

P4-SpecTec Tutorial

A Project-Based Introduction with Nano-P4

Haechan Kwon

PLRG @ KAIST

September 17, 2026

Why mechanize language specifications?

Why mechanize language specifications?

Language specifications are usually in English prose.

Why mechanize language specifications?

Language specifications are usually in English prose.

- **Ambiguous**

- One sentence may mean multiple things.
- Corner cases may be under-specified.

Why mechanize language specifications?

Language specifications are usually in English prose.

- **Ambiguous**
 - One sentence may mean multiple things.
 - Corner cases may be under-specified.
- **Spec - Implementation Divergence**
 - The prose specification and reference implementation evolve at different paces.

Why mechanize language specifications?

Language specifications are usually in English prose.

- **Ambiguous**

- One sentence may mean multiple things.
- Corner cases may be under-specified.

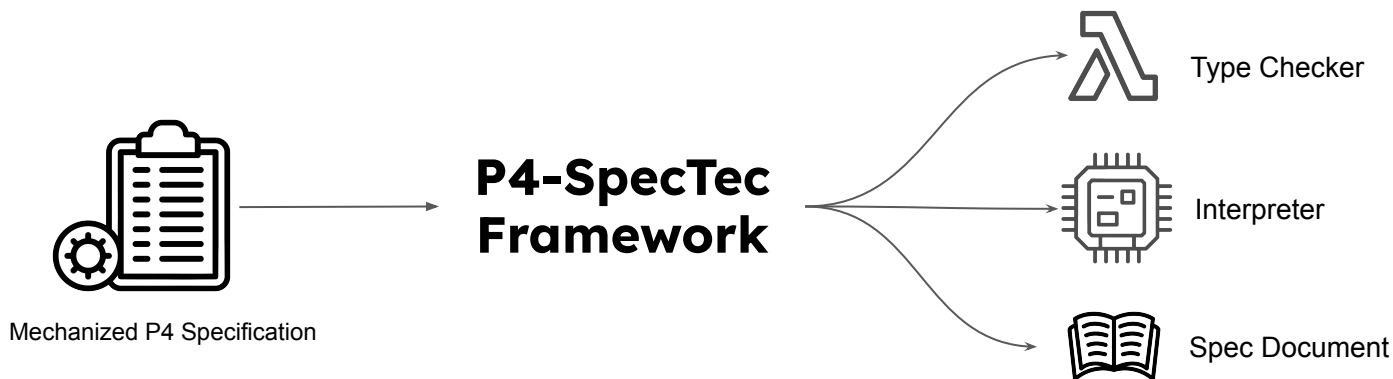
- **Spec - Implementation Divergence**

- The prose specification and reference implementation evolve at different paces.

Spec mechanization yields an **executable specification**, addressing the limitations of natural language specifications.

What is P4-SpecTec?

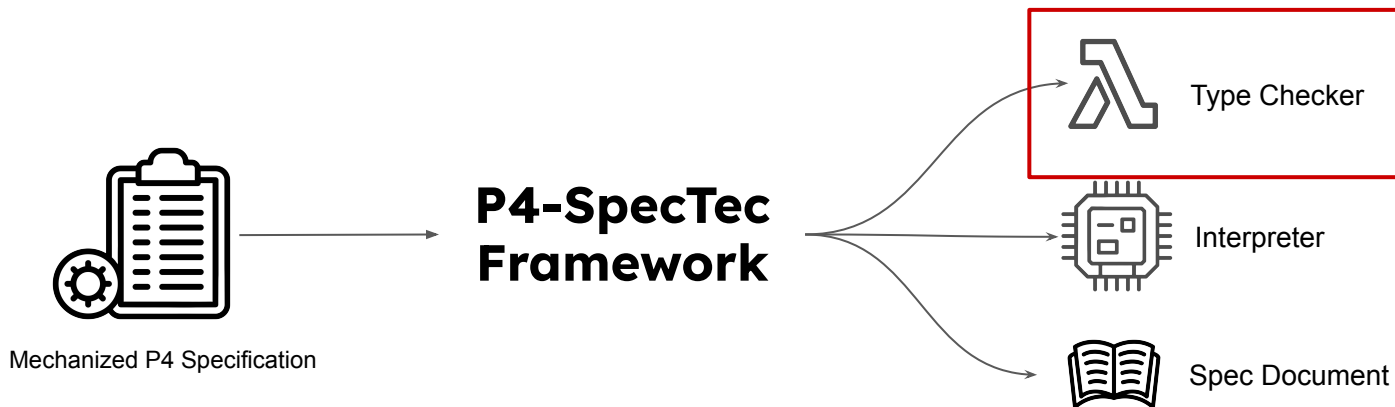
A framework for **mechanizing** the P4 language specification.



The mechanized spec provides a **single source of truth** for P4.

What is P4-SpecTec?

