we can replace with $\wedge$.

This **then** clause consists of two separate formulas.

It should be a single formula asserting that both formulas are true.

There's an implicit "and" here, and we know how to write **and** in math: we replace it with this *conjunction* symbol.

$$
\begin{array}{l}
\textbf{if } pc = \text{"}start\text{"} \\
\quad \textbf{then } (i' \in 0 \,..\, 1000) \;\land \\
\quad \qquad\;\; (pc' = \text{"}middle\text{"}) \\
\quad \textbf{else } \textbf{ if } pc = \text{"}middle\text{"} \\
\qquad\qquad \textbf{then } i' = i + 1 \\
\qquad\qquad\qquad\;\; pc' = \text{"}done\text{"} \\
\qquad\qquad \textbf{else } \text{ no next values}
\end{array}
$$

Let's make it easier to read.

This **then** clause consists of two separate formulas.

It should be a single formula asserting that both formulas are true.

There's an implicit "and" here, and we know how to write **and** in math: we replace it with this *conjunction* symbol.

Let's add some parentheses to make it easier to read.

if pc = "$start$"
  then $(i' \in 0\,..1000) \;\wedge$
  $(pc' = \text{"}middle\text{"})$
  else **if** pc = "$middle$"
    then $i' = i + 1$
    $pc' = \text{"}done\text{"}$
    else no next values

We do the same thing with the second **then** clause.

if $pc = \text{"}start\text{"}$

  then $(i' \in 0\,..\,1000) \;\land$

    $(pc' = \text{"}middle\text{"})$

  else **if** $pc = \text{"}middle\text{"}$

    then $\boxed{\begin{array}{l} i' = i + 1 \\ pc' = \text{"}done\text{"} \end{array}}$

    else no next values

We do the same thing here.

We do the same thing with the second **then** clause.

$$
\begin{array}{l}
\textbf{if } pc = \text{“}start\text{”} \\
\quad \textbf{then } (i' \in 0\,..1000) \;\land \\
\quad \phantom{\textbf{then }} (pc' = \text{“}middle\text{”}) \\
\quad \textbf{else } \textbf{if } pc = \text{“}middle\text{”} \\
\quad \phantom{\textbf{else if }} \textbf{then } (i' = i + 1) \;\land \\
\quad \phantom{\textbf{else if } \textbf{then }} (pc' = \text{“}done\text{”}) \\
\quad \phantom{\textbf{else if }} \textbf{else } \text{no next values}
\end{array}
$$

We do the same thing here.

We do the same thing with the second **then** clause.

if pc = "$start$"
  then $(i' \in 0\,..1000) \;\wedge$
    $(pc' = \text{"}middle\text{"})$
  else **if** pc = "$middle$"
    then $(i' = i + 1) \;\wedge$
      $(pc' = \text{"}done\text{"})$
    else no next values

What about "no next values", which certainly isn't a mathematical formula.

There's something important you need to understand.

if pc = "$start$"
  then $(i' \in 0 \,..\, 1000) \;\wedge$
    $(pc' = \text{“}middle\text{”})$
  else **if** pc = "$middle$"
    then $(i' = i + 1) \;\wedge$
      $(pc' = \text{“}done\text{”})$
    else no next values

What about this?

What about “no next values”, which certainly isn’t a mathematical formula.

There’s something important you need to understand.

if pc = "$start$"

  then $(i' \in 0\,..1000) \;\wedge$

   $(pc' = \text{"}middle\text{"})$

  else **if** pc = "$middle$"

   then $(i' = i + 1) \;\wedge$

    $(pc' = \text{"}done\text{"})$

   else no next values

We're not writing instructions for computing something.

What about "no next values", which certainly isn't a mathematical formula.

There's something important you need to understand.

We're not writing instructions for computing something.

if pc = "$start$"
    then ($i' \in 0\,..\,1000$) $\land$
        (pc' = "$middle$")
    else **if** pc = "$middle$"
        then ($i' = i + 1$) $\land$
            (pc' = "$done$")
        else no next values

We're not writing instructions for computing something.

We're writing a formula relating i, pc, i', and pc'.

What about "no next values", which certainly isn't a mathematical formula.

There's something important you need to understand.

We're not writing instructions for computing something.

We are writing a formula relating the values of i, pc, i', and pc'

if pc = "$start$"

then $(i' \in 0\,..\,1000) \;\land$
$(pc' = \text{"}middle\text{"})$

else **if** pc = "$middle$"

then $(i' = i + 1) \;\land$
$(pc' = \text{"}done\text{"})$

else no next values

It does not mean: if $pc = \text{"}start\text{"}$

This formula does **not** mean that if pc equals $start$

if $pc = \text{“}start\text{”}$

then $(i' \in 0\,..1000) \;\wedge$

$(pc' = \text{“}middle\text{”})$

else **if** $pc = \text{“}middle\text{”}$

then $(i' = i + 1) \;\wedge$

$(pc' = \text{“}done\text{”})$

else no next values

It does not mean: if $pc = \text{“}start\text{”}$ do the **then** part

This formula does **not** mean that if pc equals $start$

then **do** the then part

if $pc = \text{“}start\text{”}$
  then $(i' \in 0 \,..\, 1000) \;\land$
    $(pc' = \text{“}middle\text{”})$
  else **if** $pc = \text{“}middle\text{”}$
    then $(i' = i + 1) \;\land$
      $(pc' = \text{“}done\text{”})$
    else no next values

It does not mean: if $pc = \text{“}start\text{”}$ do the **then** part, otherwise do the **else** part.

This formula does **not** mean that if pc equals $start$ then **do** the then part otherwise **do** the else part.

if pc = "$start$"
  then $(i' \in 0\,..1000) \;\wedge$
    $(pc' = \text{"}middle\text{"})$
  else **if** pc = "$middle$"
    then $(i' = i + 1) \;\wedge$
      $(pc' = \text{"}done\text{"})$
    else no next values

It means: if $pc = \text{"}start\text{"}$

This formula does **not** mean that if pc equals $start$
then **do** the then part otherwise **do** the else part.

The formula **does** mean that if pc equals $start$

$$
\begin{array}{l}
\textbf{if } pc = \text{“}start\text{”} \\
\quad \textbf{then } (i' \in 0\,..\,1000) \;\wedge \\
\qquad\qquad (pc' = \text{“}middle\text{”}) \\
\quad \textbf{else } \textbf{if } pc = \text{“}middle\text{”} \\
\qquad\qquad \textbf{then } (i' = i + 1) \;\wedge \\
\qquad\qquad\qquad (pc' = \text{“}done\text{”}) \\
\qquad\qquad \textbf{else } \text{no next values}
\end{array}
$$

It means: if $pc = $ “$start$” the formula equals the **then** formula

This formula does **not** mean that if pc equals $start$
then **do** the then part otherwise **do** the else part.
The formula **does** mean that if pc equals $start$
then the value of the formula **equals** the value of the then formula

if pc = "$start$"

  then $(i' \in 0\,..\,1000)$ $\wedge$

    $(pc' = \text{"}middle\text{"})$

  else **if** pc = "$middle$"

    then $(i' = i + 1)$ $\wedge$

      $(pc' = \text{"}done\text{"})$

    else no next values

It means: if $pc = \text{"}start\text{"}$ the formula equals the **then** formula, otherwise it equals the **else** formula.

This formula does **not** mean that if pc equals $start$ then **do** the then part otherwise **do** the else part.

The formula **does** mean that if pc equals $start$ then the value of the formula **equals** the value of the then formula otherwise its value equals the value of the else formula.

if pc = “$start$”

then $(i' \in 0 \,..\, 1000) \;\wedge$

$(pc' = \text{“}middle\text{”})$

else **if** pc = “$middle$”

then $(i' = i + 1) \;\wedge$

$(pc' = \text{“}done\text{”})$

else no next values

The value of the formula equals **true** for these values of i, pc, i', and pc' because:

if $pc = \text{"}start\text{"}$

then $(i' \in 0\,..\,1000) \;\wedge$

$(pc' = \text{"}middle\text{"})$

else **if** $pc = \text{"}middle\text{"}$

then $(i' = i + 1) \;\wedge$

$(pc' = \text{"}done\text{"})$

else no next values

The formula equals **true** for these values:

$i = 17 \qquad pc = \text{"}start\text{"} \qquad i' = 534 \qquad pc' = \text{"}middle\text{"}$

The value of the formula equals **true** for these values of i, pc, i', and pc' because:

if $\boxed{pc = \text{“}start\text{”}}$

then $(i' \in 0\,..\,1000) \;\land$

$(pc' = \text{“}middle\text{”})$

else **if** $pc = \text{“}middle\text{”}$

then $(i' = i + 1) \;\land$

$(pc' = \text{“}done\text{”})$

else no next values

The formula equals **true** for these values:

$i = 17 \qquad pc = \text{“}start\text{”} \qquad i' = 534 \qquad pc' = \text{“}middle\text{”}$

The value of the formula equals **true** for these values of i, pc, i', and pc' because:

The **if** test equals true

if $\boxed{pc = \text{"}start\text{"}}$
then $(i' \in 0\,..\,1000) \;\wedge$
$(pc' = \text{"}middle\text{"})$
else **if** $pc = \text{"}middle\text{"}$
then $(i' = i + 1) \;\wedge$
$(pc' = \text{"}done\text{"})$
else no next values

The formula equals **true** for these values:

$i = 17$ $\boxed{pc = \text{"}start\text{"}}$ $i' = 534$ $pc' = \text{"}middle\text{"}$

The value of the formula equals **true** for these values of i, pc, i', and pc' because:

The **if** test equals true because pc equals *start*

if pc = "$start$"
  then $\boxed{(i' \in 0\,..\,1000) \ \wedge \ (pc' = \text{"}middle\text{"})}$
  else **if** pc = "$middle$"
    then $(i' = i + 1) \ \wedge$
      $(pc' = \text{"}done\text{"})$
    else no next values

The formula equals **true** for these values:

$i = 17 \quad pc = \text{"}start\text{"} \quad i' = 534 \quad pc' = \text{"}middle\text{"}$

The value of the formula equals **true** for these values of i, pc, i', and pc' because:

The **if** test equals true because pc equals *start*

So the value of the formula equals the value of the **then** clause

$$
\begin{array}{l}
\textbf{if } pc = \text{“}start\text{”} \\
\quad \textbf{then } \boxed{(i' \in 0\,..\,1000)} \;\wedge \\
\quad \phantom{\textbf{then }} \boxed{(pc' = \text{“}middle\text{”})} \\
\quad \textbf{else } \textbf{if } pc = \text{“}middle\text{”} \\
\quad \quad \textbf{then } (i' = i + 1) \;\wedge \\
\quad \quad \phantom{\textbf{then }} (pc' = \text{“}done\text{”}) \\
\quad \quad \textbf{else } \text{no next values}
\end{array}
$$

The formula equals **true** for these values:

$i = 17 \quad pc = \text{“}start\text{”} \quad i' = 534 \quad pc' = \text{“}middle\text{”}$