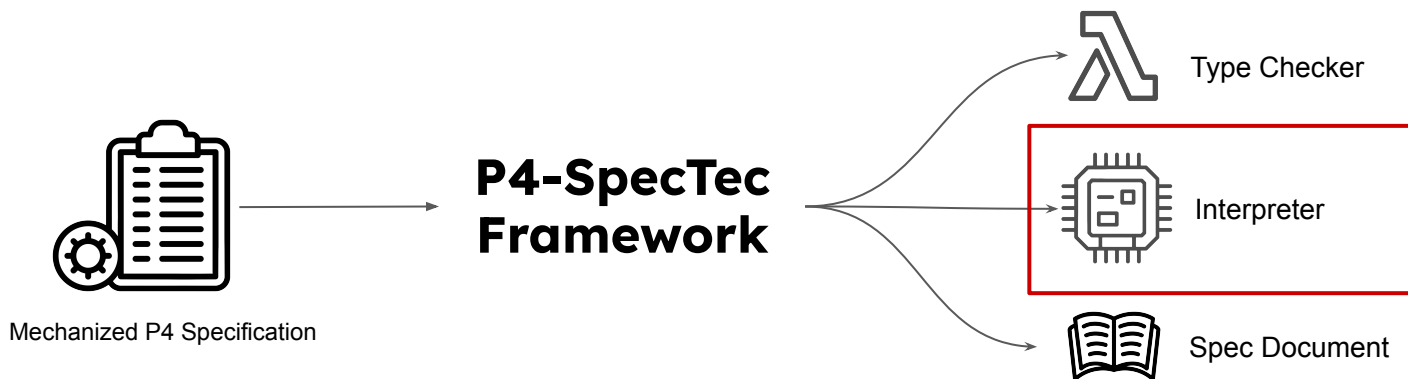
A framework for **mechanizing** the P4 language specification.



The mechanized spec provides a **single source of truth** for P4.

What is P4-SpecTec?

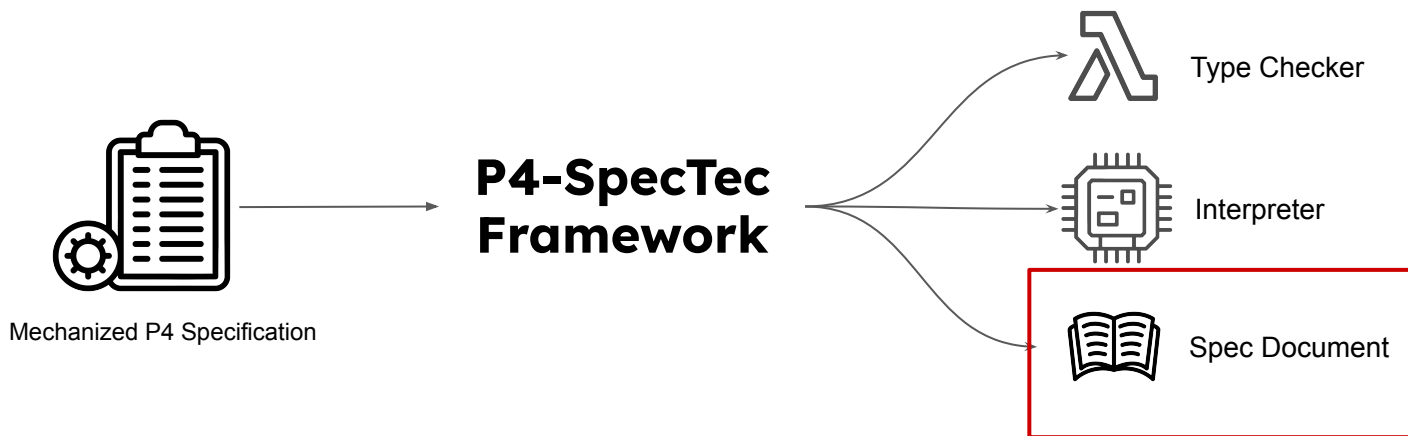
A framework for **mechanizing** the P4 language specification.



The mechanized spec provides a **single source of truth** for P4.

What is P4-SpecTec?

A framework for **mechanizing** the P4 language specification.



The mechanized spec provides a **single source of truth** for P4.

The P4-SpecTec DSL

With the DSL, we write *algorithmic inference rules*, which are executable.

```
Execution Order ↓ rule Program_ok:  
                    |- program -| TC'  
                    -- if declaration* = $flatten_program(program)  
                    -- if TC = $empty_typingContext  
                    -- Decls_ok: TC |- declaration* -| TC' ] premise(s)
```