The value of the formula equals **true** for these values of i, pc, i', and pc' because:

The **if** test equals true because pc equals *start*

So the value of the formula equals the value of the **then** clause

This clause equals true if and only these two formulas equals true.

if pc = "$start$"

then $\boxed{(i' \in 0\,..1000)} \;\wedge$

$(pc' \;=\; \text{"}middle\text{"})$

else **if** pc = "$middle$"

then $(i' \;=\; i + 1) \;\;\wedge$

$(pc' \;=\; \text{"}done\text{"})$

else no next values

The formula equals **true** for these values:

$i = 17 \qquad pc = \text{"}start\text{"} \qquad i' = 534 \qquad pc' = \text{"}middle\text{"}$

The first formula equals true because

$$
\begin{array}{l}
\textbf{if } pc = \text{“}start\text{”} \\
\quad \textbf{then } \boxed{(i' \in 0\,..\,1000)} \;\land \\
\quad\quad\quad (pc' = \text{“}middle\text{”}) \\
\quad \textbf{else } \textbf{if } pc = \text{“}middle\text{”} \\
\quad\quad\quad \textbf{then } (i' = i + 1) \;\land \\
\quad\quad\quad\quad\quad (pc' = \text{“}done\text{”}) \\
\quad\quad\quad \textbf{else } \text{no next values}
\end{array}
$$

The formula equals **true** for these values:

$$i = 17 \qquad pc = \text{“}start\text{”} \qquad \boxed{i' = 534} \qquad pc' = \text{“}middle\text{”}$$

The first formula equals true because
i' equals 534, which is an element of the set of integers from 0 to 1000.

$$
\begin{array}{l}
\textbf{if } pc = \text{"}start\text{"} \\
\quad \textbf{then } (i' \in 0\,..\,1000) \;\wedge \\
\qquad\quad \boxed{(pc' = \text{"}middle\text{"})} \\
\quad \textbf{else if } pc = \text{"}middle\text{"} \\
\qquad\qquad \textbf{then } (i' = i + 1) \;\wedge \\
\qquad\qquad\qquad\; (pc' = \text{"}done\text{"}) \\
\qquad\qquad \textbf{else } \text{ no next values}
\end{array}
$$

The formula equals **true** for these values:

$i = 17 \quad pc = \text{"}start\text{"} \quad i' = 534 \quad \boxed{pc' = \text{"}middle\text{"}}$

The first formula equals true because i' equals 534, which is an element of the set of integers from 0 to 1000.

The second formula equals true because pc' equals middle.

$$
\begin{array}{l}
\textbf{if } pc = \text{“}start\text{”} \\
\quad \textbf{then } (i' \in 0\,..\,1000) \;\wedge \\
\qquad\quad (pc' = \text{“}middle\text{”}) \\
\quad \textbf{else if } pc = \text{“}middle\text{”} \\
\qquad\quad \textbf{then } (i' = i + 1) \;\wedge \\
\qquad\qquad\quad (pc' = \text{“}done\text{”}) \\
\qquad\quad \textbf{else } \text{no next values}
\end{array}
$$

The formula equals **true** for these values:

$$i = 17 \qquad pc = \text{“}start\text{”} \qquad i' = 534 \qquad pc' = \text{“}middle\text{”}$$

The first formula equals true because
i' equals 534, which is an element of the set of integers from 0 to 1000.

The second formula equals true because pc' equals middle.

So the whole formula equals true for these values.

if pc = "$start$"

  then $(i' \in 0 \, . \, . \, 1000) \; \wedge$

  $(pc' = \text{"}middle\text{"})$

  else **if** pc = "$middle$"

   then $(i' = i + 1) \; \wedge$

   $(pc' = \text{"}done\text{"})$

   else no next values

The formula equals false for these values because

if pc = "$start$"
    then $(i' \in 0\,..1000)$ $\wedge$
        $(pc'$ = "$middle$")
    else **if** pc = "$middle$"
        then $(i' = i + 1)$ $\wedge$
            $(pc'$ = "$done$")
        else no next values

The formula equals **false** for these values:

$i = 534$ $\quad pc =$ "$middle$" $\quad i' = 77$ $\quad pc' =$ "$done$"

The formula equals false for these values because

if $\boxed{pc = \text{"}start\text{"}}$
  then $(i' \in 0 \,..\, 1000) \;\wedge$
    $(pc' = \text{"}middle\text{"})$
  else **if** $pc = \text{"}middle\text{"}$
    then $(i' = i + 1) \;\wedge$
      $(pc' = \text{"}done\text{"})$
    else no next values

The formula equals **false** for these values:

$i = 534 \quad \boxed{pc = \text{"}middle\text{"}} \quad i' = 77 \quad pc' = \text{"}done\text{"}$

The formula equals false for these values because

The **if** test equals false

if pc = "$start$"
  then $(i' \in 0\,..\,1000)\ \wedge$
    $(pc' = \text{"}middle\text{"})$
  else **if** pc = "$middle$"
    then $(i' = i + 1)\ \wedge$
      $(pc' = \text{"}done\text{"})$
    else no next values

The formula equals **false** for these values:

$i = 534 \quad pc = \text{"}middle\text{"} \quad i' = 77 \quad pc' = \text{"}done\text{"}$

The formula equals false for these values because

The **if** test equals false
so the value of the formula equals the value of the **else** clause.

That clause is an **if** formula

if pc = "$start$"
then $(i' \in 0\,..\,1000) \;\wedge$
$(pc' = \text{"}middle\text{"})$
else **if** $\boxed{pc = \text{"}middle\text{"}}$
then $(i' = i + 1) \;\wedge$
$(pc' = \text{"}done\text{"})$
else no next values

The formula equals **false** for these values:

$i = 534$ $\boxed{pc = \text{"}middle\text{"}}$ $i' = 77$ $pc' = \text{"}done\text{"}$

The formula equals false for these values because

The **if** test equals false
so the value of the formula equals the value of the **else** clause.

That clause is an **if** formula whose whose test equals true,

if pc = "$start$"

then $(i' \in 0\,..\,1000) \ \wedge$

$(pc' = \text{"}middle\text{"})$

else **if** pc = "$middle$"

then $\boxed{\begin{array}{l}(i' = i + 1) \ \wedge \\ (pc' = \text{"}done\text{"})\end{array}}$

else no next values

The formula equals **false** for these values:

$i = 534 \qquad pc = \text{"}middle\text{"} \qquad i' = 77 \qquad pc' = \text{"}done\text{"}$

The formula equals false for these values because

The **if** test equals false
so the value of the formula equals the value of the **else** clause.

That clause is an **if** formula whose whose test equals true,
so it equals its **then** clause.

The value of that clause equals true if and only if

if pc = "$start$"

  then $(i' \in 0\,..\,1000)$ $\wedge$

    $(pc'$ = "$middle$")

  else **if** pc = "$middle$"

    then $\boxed{(i' = i + 1)} \wedge$

      $\boxed{(pc' = \text{“}done\text{”})}$

    else no next values

The formula equals **false** for these values:

$i = 534$ $pc = \text{“}middle\text{”}$ $i' = 77$ $pc' = \text{“}done\text{”}$

these two formulas both equal true.

if pc = "$start$"
  then $(i' \in 0\,..\,1000) \;\wedge$
    $(pc' = \text{“}middle\text{”})$
  else **if** pc = "$middle$"
    then $\boxed{(i' = i + 1)} \;\wedge$
      $(pc' = \text{“}done\text{”})$
    else no next values

The formula equals **false** for these values:

$i = 534 \qquad pc = \text{“}middle\text{”} \qquad i' = 77 \qquad pc' = \text{“}done\text{”}$

these two formulas both equal true.

But this formula equals false

if pc = "$start$"
    then $(i' \in 0\,..1000)$ $\wedge$
        $(pc'$ = "$middle$")
    else **if** pc = "$middle$"
        then $\boxed{(i' = i + 1)}$ $\wedge$
            $(pc'$ = "$done$")
        else no next values

The formula equals **false** for these values:

$i = 534$ $\quad pc =$ "$middle$" $\quad \boxed{i' = 77}$ $\quad pc' =$ "$done$"

these two formulas both equal true.

But this formula equals false because i'

$$
\begin{array}{l}
\textbf{if } pc = \text{“}start\text{”} \\
\quad \textbf{then } (i' \in 0\,..\,1000) \;\wedge \\
\quad\qquad\;\; (pc' = \text{“}middle\text{”}) \\
\quad \textbf{else } \textbf{if } pc = \text{“}middle\text{”} \\
\quad\qquad\quad \textbf{then } \boxed{(i' = i + 1)} \;\wedge \\
\quad\qquad\qquad\quad\;\; (pc' = \text{“}done\text{”}) \\
\quad\qquad\quad \textbf{else } \text{no next values}
\end{array}
$$

The formula equals **false** for these values:

$\boxed{i = 534}$ $\quad pc = \text{“}middle\text{”}$ $\quad \boxed{i' = 77}$ $\quad pc' = \text{“}done\text{”}$

these two formulas both equal true.

But this formula equals false because i' does not equal i plus 1.

if pc = "$start$"

  then $(i' \in 0 \,..\, 1000) \;\wedge$

    $(pc' = \text{"}middle\text{"})$

  else **if** pc = "$middle$"

    then $(i' = i + 1) \;\wedge$

      $(pc' = \text{"}done\text{"})$

    else no next values

The formula equals **false** for these values:

$i = 534$ $pc = \text{"}middle\text{"}$ $i' = 77$ $pc' = \text{"}done\text{"}$

these two formulas both equal true.

But this formula equals false because i' does not equal i plus 1.

So the entire formula equals false.

$$
\begin{array}{l}
\textbf{if } pc = \text{"}start\text{"} \\
\quad \textbf{then } (i' \in 0\,..\,1000) \;\wedge \\
\quad\quad\quad\;\; (pc' = \text{"}middle\text{"}) \\
\quad \textbf{else } \textbf{if } pc = \text{"}middle\text{"} \\
\quad\quad\quad\quad \textbf{then } (i' = i + 1) \;\wedge \\
\quad\quad\quad\quad\quad\quad\;\; (pc' = \text{"}done\text{"}) \\
\quad\quad\quad\quad \textbf{else } \text{no next values}
\end{array}
$$

Now let's return to the *no next values* clause.

$$
\begin{array}{l}
\textbf{if } pc = \text{“}start\text{”} \\
\quad \textbf{then } (i' \in 0 \,..\, 1000) \;\wedge \\
\qquad\qquad (pc' = \text{“}middle\text{”}) \\
\quad \textbf{else } \textbf{if } pc = \text{“}middle\text{”} \\
\qquad\qquad \textbf{then } (i' = i + 1) \;\wedge \\
\qquad\qquad\qquad (pc' = \text{“}done\text{”}) \\
\qquad\qquad \textbf{else } \text{no next values}
\end{array}
$$

Let's return to this clause.

Now let's return to the *no next values* clause.

if pc = "$start$"
  then $(i' \in 0 \,..\, 1000) \;\wedge$
    $(pc' \;=\; \text{“}middle\text{”})$
  else **if** pc = "$middle$"
    then $(i' \;=\; i+1) \;\wedge$
      $(pc' \;=\; \text{“}done\text{”})$
    else no next values

It should be a formula that does not equal true
for any values of i, pc, i', and pc'.

Now let's return to the *no next values* clause.

This clause should be a formula that does not equal true for *any* values of i, pc, i', and pc'.

if pc = "$start$"

then $(i' \in 0\,..\,1000)\ \wedge$

$(pc' = \text{"}middle\text{"})$

else **if** pc = "$middle$"

then $(i' = i + 1)\ \wedge$

$(pc' = \text{"}done\text{"})$

else no next values

The simplest

~~It should be a~~ formula that does not equal true for any values of i, pc, i', and pc'.

Now let's return to the *no next values* clause.

This clause should be a formula that does not equal true for *any* values of i, pc, i', and pc'.

Let's use the simplest such formula, which is one that always equals false—namely,

$$
\begin{array}{l}
\textbf{if } pc = \text{“}start\text{”} \\
\quad \textbf{then } (i' \in 0\,..\,1000) \;\land \\
\quad \phantom{\textbf{then }} (pc' = \text{“}middle\text{”}) \\
\quad \textbf{else if } pc = \text{“}middle\text{”} \\
\quad \quad \quad \textbf{then } (i' = i + 1) \;\land \\
\quad \quad \quad \phantom{\textbf{then }} (pc' = \text{“}done\text{”}) \\
\quad \quad \quad \textbf{else } \text{FALSE}
\end{array}
$$

The simplest
~~It should be a~~ formula that does not equal true for any values of i, pc, i', and pc'.

Now let's return to the *no next values* clause.

This clause should be a formula that does not equal true for *any* values of i, pc, i', and pc'.

Let's use the simplest such formula, which is one that always equals false—namely, the formula *false*.

[slide 111]

$$
\begin{array}{l}
\textbf{if } pc = \text{“}start\text{”} \\
\quad \textbf{then } (i' \in 0\,..\,1000) \;\land \\
\quad \phantom{\textbf{then }} (pc' = \text{“}middle\text{”}) \\
\quad \textbf{else } \textbf{if } pc = \text{“}middle\text{”} \\
\quad\quad\quad \textbf{then } (i' = i + 1) \;\land \\
\quad\quad\quad \phantom{\textbf{then }} (pc' = \text{“}done\text{”}) \\
\quad\quad\quad \textbf{else } \text{FALSE}
\end{array}
$$

In TLA+ most keywords

$$
\begin{array}{l}
\textbf{if } pc = \text{"}start\text{"} \\
\quad \textbf{then } (i' \in 0\,..\,1000) \land \\
\qquad\qquad (pc' = \text{"}middle\text{"}) \\
\quad \textbf{else } \textbf{if } pc = \text{"}middle\text{"} \\
\qquad\qquad \textbf{then } (i' = i + 1) \land \\
\qquad\qquad\qquad (pc' = \text{"}done\text{"}) \\
\qquad\qquad \textbf{else } \text{FALSE}
\end{array}
$$

In TLA$^+$ most keywords

In TLA+ most keywords

$$
\begin{array}{l}
\textsf{IF}\ \ pc = \text{“}start\text{”} \\
\quad \textsf{THEN}\ (i' \in 0\,..\,1000) \ \wedge \\
\quad \phantom{\textsf{THEN}}\ (pc' = \text{“}middle\text{”}) \\
\quad \textsf{ELSE}\ \ \textsf{IF}\ \ pc = \text{“}middle\text{”} \\
\quad \phantom{\textsf{ELSE}\ \ }\quad \textsf{THEN}\ (i' = i + 1) \ \wedge \\
\quad \phantom{\textsf{ELSE}\ \ }\quad \phantom{\textsf{THEN}}\ (pc' = \text{“}done\text{”}) \\
\quad \phantom{\textsf{ELSE}\ \ }\quad \textsf{ELSE}\ \ \textsf{FALSE}
\end{array}
$$

In TLA+ most keywords are in uppercase.

In TLA+ most keywords are in uppercase letters.

$$
\begin{array}{l}
\text{IF } pc = \text{“}start\text{”} \\
\quad \text{THEN } (i' \in 0 \,..\, 1000) \ \wedge \\
\qquad\qquad (pc' = \text{“}middle\text{”}) \\
\quad \text{ELSE IF } pc = \text{“}middle\text{”} \\
\qquad\qquad \text{THEN } (i' = i + 1) \ \wedge \\
\qquad\qquad\qquad (pc' = \text{“}done\text{”}) \\
\qquad\qquad \text{ELSE FALSE}
\end{array}
$$

This is a TLA^{+} formula.

In TLA+ most keywords are in uppercase letters.

This is now a TLA+ formula.

$$
\begin{array}{l}
\text{IF } pc = \text{“}start\text{”} \\
\quad \text{THEN } (i' \in 0\,..\,1000) \;\land \\
\quad\qquad\quad (pc' = \text{“}middle\text{”}) \\
\quad \text{ELSE IF } pc = \text{“}middle\text{”} \\
\qquad\qquad \text{THEN } (i' = i + 1) \;\land \\
\qquad\qquad\qquad\quad (pc' = \text{“}done\text{”}) \\
\qquad\qquad \text{ELSE FALSE}
\end{array}
$$

pretty-printed

This is a TLA+ formula.

In TLA+ most keywords are in uppercase letters.

This is now a TLA+ formula. That is, a pretty-printed TLA+ formula.

$$
\begin{array}{l}
\text{IF } pc = \text{“}start\text{”} \\
\quad \text{THEN } (i' \in 0 \,..\, 1000) \land \\
\qquad\qquad (pc' = \text{“}middle\text{”}) \\
\quad \text{ELSE IF } pc = \text{“}middle\text{”} \\
\qquad\qquad \text{THEN } (i' = i + 1) \land \\
\qquad\qquad\qquad (pc' = \text{“}done\text{”}) \\
\qquad\qquad \text{ELSE FALSE}
\end{array}
$$

The TLA$^+$ source is in ASCII

In TLA+ most keywords are in uppercase letters.

This is now a TLA+ formula. That is, a pretty-printed TLA+ formula.

The TLA+ source is in ASCII,

$$
\begin{array}{l}
\text{IF } pc = \text{“}start\text{”} \\
\quad \text{THEN } (i' \in 0\,..\,1000) \;\land \\
\quad\quad\quad\quad (pc' = \text{“}middle\text{”}) \\
\quad \text{ELSE IF } pc = \text{“}middle\text{”} \\
\quad\quad\quad \text{THEN } (i' = i + 1) \;\land \\
\quad\quad\quad\quad\quad (pc' = \text{“}done\text{”}) \\
\quad\quad\quad \text{ELSE FALSE}
\end{array}
$$

The TLA$^+$ source is in ASCII, with $\land$ typed as `/\`

In TLA+ most keywords are in uppercase letters.

This is now a TLA+ formula. That is, a pretty-printed TLA+ formula.

The TLA+ source is in ASCII,
with *and* typed as forward-slash backslash

$$
\begin{array}{l}
\text{IF } pc = \text{"}start\text{"} \\
\quad \text{THEN } (i' \in 0\,..\,1000) \land \\
\qquad\quad (pc' = \text{"}middle\text{"}) \\
\quad \text{ELSE } \text{IF } pc = \text{"}middle\text{"} \\
\qquad\quad \text{THEN } (i' = i + 1) \land \\
\qquad\qquad (pc' = \text{"}done\text{"}) \\
\qquad\quad \text{ELSE } \text{FALSE}
\end{array}
$$

The TLA+ source is in ASCII, with $\land$ typed as `/\` and $\in$ typed as `\in`.

In TLA+ most keywords are in uppercase letters.

This is now a TLA+ formula. That is, a pretty-printed TLA+ formula.

The TLA+ source is in ASCII,
with *and* typed as forward-slash backslash
and the *element-of* symbol typed like this.

```
IF pc = "start"
  THEN (i' \in 0..1000) /\
       (pc' = "middle")
  ELSE IF pc = "middle"
         THEN (i' = i+1) /\
              (pc' = "done")
         ELSE FALSE
```

This is what it looks like in ASCII.

$$
\begin{array}{l}
\textsf{IF } pc = \text{“}start\text{”} \\
\quad \textsf{THEN } (i' \in 0\,..\,1000) \;\wedge \\
\quad \qquad\quad (pc' = \text{“}middle\text{”}) \\
\quad \textsf{ELSE IF } pc = \text{“}middle\text{”} \\
\quad \qquad\qquad \textsf{THEN } (i' = i + 1) \;\wedge \\
\quad \qquad\qquad \qquad\quad (pc' = \text{“}done\text{”}) \\
\quad \qquad\qquad \textsf{ELSE FALSE}
\end{array}
$$

This version is easier for most people to read.

This version is easier for most people to read.

$$
\begin{array}{l}
\textsf{IF } pc = \text{"}start\text{"} \\
\quad \textsf{THEN } (i' \in 0\,..\,1000) \ \land \\
\qquad\qquad (pc' = \text{"}middle\text{"}) \\
\quad \textsf{ELSE IF } pc = \text{"}middle\text{"} \\
\qquad\qquad \textsf{THEN } (i' = i + 1) \ \land \\
\qquad\qquad\qquad (pc' = \text{"}done\text{"}) \\
\qquad\qquad \textsf{ELSE FALSE}
\end{array}
$$

This version is easier for most people to read.

I'll use it for now.

This version is easier for most people to read.

I'll use it for now.

The Complete Mathematical Description

We have now written a complete mathematical description of the program as two formulas.

The Complete Mathematical Description

Initial-state formula: $(i = 0) \wedge (pc = \text{“}start\text{”})$

We have now written a complete mathematical description of the program as two formulas.

The initial-state formula.

The Complete Mathematical Description

Initial-state formula: $(i = 0) \land (pc = \text{“}start\text{”})$

Next-state formula:

$$
\begin{array}{l}
\text{IF } pc = \text{“}start\text{”} \\
\quad \text{THEN } (i' \in 0\,..\,1000) \land \\
\quad\qquad\quad (pc' = \text{“}middle\text{”}) \\
\quad \text{ELSE IF } pc = \text{“}middle\text{”} \\
\quad\qquad\quad \text{THEN } (i' = i + 1) \land \\
\quad\qquad\qquad\qquad (pc' = \text{“}done\text{”}) \\
\quad\qquad\quad \text{ELSE FALSE}
\end{array}
$$

We have now written a complete mathematical description of the program as two formulas.

The initial-state formula. and the next-state formula.