The P4-SpecTec DSL

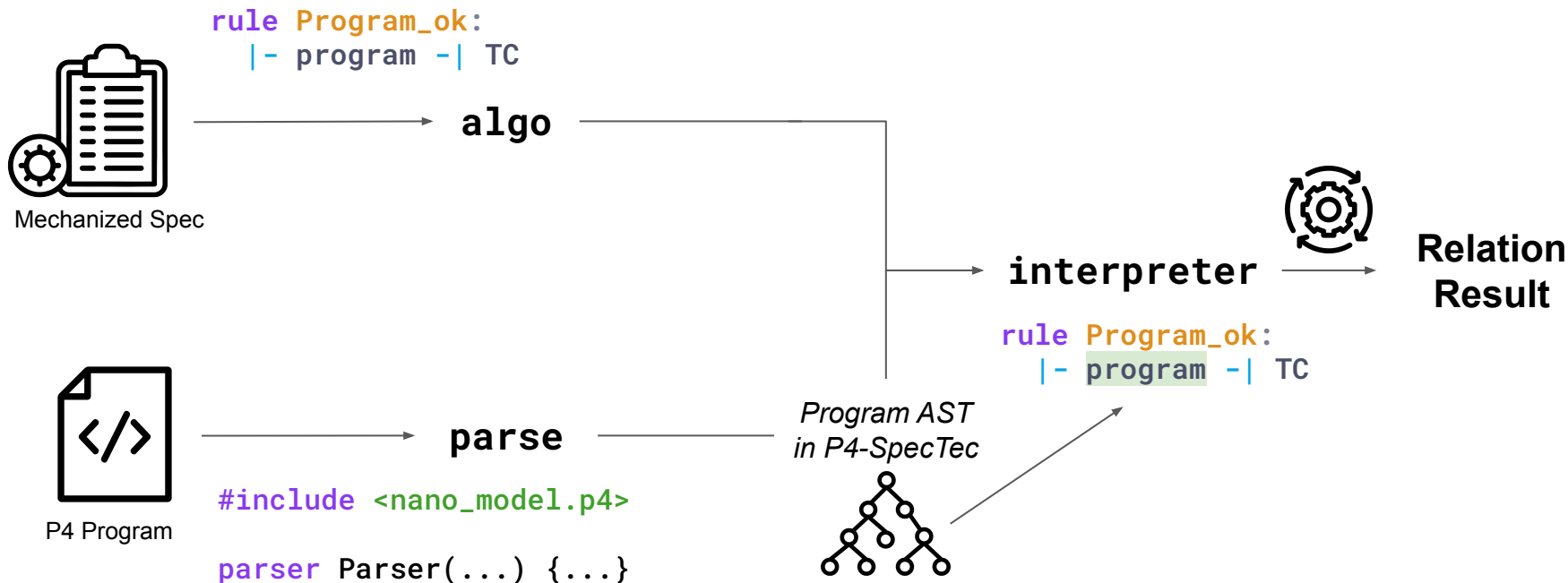
With the DSL, we write *algorithmic inference rules*, which are executable.

Execution Order ↓

```
rule Program_ok:
  |- program -| TC'
  -- if declaration* = $flatten_program(program)
  -- if TC = $empty_typingContext
  -- Decl_ok: TC |- declaration* -| TC' ] premise(s)
```

$$\frac{\text{flatten}(p) = \vec{d} \quad \emptyset \vdash_{\text{decl}} \vec{d} \dashv \text{TC}'}{\vdash_{\text{prog}} p \dashv \text{TC}'}$$