The Complete Mathematical Description

Initial-state formula: $(i = 0) \wedge (pc = \text{“}start\text{”})$

Next-state formula:

$$
\begin{array}{l}
\text{IF } pc = \text{“}start\text{”} \\
\quad \text{THEN } (i' \in 0 \,..\, 1000) \wedge \\
\quad \phantom{\text{THEN }} (pc' = \text{“}middle\text{”}) \\
\quad \text{ELSE IF } pc = \text{“}middle\text{”} \\
\quad \quad \quad \text{THEN } (i' = i + 1) \wedge \\
\quad \quad \quad \phantom{\text{THEN }} (pc' = \text{“}done\text{”}) \\
\quad \quad \quad \text{ELSE FALSE}
\end{array}
$$

There's a nicer way to write this.

We have now written a complete mathematical description of the program as two formulas.

The initial-state formula. and the next-state formula.

But there's a nicer way to write the next-state formula.

A NICER WAY TO WRITE THE NEXT-STATE FORMULA

Let’s now see how.

$$
\begin{array}{l}
\text{IF } pc = \text{"}start\text{"} \\
\quad \text{THEN } (i' \in 0 \,..\, 1000) \;\land \\
\quad \qquad\quad (pc' = \text{"}middle\text{"}) \\
\quad \text{ELSE IF } pc = \text{"}middle\text{"} \\
\quad \qquad\quad \text{THEN } (i' = i + 1) \;\land \\
\quad \qquad\qquad\qquad (pc' = \text{"}done\text{"}) \\
\quad \qquad\quad \text{ELSE FALSE}
\end{array}
$$

I'll start by hiding some of the details.

$$
\begin{array}{l}
\text{IF } pc = \text{"}start\text{"} \\
\quad \text{THEN } \boxed{\begin{array}{l}(i' \in 0 \,..\, 1000) \;\wedge \\ (pc' = \text{"}middle\text{"})\end{array}} \\
\quad \text{ELSE IF } pc = \text{"}middle\text{"} \\
\qquad\qquad \text{THEN } \boxed{\begin{array}{l}(i' = i + 1) \;\wedge \\ (pc' = \text{"}done\text{"})\end{array}} \\
\qquad\qquad \text{ELSE FALSE}
\end{array}
$$

Let's call these two formulas

I'll start by hiding some of the details.

Let's call these two formulas

IF $pc = \text{"}start\text{"}$

THEN $\boxed{A}$

ELSE IF $pc = \text{"}middle\text{"}$

THEN $\boxed{\begin{array}{l}(i' = i + 1) \wedge \\ (pc' = \text{"}done\text{"})\end{array}}$

ELSE FALSE

Let's call these two formulas A

I'll start by hiding some of the details.

Let's call these two formulas A

IF pc = "$start$"
THEN $\boxed{A}$
ELSE IF pc = "$middle$"
THEN $\boxed{B}$
ELSE FALSE

Let's call these two formulas A and B.

I'll start by hiding some of the details.

Let's call these two formulas A and B.

[slide 131]

IF $pc = \text{“}start\text{”}$

THEN A

ELSE IF $pc = \text{“}middle\text{”}$

THEN B

ELSE FALSE

IF $pc = \text{“}start\text{”}$

  THEN A

  ELSE IF $pc = \text{“}middle\text{”}$

      THEN B

      ELSE FALSE

$$
\begin{array}{l}
\text{IF } pc = \text{"}start\text{"} \\
\quad \text{THEN } A \\
\quad \text{ELSE IF } pc = \text{"}middle\text{"} \\
\qquad\qquad \text{THEN } B \\
\qquad\qquad \text{ELSE FALSE}
\end{array}
$$

There are two cases when the formula is true:

There are two cases when the formula is true:

IF $\boxed{pc = \text{“}start\text{”}}$
THEN $\boxed{A}$
ELSE IF $pc = \text{“}middle\text{”}$
THEN B
ELSE FALSE

There are two cases when the formula is true:

1. $pc = \text{“}start\text{”}$ and A are true.

There are two cases when the formula is true:

Case 1: pc equals *start* and A are both true.

IF $\boxed{pc = \text{“}start\text{”}}$

THEN $\boxed{A}$

ELSE IF $pc = \text{“}middle\text{”}$

THEN B

ELSE FALSE

There are two cases when the formula is true:

1. $(pc = \text{“}start\text{”}) \land A$ is true.

There are two cases when the formula is true:

Case 1: pc equals *start* and A are both true.

In other words, the single formula pc equals *start* and A is true.

$$
\begin{array}{l}
\text{IF } pc = \text{“}start\text{”} \\
\quad \text{THEN } A \\
\quad \text{ELSE IF } \boxed{pc = \text{“}middle\text{”}} \\
\qquad\qquad \text{THEN } \boxed{B} \\
\qquad\qquad \text{ELSE FALSE}
\end{array}
$$

There are two cases when the formula is true:

1. $(pc = \text{“}start\text{”}) \land A$ is true.
2. $pc = \text{“}middle\text{”}$ and B are true.

There are two cases when the formula is true:

Case 1: pc equals *start* and A are both true.

In other words, the single formula pc equals *start* and A is true.

Case 2: pc equals *middle* and B are both true.

IF pc = "$start$"

THEN A

ELSE IF $\boxed{pc = \text{"}middle\text{"}}$

THEN $\boxed{B}$

ELSE FALSE

There are two cases when the formula is true:

1. $(pc = \text{"}start\text{"}) \land A$ is true.
2. $(pc = \text{"}middle\text{"}) \land B$ is true.

There are two cases when the formula is true:

Case 1: pc equals *start* and A are both true.

In other words, the single formula pc equals *start* and A is true.

Case 2: pc equals *middle* and B are both true.

In other words, the single formula pc equals *middle* and B is true.

[slide 138]

IF pc = "$start$"
THEN A
ELSE IF pc = "$middle$"
THEN B
ELSE FALSE

$(pc = \text{“}start\text{”}) \;\wedge\; A$

or $(pc = \text{“}middle\text{”}) \;\wedge\; B$

There are two cases when the formula is true:

1. $(pc = \text{“}start\text{”}) \;\wedge\; A$ is true.
2. $(pc = \text{“}middle\text{”}) \;\wedge\; B$ is true.

So we can rewrite the formula like this.
To turn it into a mathematical formula,

IF pc = "$start$"
THEN A
ELSE IF pc = "$middle$"
THEN B
ELSE FALSE

$(pc = \text{“}start\text{”}) \land A$

or $(pc = \text{“}middle\text{”}) \land B$

Must replace “or” by a mathematical operator.

So we can rewrite the formula like this.
To turn it into a mathematical formula,
we must replace the word *or* by a mathematical operator.

IF $pc = \text{“}start\text{”}$
THEN A
ELSE IF $pc = \text{“}middle\text{”}$
THEN B
ELSE FALSE

$(pc = \text{“}start\text{”}) \land A$

or $(pc = \text{“}middle\text{”}) \land B$

Must replace “or” by a mathematical operator.

Written $||$ in some programming languages.

So we can rewrite the formula like this.
To turn it into a mathematical formula,
we must replace the word *or* by a mathematical operator.

That operator is written bar bar in some programming languages.

IF $pc = \text{"}start\text{"}$
THEN A
ELSE IF $pc = \text{"}middle\text{"}$
THEN B
ELSE FALSE

$$\begin{array}{ll} & ((pc = \text{"}start\text{"}) \land A) \\ \lor & ((pc = \text{"}middle\text{"}) \land B) \end{array}$$

Must replace “or” by a mathematical operator.

Written || in some programming languages.

Written $\lor$ in mathematics.

So we can rewrite the formula like this.
To turn it into a mathematical formula,
we must replace the word *or* by a mathematical operator.

That operator is written bar bar in some programming languages.

It’s written as this symbol in mathematics.

$$
\begin{array}{l}
\quad ((pc = \text{“}start\text{”}) \ \wedge \ A) \\
\vee \ ((pc = \text{“}middle\text{”}) \ \wedge \ B)
\end{array}
$$

$$
\begin{aligned}
&((pc = \text{“}start\text{”}) \;\wedge\; A) \\
\vee\; &((pc = \text{“}middle\text{”}) \;\wedge\; B)
\end{aligned}
$$

Let’s replace A and B by their original formulas.

Now let’s replace A and B by their original formulas.

$$
\begin{array}{ll}
 & ((pc = \text{“}start\text{”}) \;\wedge \\
 & \quad A) \\
\vee & ((pc = \text{“}middle\text{”}) \;\wedge \\
 & \quad B)
\end{array}
$$

Now let's replace A and B by their original formulas.

First let's give us some more room.

$$
\begin{array}{l}
\quad ((pc = \text{“}start\text{”}) \;\land \\
\quad\;\; (i' \in 0\,..1000) \;\land \\
\quad\;\; (pc' = \text{“}middle\text{”})) \\
\lor \;((pc = \text{“}middle\text{”}) \;\land \\
\quad\;\;\; B)
\end{array}
$$

Now let's replace A and B by their original formulas.

First let's give us some more room.

We replace A.

$$
\begin{array}{l}
\quad ((pc = \text{“}start\text{”}) \;\wedge \\
\quad\;\; (i' \in 0\,..\,1000) \;\wedge \\
\quad\;\; (pc' = \text{“}middle\text{”})) \\
\vee\; ((pc = \text{“}middle\text{”}) \;\wedge \\
\quad\;\; (i' = i + 1) \;\wedge \\
\quad\;\; (pc' = \text{“}done\text{”}))
\end{array}
$$

Now let's replace A and B by their original formulas.

First let's give us some more room.

We replace A.

And we replace B.

$$
\begin{array}{l}
\quad ((pc = \text{“}start\text{”}) \ \land \\
\quad\ \ (i' \in 0\,..1000) \ \land \\
\quad\ \ (pc' = \text{“}middle\text{”})) \\
\lor \ ((pc = \text{“}middle\text{”}) \ \land \\
\quad\ \ (i' = i + 1) \ \land \\
\quad\ \ (pc' = \text{“}done\text{”}))
\end{array}
$$

Let’s format it better.

Now let’s replace A and B by their original formulas.

First let’s give us some more room.

We replace A.

And we replace B.

And now let’s format it a little better.

$$
\begin{array}{ll}
& (\quad (pc = \text{“}start\text{”}) \\
& \quad \wedge (i' \in 0\,..\,1000) \\
& \quad \wedge (pc' = \text{“}middle\text{”})\,) \\
\vee & (\quad (pc = \text{“}middle\text{”}) \\
& \quad \wedge (i' = i + 1) \\
& \quad \wedge (pc' = \text{“}done\text{”})\,)
\end{array}
$$

$$
\begin{array}{rl}
& (\quad (pc = \text{“}start\text{”}) \\
& \quad \land\ (i' \in 0\,..\,1000) \\
& \quad \land\ (pc' = \text{“}middle\text{”})\,) \\
\lor & (\quad (pc = \text{“}middle\text{”}) \\
& \quad \land\ (i' = i + 1) \\
& \quad \land\ (pc' = \text{“}done\text{”})\,)
\end{array}
$$

These parentheses aren’t needed and don’t help

These parenthese aren’t necessary and with this formatting they don’t help.

$$
\begin{array}{ll}
 & (\quad pc = \text{“}start\text{”} \\
 & \quad \land\ i' \in 0\,..\,1000 \\
 & \quad \land\ pc' = \text{“}middle\text{”}\,) \\
\lor & (\quad pc = \text{“}middle\text{”} \\
 & \quad \land\ i' = i + 1 \\
 & \quad \land\ pc' = \text{“}done\text{”}\,)
\end{array}
$$

So let's remove them.

$$
\begin{array}{ll}
 & \boxed{(}\quad pc = \text{“}start\text{”} \\
 & \quad\;\; \land\ i' \in 0\,..\,1000 \\
 & \quad\;\; \land\ pc' = \text{“}middle\text{”}\boxed{)} \\
\lor & (\quad pc = \text{“}middle\text{”} \\
 & \quad\;\; \land\ i' = i + 1 \\
 & \quad\;\; \land\ pc' = \text{“}done\text{”}\,)
\end{array}
$$

Widely separated matching parentheses make formulas hard to read.