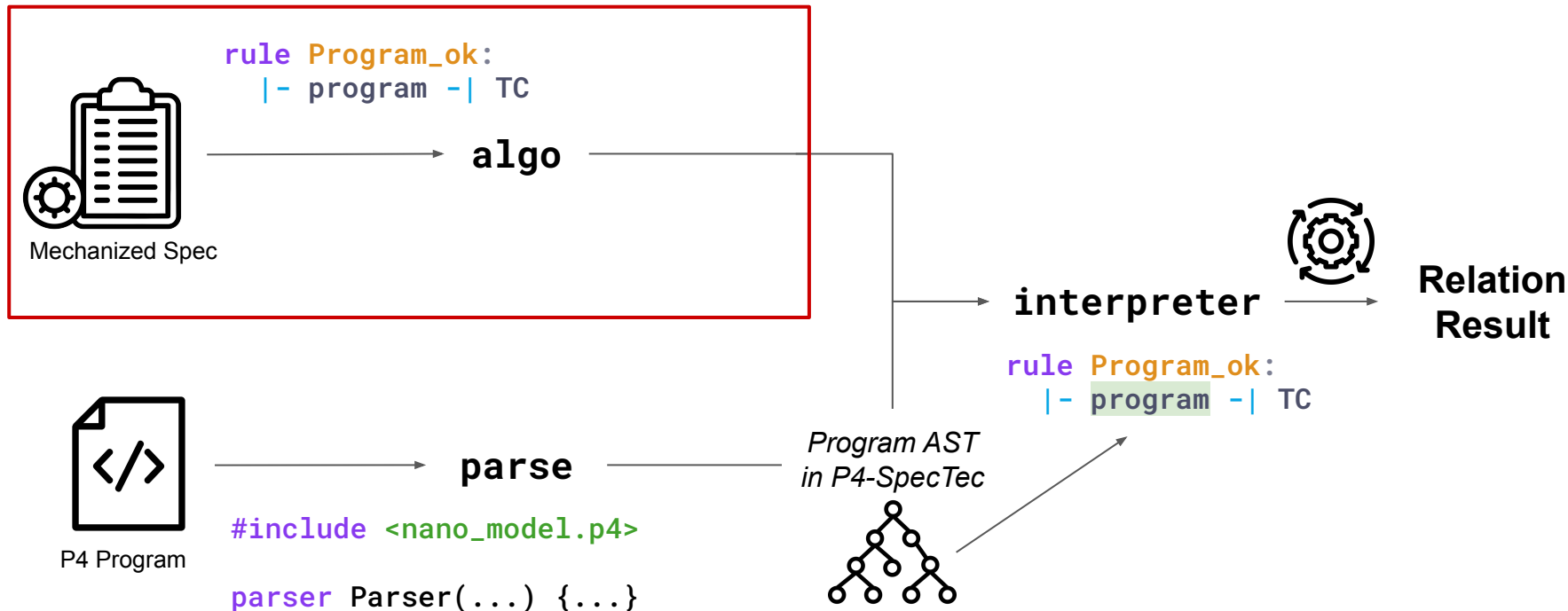
P4-SpecTec Toolchain Pipeline



algo: checks that every rule in EL is algorithmically executable

parse: parses and transforms the P4 source to a meta-value

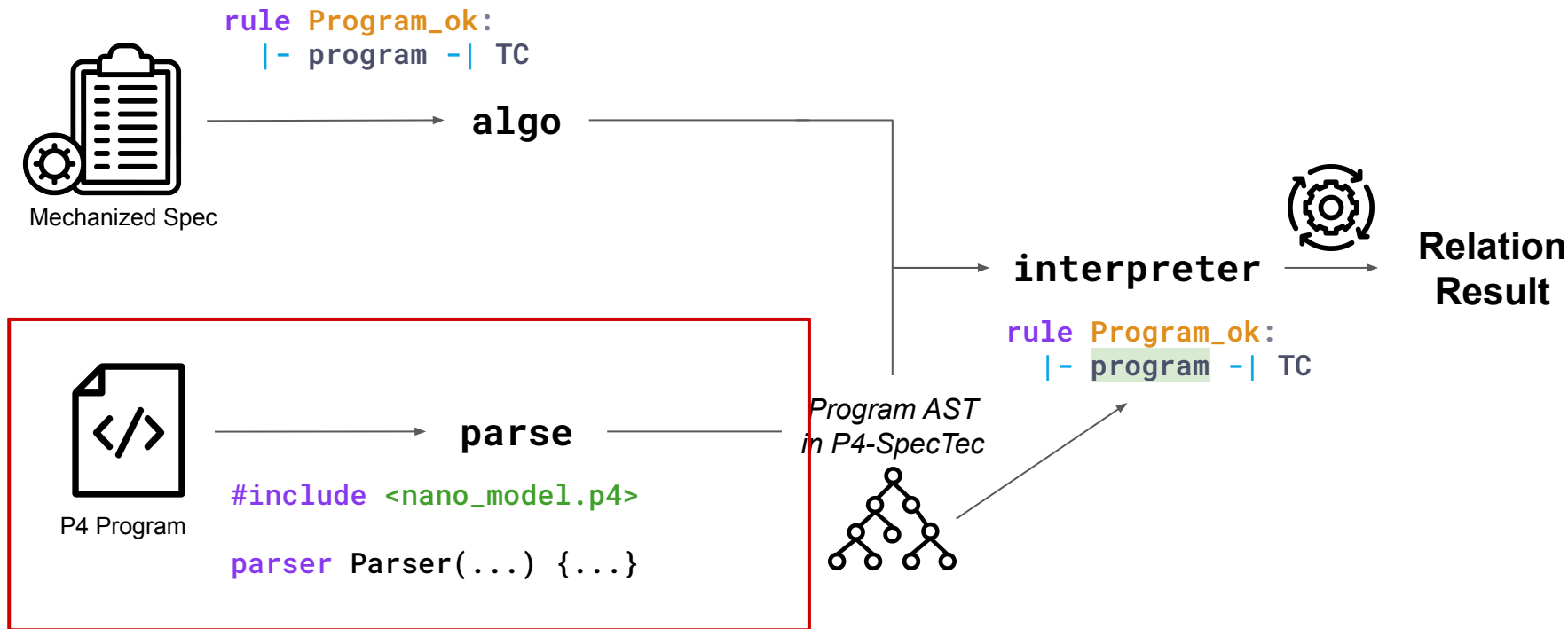
P4-SpecTec Toolchain Pipeline



algo: checks that every rule in EL is algorithmically executable

parse: parses and transforms the P4 source to a meta-value

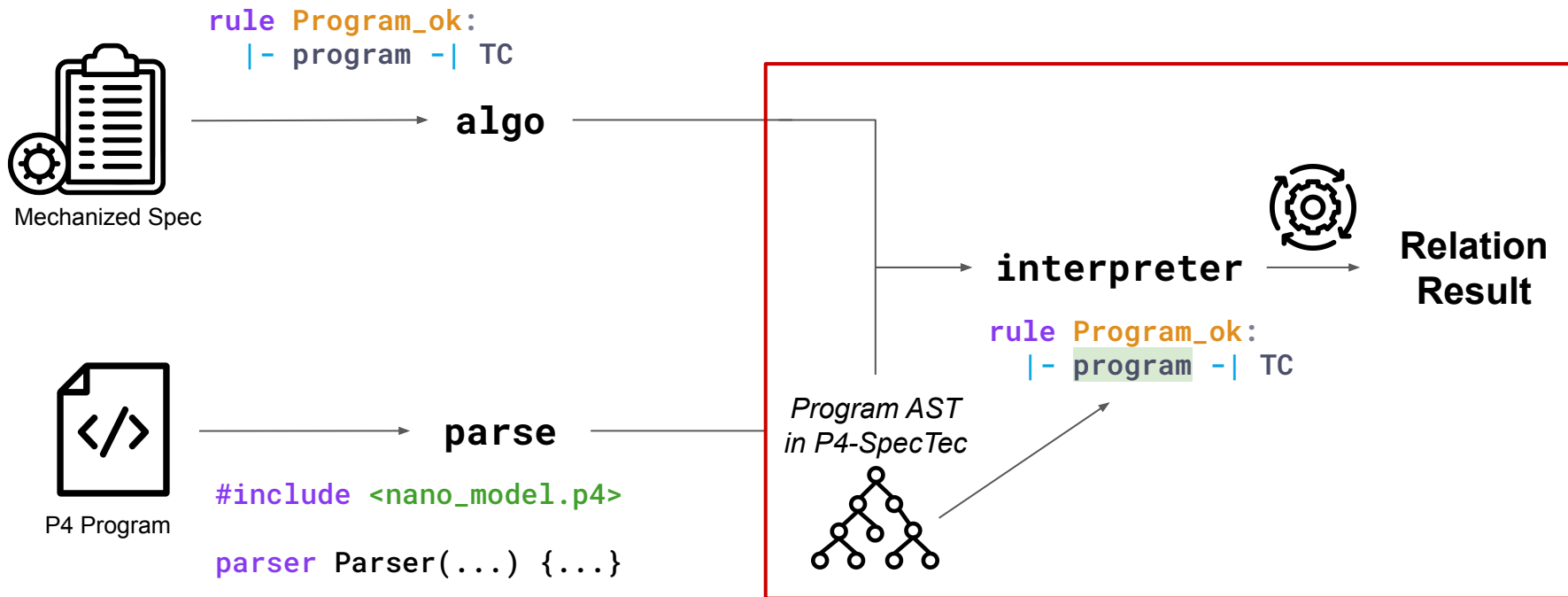
P4-SpecTec Toolchain Pipeline



algo: checks that every rule in EL is algorithmically executable

parse: parses and transforms the P4 source to a meta-value

P4-SpecTec Toolchain Pipeline



algo: checks that every rule in EL is algorithmically executable

parse: parses and transforms the P4 source to a meta-value

A Mechanized Specification for P4

Mechanized Spec

- Passes 95% of positive typecheck tests in p4c
- Passes 98% of V1Model tests in p4c
- Automatically generated prose document

Below is a prose algorithm that implements `Program_ok`. Details of what each meta-variable means are provided in the subsequent sections.