(They’re not very far apart here, but they could be in a larger formula.)

$$
\begin{array}{ll}
 & \boxed{\land} \;\; pc = \text{“}start\text{”} \\
 & \land \;\; i' \in 0 \,..\, 1000 \\
 & \land \;\; pc' = \text{“}middle\text{”} \\
\lor \;\; (& \;\; pc = \text{“}middle\text{”} \\
 & \land \;\; i' = i + 1 \\
 & \land \;\; pc' = \text{“}done\text{”}\,)
\end{array}
$$

Widely separated matching parentheses make formulas hard to read.

(They're not very far apart here, but they could be in a larger formula.)

TLA+ lets us eliminate them by adding this extra *and* symbol.

$$\boxed{\begin{array}{l}\land \\ \land \\ \land\end{array}}\begin{array}{l} pc = \text{“}start\text{”} \\ i' \in 0\,..\,1000 \\ pc' = \text{“}middle\text{”}\end{array}$$

Widely separated matching parentheses make formulas hard to read.

(They're not very far apart here, but they could be in a larger formula.)

TLA+ lets us eliminate them by adding this extra *and* symbol.

This turns the subformula into a bulleted *and* list that is ended by

$$\begin{array}{l} \land\ \ pc = \text{"}start\text{"} \\ \land\ \ i' \in 0\,..\,1000 \\ \land\ \ pc' = \text{"}middle\text{"} \end{array}$$

$\lor$

Widely separated matching parentheses make formulas hard to read.

(They're not very far apart here, but they could be in a larger formula.)

TLA+ lets us eliminate them by adding this extra *and* symbol.

This turns the subformula into a bulleted *and* list that is ended by any following token to the left of the *and* symbols.

[slide 155]

$$
\begin{array}{l}
\quad \boxed{(} \wedge \; pc = \text{“}start\text{”} \\
\qquad \wedge \; i' \in 0 \,..\, 1000 \\
\qquad \wedge \; pc' = \text{“}middle\text{”}\boxed{)} \\
\vee
\end{array}
$$

As if these parentheses were there.

$$
\begin{aligned}
\lor\ (\quad & pc = \text{“}middle\text{”} \\
\land\ & i' = i + 1 \\
\land\ & pc' = \text{“}done\text{”}\ )
\end{aligned}
$$

As if these parentheses were there.

Let’s do the same thing with this subformula.

$$
\begin{array}{ll}
\lor & \land\ pc = \text{“}middle\text{”} \\
 & \land\ i' = i + 1 \\
 & \land\ pc' = \text{“}done\text{”}
\end{array}
$$

As if these parentheses were there.

Let’s do the same thing with this subformula.

$$
\begin{array}{ll}
 & \land\ pc = \text{“}start\text{”} \\
 & \land\ i' \in 0\,..\,1000 \\
 & \land\ pc' = \text{“}middle\text{”} \\
\lor & \land\ pc = \text{“}middle\text{”} \\
 & \land\ i' = i + 1 \\
 & \land\ pc' = \text{“}done\text{”}
\end{array}
$$

Let’s do the same thing

$$
\begin{array}{ll}
 & \land \ pc = \text{“}start\text{”} \\
 & \land \ i' \in 0 \,..\, 1000 \\
 & \land \ pc' = \text{“}middle\text{”} \\
\boxed{\lor} & \land \ pc = \text{“}middle\text{”} \\
 & \land \ i' = i + 1 \\
 & \land \ pc' = \text{“}done\text{”}
\end{array}
$$

Let’s do the same thing for the *or*.

$$
\begin{array}{l}
\boxed{\begin{array}{l}\lor\\ \\ \\ \lor\\ \\ \\ \end{array}}
\begin{array}{l}
\land\ pc = \text{"}start\text{"}\\
\land\ i' \in 0 \,..\, 1000\\
\land\ pc' = \text{"}middle\text{"}\\
\land\ pc = \text{"}middle\text{"}\\
\land\ i' = i + 1\\
\land\ pc' = \text{"}done\text{"}
\end{array}
\end{array}
$$

Let's do the same thing for the *or*.

TLA+ also allows bulleted *or* lists.

$$\left(\begin{array}{l} \lor \land\ pc = \text{“}start\text{”} \\ \quad \land\ i' \in 0 \,..\, 1000 \\ \quad \land\ pc' = \text{“}middle\text{”} \\ \lor \land\ pc = \text{“}middle\text{”} \\ \quad \land\ i' = i + 1 \\ \quad \land\ pc' = \text{“}done\text{”} \end{array} \right)$$

Let's do the same thing for the *or*.

TLA+ also allows bulleted *or* lists.

There are implicit parentheses around the formula.

$$
\begin{array}{ll}
\lor & \land \; pc = \text{“}start\text{”} \\
& \land \; i' \in 0 \,..\, 1000 \\
& \land \; pc' = \text{“}middle\text{”} \\
\lor & \land \; pc = \text{“}middle\text{”} \\
& \land \; i' = i + 1 \\
& \land \; pc' = \text{“}done\text{”}
\end{array}
$$

Let's do the same thing for the *or*.

TLA+ also allows bulleted *or* lists.

There are implicit parentheses around the formula.

$$
\begin{array}{ll}
\lor & \land\ pc = \text{“}start\text{”} \\
 & \land\ i' \in 0\,..\,1000 \\
 & \land\ pc' = \text{“}middle\text{”} \\
\lor & \land\ pc = \text{“}middle\text{”} \\
 & \land\ i' = i + 1 \\
 & \land\ pc' = \text{“}done\text{”}
\end{array}
$$

Let’s compare the TLA+ formula with the corresponding C code.

Let’s do the same thing for the *or*.

TLA+ also allows bulleted *or* lists.

There are implicit parentheses around the formula.

Now let’s compare the TLA+ formula with the corresponding C code, which. . .

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

$$
\begin{aligned}
\lor\ & \land\ pc = \text{“}start\text{”} \\
& \land\ i' \in 0 \,..\, 1000 \\
& \land\ pc' = \text{“}middle\text{”} \\
\lor\ & \land\ pc = \text{“}middle\text{”} \\
& \land\ i' = i + 1 \\
& \land\ pc' = \text{“}done\text{”}
\end{aligned}
$$

is the C code without the declaration of i.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

$$
\begin{array}{l}
\lor \;\land\; pc = \text{“}start\text{”} \\
\quad\;\; \land\; i' \in 0 \,..\, 1000 \\
\quad\;\; \land\; pc' = \text{“}middle\text{”} \\
\lor \;\land\; pc = \text{“}middle\text{”} \\
\quad\;\; \land\; i' = i + 1 \\
\quad\;\; \land\; pc' = \text{“}done\text{”}
\end{array}
$$

The C code probably seems simpler because it's more familiar.

is the C code without the declaration of i.

The C code probably seems simpler than the TLA+ formula because it's more familiar to you.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

$$
\begin{aligned}
\lor\ & \land\ pc = \text{“}start\text{”} \\
& \land\ i' \in 0 \,..\, 1000 \\
& \land\ pc' = \text{“}middle\text{”} \\
\lor\ & \land\ pc = \text{“}middle\text{”} \\
& \land\ i' = i + 1 \\
& \land\ pc' = \text{“}done\text{”}
\end{aligned}
$$

The C code probably seems simpler because it's more familiar.

But it isn't really simpler.

is the C code without the declaration of i.

The C code probably seems simpler than the TLA+ formula because it's more familiar to you.

But the C code isn't really simpler.

```
int i ;
void main()
  { i = someNumber() ;
    i = i + 1 ;
  }
```

$$\begin{array}{l} \lor \land\ pc = \text{"}start\text{"} \\ \quad \land\ i' \in 0 \,..\, 1000 \\ \quad \land\ pc' = \text{"}middle\text{"} \\ \lor \land\ pc = \text{"}middle\text{"} \\ \quad \land\ i' = i + 1 \\ \quad \land\ pc' = \text{"}done\text{"} \end{array}$$

= in TLA$^+$ means equality, as in $2 + 2 = 4$.

is the C code without the declaration of i.

The C code probably seems simpler than the TLA+ formula because it's more familiar to you.

But the C code isn't really simpler.

For one thing, the equal sign in TLA+ means equality, just as in grammar school, when you wrote two plus two equals 4.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

$$
\begin{array}{l}
\lor \land pc = \text{"}start\text{"} \\
 \land i' \in 0 \,..\, 1000 \\
 \land pc' = \text{"}middle\text{"} \\
\lor \land pc = \text{"}middle\text{"} \\
 \land i' = i + 1 \\
 \land pc' = \text{"}done\text{"}
\end{array}
$$

= in TLA+ means equality, as in $2 + 2 = 4$.

= in C means assignment, which isn't so simple.

The equals sign in C means assignment, which isn't so simple.

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

$$
\begin{array}{ll}
\lor & \land \ pc = \text{“}start\text{”} \\
 & \land \ i' \in 0 \,..\, 1000 \\
 & \land \ pc' = \text{“}middle\text{”} \\
\lor & \land \ pc = \text{“}middle\text{”} \\
 & \land \ i' = i + 1 \\
 & \land \ pc' = \text{“}done\text{”}
\end{array}
$$

The big difference between math and C:

Math is much more expressive.

The equals sign in C means assignment, which isn't so simple.

But the big difference between math and C is that math is much, much more expressive.

```
int i ;
void main()
  { i = someNumber() ;
    i = i + 1 ;
  }
```

$$
\begin{array}{ll}
\vee & \wedge\ pc = \text{“}start\text{”} \\
 & \wedge\ i' \in 0 \,..\, 1000 \\
 & \wedge\ pc' = \text{“}middle\text{”} \\
\vee & \wedge\ pc = \text{“}middle\text{”} \\
 & \wedge\ i' = i + 1 \\
 & \wedge\ pc' = \text{“}done\text{”}
\end{array}
$$

What about $someNumber$?

The equals sign in C means assignment, which isn't so simple.

But the big difference between math and C is that math is much, much more expressive.