▼ Click to view the specification source

```
rule Program_ok:
  |- p4program : (declarationIR* ';')
  ==> if declaration* = $flatten_p4program(p4program)
  ==> if TC_0 = $empty_typingContext()
  ==> Decl_ok: TC_0 |- declaration* : TC_0 declarationIR*
```

▼ Click to view the mathematical definition

`Program_ok`

(1) `declaration*` = `flatten_p4program (p4program)`
(2) `TC_0` = `empty_typingContext ()`
(3) $\frac{TC_0 \vdash declaration^* : TC_0 declarationIR^*}{\vdash p4program : (declarationIR^* ;)}$

Typing p4program:

1. Let `declaration*` be the list of declarations in `p4program`. [FAIL]
2. Let `TC_0` be an empty typing context. [FAIL]
3. Let context `TC_1` and `declarationIR*` be the result of typing `declaration*` under context `TC_0`.
4. The result is typed P4 program `declarationIR*`. ';

Prose document generated from the mechanized P4
spec

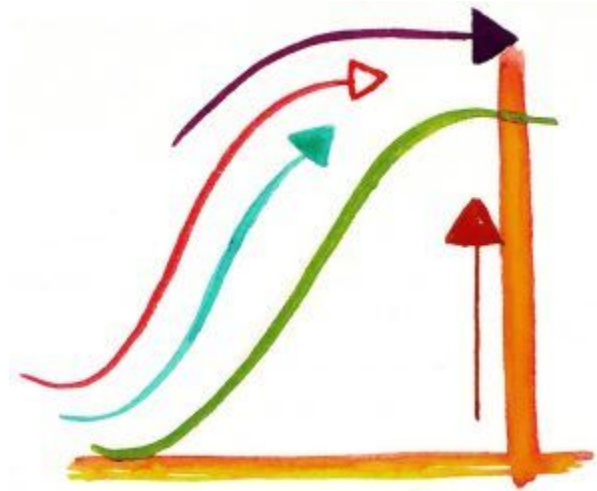
using P4-SpecTec's prose backend

Towards Adoption

P4-SpecTec is **conditionally adopted** by the P4 LDWG as the official language specification authoring toolchain for P4.