What about $someNumber$?

```
int i ;
void main()
    { i = someNumber() ;
      i = i + 1 ;
    }
```

$$\begin{array}{ll} \vee & \wedge\ pc = \text{"}start\text{"} \\ & \wedge\ i' \in 0\,..\,1000 \\ & \wedge\ pc' = \text{"}middle\text{"} \\ \vee & \wedge\ pc = \text{"}middle\text{"} \\ & \wedge\ i' = i + 1 \\ & \wedge\ pc' = \text{"}done\text{"} \end{array}$$

Its execution is nondeterministic.

The equals sign in C means assignment, which isn't so simple.

But the big difference between math and C is that math is much, much more expressive.

What about $someNumber$?

Its execution is nondeterministic.

```
int i ;
void main()
  { i = someNumber() ;
    i = i + 1 ;
  }
```

$$
\begin{array}{ll}
\lor & \land\ pc = \text{"}start\text{"} \\
 & \land\ i' \in 0\,..\,1000 \\
 & \land\ pc' = \text{"}middle\text{"} \\
\lor & \land\ pc = \text{"}middle\text{"} \\
 & \land\ i' = i + 1 \\
 & \land\ pc' = \text{"}done\text{"}
\end{array}
$$

We need nondeterminism to describe systems,

We need nondeterminism like this to describe systems,

```
int i ;
void main()
  { i = someNumber() ;
    i = i + 1 ;
  }
```

$$
\begin{array}{ll}
\vee & \wedge\ pc = \text{"}start\text{"} \\
 & \wedge\ i' \in 0\,..\,1000 \\
 & \wedge\ pc' = \text{"}middle\text{"} \\
\vee & \wedge\ pc = \text{"}middle\text{"} \\
 & \wedge\ i' = i + 1 \\
 & \wedge\ pc' = \text{"}done\text{"}
\end{array}
$$

We need nondeterminism to describe systems,
because we can't predict in what order things happen.

We need nondeterminism like this to describe systems,
because we can't predict in what order things happen.

```
int i ;
void main()
  { i = someNumber() ;
    i = i + 1 ;
  }
```

$$
\begin{array}{ll}
\vee & \wedge\ pc = \text{“}start\text{”} \\
 & \wedge\ \boxed{i' \in 0\,..\,1000} \\
 & \wedge\ pc' = \text{“}middle\text{”} \\
\vee & \wedge\ pc = \text{“}middle\text{”} \\
 & \wedge\ i' = i + 1 \\
 & \wedge\ pc' = \text{“}done\text{”}
\end{array}
$$

Look how easily it’s described in math.

We need nondeterminism like this to describe systems, because we can’t predict in what order things happen.

Look how easily nondeterminism is described in math.

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

$$\begin{array}{l} \lor \ \land\ pc = \text{“}start\text{”} \\ \quad\ \land\ \boxed{i' \in 0\,..\,1000} \\ \quad\ \land\ pc' = \text{“}middle\text{”} \\ \lor \ \land\ pc = \text{“}middle\text{”} \\ \quad\ \land\ i' = i + 1 \\ \quad\ \land\ pc' = \text{“}done\text{”} \end{array}$$

Look how easily it's described in math.

Programming languages weren't designed to express nondeterminism.

Commonly used programming languages were not designed to express nondeterminism.

```
int i ;
void main()
  { i = someNumber() ;
    i = i + 1 ;
  }
```

$$\begin{array}{ll} \vee & \wedge\ pc = \text{“}start\text{”} \\ & \wedge\ i' \in 0 \,..\, 1000 \\ & \wedge\ pc' = \text{“}middle\text{”} \\ \vee & \wedge\ pc = \text{“}middle\text{”} \\ & \wedge\ i' = i + 1 \\ & \wedge\ pc' = \text{“}done\text{”} \end{array}$$

They lack more than constructs for nondeterminism.

Commonly used programming languages were not designed to express nondeterminism.