- The P4 Language Design Working Group has decided to officially conditionally accept P4-SpecTec as the official language specification authoring toolchain for P4.
 - We hope to transition this to an unconditional acceptance by the end of 2026, given that the timeline in slide 13 above is followed and the questions asked by the community below are handled satisfactorily.

P4 LDWG Notes - March 2nd, 2026



Learning resource is key to adoption!

About the Tutorial

Mechanizing the Nano-P4 specification with P4-SpecTec

Developed during 2026 Google Summer of Code with the P4 Language Consortium.

Proposes an end-to-end tutorial documentation, to lower the barrier into P4-SpecTec.

Mechanizing the Nano-P4 specification with P4-SpecTec

Overview

This guide walks you through spec mechanization in [P4-SpecTec](#), using its core constructs to describe the semantics of Nano-P4, an educational dialect of P4.

Audience and Scope

Audience. This tutorial is written for the P4 community. It focuses on the practical use of P4-SpecTec for specifying a P4-like language, so a basic familiarity with P4 is recommended. A background in programming-language theory is helpful, but no background in formal methods or proof assistants is required.

What is covered.

- Syntax definition for Nano-P4
- Type-checking and evaluation specifications for Nano-P4
- Prose specification generation
- Hands-on exercises at the end of each section

What is not covered.

- A complete walkthrough of P4-SpecTec. The basics are covered, but P4-SpecTec is treated as a tool throughout, not as the subject of study.
- The full P4 language. Nano-P4 is a deliberately small educational subset, and the tutorial stays within that boundary throughout.

By the end, you will have a full mechanized specification, a reference type checker and interpreter, and natural-language documentation for Nano-P4. Most importantly, you will gain hands-on experience writing a mechanized specification in P4-SpecTec, a skill transferable to any project using the toolchain, including the full P4 language mechanization.

Goals For This Talk

Demystify P4-SpecTec and help you get started today:

- Understand the basics of the P4-SpecTec DSL
- Understand Nano-P4 and its purpose
- Get familiar with the P4-SpecTec CLI

P4-SpecTec DSL: Basics

Built-in Data Types

bool:	Boolean	true, false
int:	Arbitrary-precision integer	0, 1, -1, ...
nat:	Non-negative integer	0, 1, 2, ...
text:	String	"package", ""
T*:	List	eps, [42, 7], 1 :: [2], ...
T?:	Option	eps, <some value of type T>

Basics

A SpecTec program is a list of:

- Syntax `syntax parenthesizedExpression = `(expression `)`
- Function `dec $ite<X>(bool, X, X) : X`
- Relation `relation Program_ok:
 |- program -| typingContext`
- Rule `rule Program_ok:
 |- program -| TC'`
- ... `-- if declaration* = $flatten_program(program)
 ...`

Variant Definition via syntax

Defines a type.

Commonly used to define grammar productions.

```
syntax controlDeclaration =  
    CONTROL name  
    ` ( parameterList ` )  
    `{ controlLocalDeclarationList APPLY controlBody ` }
```

```
control Filter(inout Header hdr, out bool pass) {  
    apply { pass = false; }  
}
```



Variant Definition via syntax

Defines a type.

Commonly used to define grammar productions.

```
syntax controlDeclaration =  
    CONTROL name  
    ` ( parameterList ` )  
    ` { controlLocalDeclarationList APPLY controlBody ` }
```

Terminal Symbols

Nonterminal Symbols

```
control Filter ( inout Header hdr, out bool pass ) {  
    apply { pass = false; }  
}
```



Variant Definition via syntax

```
syntax declaration =  
  | instantiation  
  | actionDeclaration  
  | matchKindDeclaration  
  | externDeclaration  
  | parserDeclaration  
  | controlDeclaration  
  | typeDeclaration
```

A declaration is one of instantiation, actionDeclaration, ..., or typeDeclaration.

Variant Definition via syntax

```
syntax declaration =  
  | instantiation  
  | actionDeclaration  
  | matchKindDeclaration  
  | externDeclaration  
  | parserDeclaration  
  | controlDeclaration  
  | typeDeclaration
```

A declaration is one of instantiation, actionDeclaration, ..., or typeDeclaration.

```
declaration  
  : constantDeclaration  
  | externDeclaration  
  | actionDeclaration  
  | parserDeclaration  
  | typeDeclaration  
  | controlDeclaration  
  | instantiation  
  | errorDeclaration  
  | matchKindDeclaration  
  | functionDeclaration  
  ;
```

P4₁₆ Language Specification – Declaration Syntax

Functions

Declared with `dec` and defined with `def`

A function definition may have multiple *clauses*.

```
dec $fibonacci(int) : int
def $fibonacci(0) = 1
def $fibonacci(1) = 1
def $fibonacci(i) = $( $fibonacci($(i - 1)) + $fibonacci($(i - 2)))

dec $ite<X>(bool, X, X) : X
def $ite<X>(true, X_t, X_f) = X_t
def $ite<X>(false, X_t, X_f) = X_f
```

Relations & Rules

A relation defines the signature for a set of rules.

An input hint specifies which positional components are inputs to the relation.

```
relation Program_ok:  
  |- program -| typingContext  
  hint(input %0)
```

Type-checking program yields typing context
typingContext

```
relation Expr_ok:  
  scope typingContext |- expression : typeIR  
  hint(input %0 %1 %2)
```

In scope and under typing context typingContext,
expression is type-checked as typeIR

Relations & Rules

A rule defines *when* a relation holds.

Each rule has a *judgment* and zero or more *premises*.

```
syntax expr =  
  | id           ;; reference expression  
  | unop expr    ;; unary expression  
  | expr binop expr ;; binary expression
```

```
relation Expr_eval:  
  scope evalContext |- expr : value  
  hint(input %0 %1 %2)
```

```
rule Expr_eval/referenceExpression:  
  scope EC |- id : value  
  ;; ... premises ...
```

```
rule Expr_eval/unaryExpression:  
  scope EC |- unop expr : value'  
  ;; ... premises ...
```

```
rule Expr_eval/binaryExpression:  
  scope EC |- expr_l binop expr_r : value  
  ;; ... premises ...
```

Premises

A premise defines the conditions required for a rule to hold.

- Variable binding `-- if nat = 3`
- Side condition `-- if type_a = type_b`
- Relation invocation `-- Expr_ok: scope TC |- expression : typeIR`
- ...(more)...

Premises – Binding

Introduced with an `if` premise

```
-- if int = -3  
-- if nat = 3
```

Premises – Binding

Introduced with an `if` premise

```
-- if int = -3  
-- if nat = 3
```

A variable's type is inferred from its name.

```
-- if nat' = 3           ;; Expected nat.  
-- if nat_3 = 3          ;; Expected nat.  
-- if nat_foo = "foo"    ;; ERROR! Expected nat.
```

Premises – Binding

Introduced with an `if` premise

A variable's type is inferred from its name.

```
-- if int = -3
-- if nat = 3
-- if nat' = 3
-- if nat_3 = 3
-- if nat_foo = "foo"
```

;; Expected nat.
;; Expected nat.
;; ERROR! Expected nat.

Pattern Match (Destructuring Binding)

If one side is *bound*, attempt pattern match

```
syntax unaryExpression = unop expression
```

```
-- if unop expression = unaryExpression ;; unaryExpression is bound
```

Premises – Binding

Introduced with an `if` premise

A variable's type is inferred from its name.

```
-- if int = -3
-- if nat = 3
-- if nat' = 3
-- if nat_3 = 3
-- if nat_foo = "foo"
```

;; Expected nat.
;; Expected nat.
;; ERROR! Expected nat.

Pattern Match (Destructuring Binding)

If one side is *bound*, attempt pattern match

```
syntax unaryExpression = unop expression
```

```
-- if unop expression = unaryExpression ;; unaryExpression is bound
```

```
syntax parenthesizedExpression = `( expression `)
```

```
-- if `( expression `) = parenthesizedExpression ;; parenthesizedExpression is bound
```

Premises – Side Condition

An `if` premise with a boolean expression adds a side condition.

The condition must evaluate to `true` for the rule to continue.

```
rule Expr_ok/integer:
  scope TC |- unop expression : integerTypeIR
  -- if unop <- [ '~', '-', '+' ]
  -- Expr_ok: scope TC |- expression : integerTypeIR
```

Premises – Relation Invocation

Uses a declared relation as a premise.

The judgment must *hold* for the rule to continue.

```
rule Expr_ok/parenthesizedExpression:  
  scope TC |- `( expression `) : typeIR  
  -- Expr_ok: scope TC |- expression : typeIR
```

In `scope` under typing context `TC`, ``(expression `)` type-checks to `typeIR` if:

1. In `scope` under typing context `TC`, `expression` type-checks to `typeIR`.

Premises – Relation Invocation

When a relation is invoked:

- Rules (part of that relation) are tried in declaration order
- A *failed* premise (failed pattern match, if premise ..) aborts the current rule
- If no rule matches, the relation *fails*

```
-- Expr_ok:  
  scope TC |- ( 3 '+' 5 ): typeIR
```

```
rule Expr_eval/referenceExpression:  
  scope EC |- id : value  
  ;; ... premises ...
```

```
rule Expr_eval/unaryExpression:  
  scope EC |- unop expr : value'  
  ;; ... premises ...
```

```
rule Expr_eval/binaryExpression:  
  scope EC |- expr_l binop expr_r : value  
  ;; ... premises ...
```