Programming languages lack much more than constructs for nondeterminism.

```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

$$
\begin{array}{ll}
\lor & \land\ pc = \text{“}start\text{”} \\
 & \land\ i' \in 0 \mathrel{..} 1000 \\
 & \land\ pc' = \text{“}middle\text{”} \\
\lor & \land\ pc = \text{“}middle\text{”} \\
 & \land\ i' = i + 1 \\
 & \land\ pc' = \text{“}done\text{”}
\end{array}
$$

They lack more than constructs for nondeterminism.

Programming languages don’t abstract above the code level.

Commonly used programming languages were not designed to express nondeterminism.

Programming languages lack much more than constructs for nondeterminism.

They don’t let you abstract above the code level.

$$
\begin{array}{l}
\lor \;\; \land \; pc = \text{“}start\text{”} \\
 \;\; \land \; i' \in 0 \,..\, 1000 \\
 \;\; \land \; pc' = \text{“}middle\text{”} \\
\lor \;\; \land \; pc = \text{“}middle\text{”} \\
 \;\; \land \; i' = i + 1 \\
 \;\; \land \; pc' = \text{“}done\text{”}
\end{array}
$$

It’s important to remember that this is a formula

It’s important to remember that this is a formula,

$$
\begin{array}{l}
\lor \ \land \ pc = \text{“}start\text{”} \\
\quad\ \land \ i' \in 0 \,..\, 1000 \\
\quad\ \land \ pc' = \text{“}middle\text{”} \\
\lor \ \land \ pc = \text{“}middle\text{”} \\
\quad\ \land \ i' = i + 1 \\
\quad\ \land \ pc' = \text{“}done\text{”}
\end{array}
$$

It’s important to remember that this is a formula,
not a sequence of commands.

It’s important to remember that this is a formula,
not a sequence of commands.

$$\begin{array}{l} \boxed{\begin{array}{l} \lor \\ \\ \\ \lor \\ \\ \\ \end{array}} \begin{array}{l} \land\ pc = \text{“}start\text{”} \\ \land\ i' \in 0\,..\,1000 \\ \land\ pc' = \text{“}middle\text{”} \\ \land\ pc = \text{“}middle\text{”} \\ \land\ i' = i + 1 \\ \land\ pc' = \text{“}done\text{”} \end{array} \end{array}$$

$\lor$ is commutative

It's important to remember that this is a formula, not a sequence of commands.

or is commutative

$$
\begin{array}{l}
\lor \boxed{\begin{array}{l} \land\ pc = \text{“}start\text{”} \\ \land\ i' \in 0 \,..\, 1000 \\ \land\ pc' = \text{“}middle\text{”} \end{array}} \\
\lor \boxed{\begin{array}{l} \land\ pc = \text{“}middle\text{”} \\ \land\ i' = i + 1 \\ \land\ pc' = \text{“}done\text{”} \end{array}}
\end{array}
$$

$\lor$ is commutative, so interchanging these sub-formulas

It's important to remember that this is a formula, not a sequence of commands.

or is commutative so interchanging these sub-formulas

$$\begin{array}{l}\lor\ \boxed{\begin{array}{l}\land\ pc = \text{“}start\text{”}\\ \land\ i' \in 0\,..\,1000\\ \land\ pc' = \text{“}middle\text{”}\end{array}}\\ \lor\ \boxed{\begin{array}{l}\land\ pc = \text{“}middle\text{”}\\ \land\ i' = i + 1\\ \land\ pc' = \text{“}done\text{”}\end{array}}\end{array} \qquad \begin{array}{l}\lor\ \boxed{\begin{array}{l}\land\ pc = \text{“}middle\text{”}\\ \land\ i' = i + 1\\ \land\ pc' = \text{“}done\text{”}\end{array}}\\ \lor\ \boxed{\begin{array}{l}\land\ pc = \text{“}start\text{”}\\ \land\ i' \in 0\,..\,1000\\ \land\ pc' = \text{“}middle\text{”}\end{array}}\end{array}$$

$\lor$ is commutative, so interchanging these sub-formulas yields an equivalent formula.

It’s important to remember that this is a formula, not a sequence of commands.

or is commutative so interchanging these sub-formulas yields an equivalent formula.

$$\begin{array}{ll}
\lor \land pc = \text{“}start\text{”} & \lor \land pc = \text{“}middle\text{”} \\
\quad \land i' \in 0\,..\,1000 & \quad \land i' = i + 1 \\
\quad \land pc' = \text{“}middle\text{”} & \quad \land pc' = \text{“}done\text{”} \\
\lor \land pc = \text{“}middle\text{”} & \lor \land pc = \text{“}start\text{”} \\
\quad \land i' = i + 1 & \quad \land i' \in 0\,..\,1000 \\
\quad \land pc' = \text{“}done\text{”} & \quad \land pc' = \text{“}middle\text{”}
\end{array}$$

$\lor$ is commutative, so interchanging these sub-formulas yields an equivalent formula.

It's important to remember that this is a formula, not a sequence of commands.

or is commutative so interchanging these sub-formulas yields an equivalent formula.

$$
\begin{array}{l}
\lor \ \boxed{\begin{array}{l}\land\\ \land\\ \land\end{array}} \begin{array}{l} pc = \text{“}start\text{”} \\ i' \in 0\,..\,1000 \\ pc' = \text{“}middle\text{”}\end{array} \\
\lor \ \land \ pc = \text{“}middle\text{”} \\
\quad\ \land \ i' = i + 1 \\
\quad\ \land \ pc' = \text{“}done\text{”}
\end{array}
$$

$\land$ is also commutative

It's important to remember that this is a formula,
not a sequence of commands.

or is commutative so interchanging these sub-formulas
yields an equivalent formula.

and is also commutative

$$
\begin{array}{ll}
\lor & \boxed{\begin{array}{l} \land\ pc = \text{“}start\text{”} \\ \land\ i' \in 0\,..\,1000 \end{array}} \\
 & \boxed{\land\ pc' = \text{“}middle\text{”}} \\
\lor & \land\ pc = \text{“}middle\text{”} \\
 & \land\ i' = i + 1 \\
 & \land\ pc' = \text{“}done\text{”}
\end{array}
$$

$\land$ is also commutative, so interchanging these sub-formulas

It's important to remember that this is a formula, not a sequence of commands.

or is commutative so interchanging these sub-formulas yields an equivalent formula.

and is also commutative so interchanging these sub-formulas

$$\begin{array}{ll}
\lor & \boxed{\begin{array}{l}\land\ pc = \text{“}start\text{”} \\ \land\ i' \in 0\,..\,1000\end{array}} \\
 & \boxed{\land\ pc' = \text{“}middle\text{”}} \\
\lor & \land\ pc = \text{“}middle\text{”} \\
 & \land\ i' = i + 1 \\
 & \land\ pc' = \text{“}done\text{”}
\end{array}
\qquad
\begin{array}{ll}
\lor & \boxed{\land\ pc' = \text{“}middle\text{”}} \\
 & \boxed{\begin{array}{l}\land\ pc = \text{“}start\text{”} \\ \land\ i' \in 0\,..\,1000\end{array}} \\
\lor & \land\ pc = \text{“}middle\text{”} \\
 & \land\ i' = i + 1 \\
 & \land\ pc' = \text{“}done\text{”}
\end{array}$$

$\land$ is also commutative, so interchanging these sub-formulas also yields an equivalent formula.

It's important to remember that this is a formula, not a sequence of commands.

or is commutative so interchanging these sub-formulas yields an equivalent formula.

and is also commutative so interchanging these sub-formulas also yields an equivalent formula.

$$\begin{array}{ll}
\lor \land pc = \text{"start"} & \lor \land pc' = \text{"middle"} \\
\quad \land i' \in 0 \,..\, 1000 & \quad \land pc = \text{"start"} \\
\quad \land pc' = \text{"middle"} & \quad \land i' \in 0 \,..\, 1000 \\
\lor \land pc = \text{"middle"} & \lor \land pc = \text{"middle"} \\
\quad \land i' = i + 1 & \quad \land i' = i + 1 \\
\quad \land pc' = \text{"done"} & \quad \land pc' = \text{"done"}
\end{array}$$

$\land$ is also commutative, so interchanging these sub-formulas also yields an equivalent formula.

It's important to remember that this is a formula, not a sequence of commands.

or is commutative so interchanging these sub-formulas yields an equivalent formula.

and is also commutative so interchanging these sub-formulas also yields an equivalent formula.

$$
\begin{array}{ll}
\lor & \land\ pc = \text{“}start\text{”} \\
 & \land\ i' \in 0 \,..\, 1000 \\
 & \land\ pc' = \text{“}middle\text{”} \\
\lor & \land\ pc = \text{“}middle\text{”} \\
 & \land\ i' = i + 1 \\
 & \land\ pc' = \text{“}done\text{”}
\end{array}
$$

It's important to remember that this is a formula,
not a sequence of commands.

or is commutative so interchanging these sub-formulas
yields an equivalent formula.

and is also commutative so interchanging these sub-formulas
also yields an equivalent formula.

THE COMPLETE TLA$^+$ SPEC

The complete TLA+ Specification.

The Complete Spec in Math

Initial-state formula: $(i = 0) \land (pc = \text{“}start\text{”})$

Next-state formula:

$$
\begin{array}{l}
\lor \;\; \land \; pc = \text{“}start\text{”} \\
 \;\; \land \; i' \in 0 \,..\, 1000 \\
 \;\; \land \; pc' = \text{“}middle\text{”} \\
\lor \;\; \land \; pc = \text{“}middle\text{”} \\
 \;\; \land \; i' = i + 1 \\
 \;\; \land \; pc' = \text{“}done\text{”}
\end{array}
$$

This is the complete specification in mathematics.

The Complete Spec in Math

Initial-state formula: $(i = 0) \land (pc = \text{“}start\text{”})$

This is the complete specification in mathematics.

The initial-state formula can also be written like this.

The Complete Spec in Math

Initial-state formula: $\land \; i = 0$
$\land \; pc = \text{“}start\text{”}$

This is the complete specification in mathematics.

The initial-state formula can also be written like this.

But this

The Complete Spec in Math

Initial-state formula: $(i = 0) \land (pc = \text{“}start\text{”})$

This is the complete specification in mathematics.

The initial-state formula can also be written like this.

But this takes less space.

The Complete Spec in Math

Initial-state formula: $(i = 0) \land (pc = \text{“}start\text{”})$

Next-state formula:

$$
\begin{aligned}
\lor\ & \land\ pc = \text{“}start\text{”} \\
& \land\ i' \in 0 \,..\, 1000 \\
& \land\ pc' = \text{“}middle\text{”} \\
\lor\ & \land\ pc = \text{“}middle\text{”} \\
& \land\ i' = i + 1 \\
& \land\ pc' = \text{“}done\text{”}
\end{aligned}
$$

A TLA+ specification has some additional stuff.

MODULE *SimpleProgram*

A TLA$^+$ spec appears in a module.

MODULE *SimpleProgram*

A TLA^{+} spec appears in a module.

A TLA^{+} spec appears in a module.

MODULE $\boxed{SimpleProgram}$

This module is named $SimpleProgram$.

A TLA+ spec appears in a module.

This module is named $SimpleProgram$.

MODULE *SimpleProgram*

EXTENDS *Integers*

A TLA$^+$ spec appears in a module.

This module is named *SimpleProgram*.

This EXTENDS statement

MODULE *SimpleProgram*

EXTENDS *Integers*

Imports arithmetic operators like $+$ and $..$

A TLA+ spec appears in a module.

This module is named *SimpleProgram*.

This EXTENDS statement imports arithmetic operators like plus and dot-dot.

MODULE *SimpleProgram*

EXTENDS *Integers*

VARIABLES i, pc

Identifiers must be defined or declared before they're used.

MODULE *SimpleProgram*

EXTENDS *Integers*

VARIABLES i, pc

Declares the variables.

Identifiers must be defined or declared before they're used.

This statement declares the variables.

MODULE *SimpleProgram*

EXTENDS *Integers*

VARIABLES i, pc

$$Init \triangleq (pc = \text{"start"}) \land (i = 0)$$

Identifiers must be defined or declared before they're used.

This statement declares the variables.

This is a definition.

MODULE $SimpleProgram$

EXTENDS $Integers$

VARIABLES i, pc

$\boxed{Init \triangleq}$ $(pc = \text{"start"}) \wedge (i = 0)$

Defines $Init$ to be equal to

It defines $Init$ to be equal to

MODULE *SimpleProgram*

EXTENDS *Integers*

VARIABLES i, pc

$$Init \triangleq \boxed{(pc = \text{"start"}) \land (i = 0)}$$

Defines $Init$ to be equal to the initial formula.

It defines $Init$ to be equal to the initial formula.

$$\text{MODULE } SimpleProgram$$

EXTENDS *Integers*

VARIABLES i, pc

$$Init \triangleq (pc = \text{“start”}) \land (i = 0)$$

$$
\begin{aligned}
Next \triangleq\ & \lor \land pc = \text{“start”} \\
& \quad \land i' \in 0 \,..\, 1000 \\
& \quad \land pc' = \text{“middle”} \\
& \lor \land pc = \text{“middle”} \\
& \quad \land i' = i + 1 \\
& \quad \land pc' = \text{“done”}
\end{aligned}
$$

It defines *Init* to be equal to the initial formula.

Similarly, this statement

MODULE *SimpleProgram*

EXTENDS *Integers*

VARIABLES i, pc

$$Init \triangleq (pc = \text{"start"}) \land (i = 0)$$

$$\boxed{Next \triangleq} \quad \begin{array}{l} \lor \land pc = \text{"start"} \\ \quad \land i' \in 0 \,..\, 1000 \\ \quad \land pc' = \text{"middle"} \\ \lor \land pc = \text{"middle"} \\ \quad \land i' = i + 1 \\ \quad \land pc' = \text{"done"} \end{array}$$

Defines $Next$ to equal

It defines $Init$ to be equal to the initial formula.

Similarly, this statement defines $Next$ to equal

MODULE *SimpleProgram*

EXTENDS *Integers*

VARIABLES i, pc

$$Init \triangleq (pc = \text{"start"}) \land (i = 0)$$

$$Next \triangleq \boxed{\begin{array}{l} \lor \land pc = \text{"start"} \\ \quad \land i' \in 0 \,..\, 1000 \\ \quad \land pc' = \text{"middle"} \\ \lor \land pc = \text{"middle"} \\ \quad \land i' = i + 1 \\ \quad \land pc' = \text{"done"} \end{array}}$$

Defines $Next$ to equal the next-state formula.

It defines $Init$ to be equal to the initial formula.

Similarly, this statement defines $Next$ to equal the next-state formula.

MODULE *SimpleProgram*

EXTENDS *Integers*

VARIABLES i, pc

$$Init \triangleq (pc = \text{“start”}) \land (i = 0)$$

$$
\begin{aligned}
Next \triangleq \;& \lor \land pc = \text{“start”} \\
& \land i' \in 0 \,..\, 1000 \\
& \land pc' = \text{“middle”} \\
& \lor \land pc = \text{“middle”} \\
& \land i' = i + 1 \\
& \land pc' = \text{“done”}
\end{aligned}
$$

It defines *Init* to be equal to the initial formula.

Similarly, this statement defines *Next* to equal the next-state formula.

MODULE *SimpleProgram*

EXTENDS *Integers*

VARIABLES i, pc

$$\boxed{Init} \triangleq (pc = \text{"start"}) \wedge (i = 0)$$

You can use any names.

$$\boxed{Next} \triangleq \begin{array}{l} \vee \wedge pc = \text{"start"} \\ \quad \wedge i' \in 0 \,..\, 1000 \\ \quad \wedge pc' = \text{"middle"} \\ \vee \wedge pc = \text{"middle"} \\ \quad \wedge i' = i + 1 \\ \quad \wedge pc' = \text{"done"} \end{array}$$

It defines $Init$ to be equal to the initial formula.

Similarly, this statement defines $Next$ to equal the next-state formula.

You can use any names instead of $Init$ and $Next$,

MODULE *SimpleProgram*

EXTENDS *Integers*

VARIABLES i, pc

$$\boxed{Init} \triangleq (pc = \text{"start"}) \land (i = 0)$$

You can use any names.

$$\boxed{Next} \triangleq \begin{array}{l} \lor \land pc = \text{"start"} \\ \quad \land i' \in 0 \,..\, 1000 \\ \quad \land pc' = \text{"middle"} \\ \lor \land pc = \text{"middle"} \\ \quad \land i' = i + 1 \\ \quad \land pc' = \text{"done"} \end{array}$$

These are conventional.

It defines $Init$ to be equal to the initial formula.

Similarly, this statement defines $Next$ to equal the next-state formula.

You can use any names instead of $Init$ and $Next$,
But they are the ones normally used by convention.

MODULE *SimpleProgram*

EXTENDS *Integers*

VARIABLES i, pc

$$Init \triangleq (pc = \text{“start”}) \land (i = 0)$$

$$
\begin{aligned}
Next \triangleq\ & \lor \land pc = \text{“start”} \\
& \quad \land i' \in 0 \,..\, 1000 \\
& \quad \land pc' = \text{“middle”} \\
& \lor \land pc = \text{“middle”} \\
& \quad \land i' = i + 1 \\
& \quad \land pc' = \text{“done”}
\end{aligned}
$$

It defines $Init$ to be equal to the initial formula.

Similarly, this statement defines $Next$ to equal the next-state formula.

You can use any names instead of $Init$ and $Next$,
But they are the ones normally used by convention.

This is the pretty-printed version of the spec.

```
---------------- MODULE SimpleProgram ----------------
EXTENDS Integers
VARIABLES i, pc

Init == (pc = "start") /\ (i = 0)

Next == \/ /\ pc = "start"
           /\ i' \in 0..1000
           /\ pc' = "middle"
        \/ /\ pc = "middle"
           /\ i' = i + 1
           /\ pc' = "done"
=====================================================
```

Here is how you type the spec into the TLA+ Toolbox.

On command, the Toolbox will display

$$\textsc{module}\ \mathit{SimpleProgram}$$

$\textsc{extends}\ \mathit{Integers}$

$\textsc{variables}\ i,\ pc$

$$\mathit{Init} \triangleq (pc = \text{"start"}) \land (i = 0)$$

$$\begin{aligned}
\mathit{Next} \triangleq\ & \lor\ \land\ pc = \text{"start"} \\
& \quad\ \land\ i' \in 0\,..\,1000 \\
& \quad\ \land\ pc' = \text{"middle"} \\
& \lor\ \land\ pc = \text{"middle"} \\
& \quad\ \land\ i' = i + 1 \\
& \quad\ \land\ pc' = \text{"done"}
\end{aligned}$$

this pretty-printed version.

DECOMPOSING LARGE SPECS

Decomposing large specs.

The next-state formula can be 100s of lines.

For real specs, the next-state formula can be hundreds or even thousands of lines.

The next-state formula can be 100s of lines.

We can understand a big formula by splitting it into smaller parts.

For real specs, the next-state formula can be hundreds or even thousands of lines.

We can understand a big formula by splitting it into smaller parts.

The next-state formula can be 100s of lines.

We can understand a big formula by splitting it into smaller parts.

Math has a simple and powerful way to do that:

For real specs, the next-state formula can be hundreds or even thousands of lines.

We can understand a big formula by splitting it into smaller parts.

Math has a simple and very powerful way to do that:

The next-state formula can be 100s of lines.

We can understand a big formula by splitting it into smaller parts.

Math has a simple and powerful way to do that:

Using definitions.

For real specs, the next-state formula can be hundreds or even thousands of lines.

We can understand a big formula by splitting it into smaller parts.

Math has a simple and very powerful way to do that: Using definitions.

$$\text{MODULE } \mathit{SimpleProgram}$$

EXTENDS $\mathit{Integers}$

VARIABLES i, pc

$$\mathit{Init} \triangleq (pc = \text{"start"}) \land (i = 0)$$

$$\begin{aligned}\mathit{Next} \triangleq\ & \lor \land pc = \text{"start"} \\ & \quad \land i' \in 0 \,..\, 1000 \\ & \quad \land pc' = \text{"middle"} \\ & \lor \land pc = \text{"middle"} \\ & \quad \land i' = i + 1 \\ & \quad \land pc' = \text{"done"}\end{aligned}$$

This spec is too simple to need splitting into parts, but let's do it anyway.

An obvious way to decompose this spec is

$$\text{MODULE } \mathit{SimpleProgram}$$

EXTENDS *Integers*

VARIABLES i, pc

$$Init \triangleq (pc = \text{"start"}) \land (i = 0)$$

$$Next \triangleq \lor \boxed{\begin{array}{l} \land\ pc = \text{"start"} \\ \land\ i' \in 0 \,..\, 1000 \\ \land\ pc' = \text{"middle"} \end{array}}$$
$$\qquad\qquad \lor \boxed{\begin{array}{l} \land\ pc = \text{"middle"} \\ \land\ i' = i + 1 \\ \land\ pc' = \text{"done"} \end{array}}$$

This spec is too simple to need splitting into parts, but let's do it anyway.

An obvious way to decompose this spec is

by giving names to these two subformulas.

We could call them anything,

$$\text{MODULE } \mathit{SimpleProgram}$$

EXTENDS *Integers*

VARIABLES i, pc

$$Init \triangleq (pc = \text{“start”}) \land (i = 0)$$

$$
\begin{aligned}
Next \triangleq\ & \lor \boxed{\begin{aligned}&\land pc = \text{“start”}\\ &\land i' \in 0 \,..\, 1000\\ &\land pc' = \text{“middle”}\end{aligned}} \quad \text{Fred}\\
& \lor \boxed{\begin{aligned}&\land pc = \text{“middle”}\\ &\land i' = i + 1\\ &\land pc' = \text{“done”}\end{aligned}} \quad \text{Mary}
\end{aligned}
$$

This spec is too simple to need splitting into parts, but let's do it anyway.

An obvious way to decompose this spec is

by giving names to these two subformulas.

We could call them anything, say *Fred* and *Mary*.
But more descriptive names are better, such as

MODULE *SimpleProgram*

EXTENDS *Integers*

VARIABLES i, pc

$$Init \triangleq (pc = \text{"start"}) \land (i = 0)$$

$$Next \triangleq \lor \boxed{\begin{array}{l} \land\ pc = \text{"start"} \\ \land\ i' \in 0 \,..\, 1000 \\ \land\ pc' = \text{"middle"} \end{array}} \quad \text{Pick}$$
$$\qquad\qquad \lor \boxed{\begin{array}{l} \land\ pc = \text{"middle"} \\ \land\ i' = i + 1 \\ \land\ pc' = \text{"done"} \end{array}} \quad \text{Add1}$$

This spec is too simple to need splitting into parts, but let's do it anyway.

An obvious way to decompose this spec is

by giving names to these two subformulas.

We could call them anything, say *Fred* and *Mary*.
But more descriptive names are better, such as Pick and Add1

$$
\begin{array}{lll}
Next \triangleq & \lor & \land\ pc = \text{"start"} \\
& & \land\ i' \in 0 \,..\, 1000 \\
& & \land\ pc' = \text{"middle"} \\
& \lor & \land\ pc = \text{"middle"} \\
& & \land\ i' = i + 1 \\
& & \land\ pc' = \text{"done"}
\end{array}
$$

So let's replace this definition of $Next$

$$
\begin{aligned}
Pick \;&\triangleq\; \land\ pc = \text{"start"} \\
&\;\; \land\ i' \in 0 \,..\, 1000 \\
&\;\; \land\ pc' = \text{"middle"} \\[1em]
Add1 \;&\triangleq\; \land\ pc = \text{"middle"} \\
&\;\; \land\ i' = i + 1 \\
&\;\; \land\ pc' = \text{"done"} \\[1em]
Next \;&\triangleq\; Pick \lor Add1
\end{aligned}
$$

with these three definitions.

$$\boxed{\begin{aligned} Pick \;\triangleq\; & \land\, pc = \text{"start"} \\ & \land\, i' \in 0 \,..\, 1000 \\ & \land\, pc' = \text{"middle"} \end{aligned}}$$

$$\begin{aligned} Add1 \;\triangleq\; & \land\, pc = \text{"middle"} \\ & \land\, i' = i + 1 \\ & \land\, pc' = \text{"done"} \end{aligned}$$

$$Next \;\triangleq\; Pick \lor Add1$$

with these three definitions.

We define $Pick$

$$
\begin{aligned}
Pick \;&\triangleq\; \land\ pc = \text{“start”} \\
&\;\; \land\ i' \in 0 \,..\, 1000 \\
&\;\; \land\ pc' = \text{“middle”}
\end{aligned}
$$

$$
\boxed{
\begin{aligned}
Add1 \;&\triangleq\; \land\ pc = \text{“middle”} \\
&\;\; \land\ i' = i + 1 \\
&\;\; \land\ pc' = \text{“done”}
\end{aligned}
}
$$

$$
Next \;\triangleq\; Pick \lor Add1
$$

with these three definitions.

We define $Pick$ and $Add1$

$$
\begin{aligned}
Pick \;&\triangleq\; \land\ pc = \text{“start”} \\
&\;\; \land\ i' \in 0 \,..\, 1000 \\
&\;\; \land\ pc' = \text{“middle”}
\end{aligned}
$$

$$
\begin{aligned}
Add1 \;&\triangleq\; \land\ pc = \text{“middle”} \\
&\;\; \land\ i' = i + 1 \\
&\;\; \land\ pc' = \text{“done”}
\end{aligned}
$$

$$\boxed{Next \;\triangleq\; Pick \lor Add1}$$

with these three definitions.

We define $Pick$ and $Add1$ and then define $Next$ to equal $Pick$ or $Add1$

$$
\begin{aligned}
Pick &\triangleq \land\ pc = \text{"start"} \\
&\quad\ \land\ i' \in 0 \,..\, 1000 \\
&\quad\ \land\ pc' = \text{"middle"}
\end{aligned}
$$

$$
\begin{aligned}
Add1 &\triangleq \land\ pc = \text{"middle"} \\
&\quad\ \land\ i' = i + 1 \\
&\quad\ \land\ pc' = \text{"done"}
\end{aligned}
$$

$$
Next \triangleq Pick \lor Add1
$$

with these three definitions.

We define $Pick$ and $Add1$ and then define $Next$ to equal $Pick$ or $Add1$ This definition of $Next$

$$Pick \triangleq \land\ pc = \text{"start"}$$
$$\qquad\quad \land\ i' \in 0 \,..\, 1000$$
$$\qquad\quad \land\ pc' = \text{"middle"}$$

$$Add1 \triangleq \land\ pc = \text{"middle"}$$
$$\qquad\quad \land\ i' = i + 1$$
$$\qquad\quad \land\ pc' = \text{"done"}$$

$$Next \triangleq Pick \lor Add1$$

$$\begin{aligned} Next \triangleq\ & \lor\ \land\ pc = \text{"start"} \\ & \quad\ \land\ i' \in 0 \,..\, 1000 \\ & \quad\ \land\ pc' = \text{"middle"} \\ & \lor\ \land\ pc = \text{"middle"} \\ & \quad\ \land\ i' = i + 1 \\ & \quad\ \land\ pc' = \text{"done"} \end{aligned}$$

These are equivalent definitions of $Next$.

Is completely equivalent to our original definition.

$$
\begin{array}{lcl}
Pick & \triangleq & \land\ pc = \text{“start”} \\
 & & \land\ i' \in 0\,..\,1000 \\
 & & \land\ pc' = \text{“middle”} \\[1em]
Add1 & \triangleq & \land\ pc = \text{“middle”} \\
 & & \land\ i' = i + 1 \\
 & & \land\ pc' = \text{“done”} \\[1em]
Next & \triangleq & Pick \lor Add1
\end{array}
$$

$$
\begin{array}{lcl}
Next & \triangleq & \lor\ \land\ pc = \text{“}start\text{”} \\
 & & \ \land\ i' \in 0\,..\,1000 \\
 & & \ \land\ pc' = \text{“}middle\text{”} \\
 & & \lor\ \land\ pc = \text{“}middle\text{”} \\
 & & \ \land\ i' = i + 1 \\
 & & \ \land\ pc' = \text{“}done\text{”}
\end{array}
$$

These are equivalent definitions of $Next$.

Is completely equivalent to our original definition.

It doesn’t matter which one we use.

This C code example is tiny. Most of the examples I will present are simple.

I believe you'll learn more by carefully studying simple examples than by skimming complex ones.

For now, you'll have to trust me — and the engineers at Amazon Web Services and elsewhere who use it — when we say that TLA$^+$ is good for specifying real systems, not just toy examples.

End of Lecture 2

STATE MACHINES IN TLA$^+$

This is the end of Lecture 2 of the TLA$^+$ Video Course

—

State Machines in Math