Premises – Relation Invocation

When a relation is invoked:

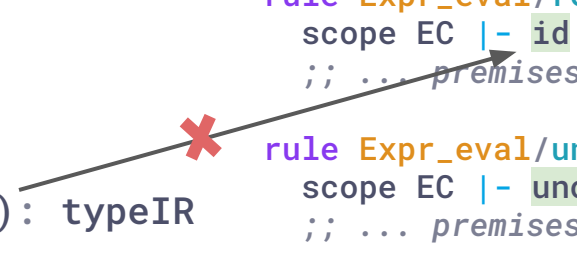
- Rules (part of that relation) are tried in declaration order
- A *failed* premise (failed pattern match, if premise ..) aborts the current rule
- If no rule matches, the relation *fails*

```
-- Expr_ok:
  scope TC |- ( 3 '+' 5 ): typeIR

rule Expr_eval/referenceExpression:
  scope EC |- id : value
  ;; ... premises ...

rule Expr_eval/unaryExpression:
  scope EC |- unop expr : value'
  ;; ... premises ...

rule Expr_eval/binaryExpression:
  scope EC |- expr_l binop expr_r : value
  ;; ... premises ...
```



Premises – Relation Invocation

When a relation is invoked:

- Rules (part of that relation) are tried in declaration order
- A *failed* premise (failed pattern match, if premise ..) aborts the current rule
- If no rule matches, the relation *fails*

```
-- Expr_ok:
  scope TC |- ( 3 '+' 5 ): typeIR

rule Expr_eval/referenceExpression:
  scope EC |- id : value
  ;; ... premises ...

rule Expr_eval/unaryExpression:
  scope EC |- unop expr : value'
  ;; ... premises ...

rule Expr_eval/binaryExpression:
  scope EC |- expr_l binop expr_r : value
  ;; ... premises ...
```

Premises – Relation Invocation

When a relation is invoked:

- Rules (part of that relation) are tried in declaration order
- A *failed* premise (failed pattern match, if premise ..) aborts the current rule
- If no rule matches, the relation *fails*

```
-- Expr_ok:  
  scope TC |- ( 3 '+' 5 ): typeIR
```

```
rule Expr_eval/referenceExpression:  
  scope EC |- id : value  
  ;; ... premises ...
```

```
rule Expr_eval/unaryExpression:  
  scope EC |- unop expr : value'  
  ;; ... premises ...
```

```
rule Expr_eval/binaryExpression:  
  scope EC |- expr_l binop expr_r : value  
  ;; ... premises ...
```



Nano-P4

An Educational Dialect of P4

Nano-P4: An educational dialect of P4

The tutorial “Mechanizing the Nano-P4 specification with P4-SpecTec” uses P4-SpecTec to formalize *Nano-P4*, an educational dialect of P4.

The only architecture available to Nano-P4 is NanoSwitch, a simplified version of the eBPF architecture.

Nano-P4 & P4

Nano-P4 serves as a stepping stone to understanding P4.

- Simplifies many of P4's complex language features (e.g. generics, type inference)
- Retains the core structure of P4 (e.g. parser, control, table, action)

```
relation Program_ok:
  |- p4program : p4programIR
  hint(input %0)

rule Program_ok:
  |- p4program : (declarationIR* ';' )
  -- if declaration* = $flatten_p4program(p4program)
  -- if TC_0 = $empty_typingContext
  -- Decls_ok: TC_0 |- declaration* : TC_1 declarationIR*
```

Program_ok in the [P4 spec](#)

```
relation Program_ok:
  |- program -| typingContext
  hint(input %0)

rule Program_ok:
  |- program -| TC'
  -- if declaration* = $flatten_program(program)
  -- if TC = $empty_typingContext
  -- Decls_ok: TC |- declaration* -| TC'
```

Program_ok in the [Nano-P4 spec](#)

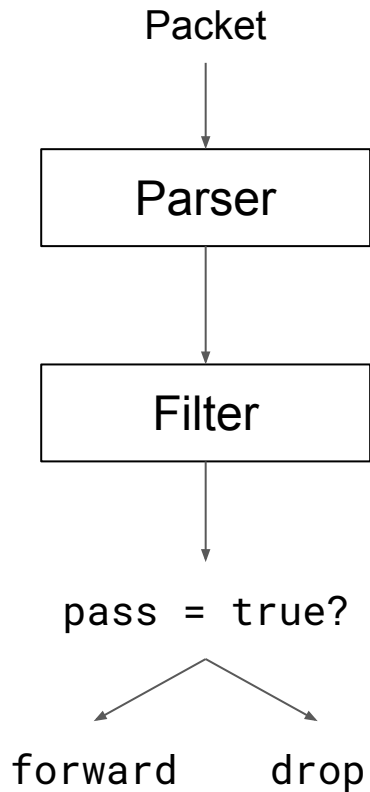
Example Nano-P4 Program

```
#include <nano_model.p4>

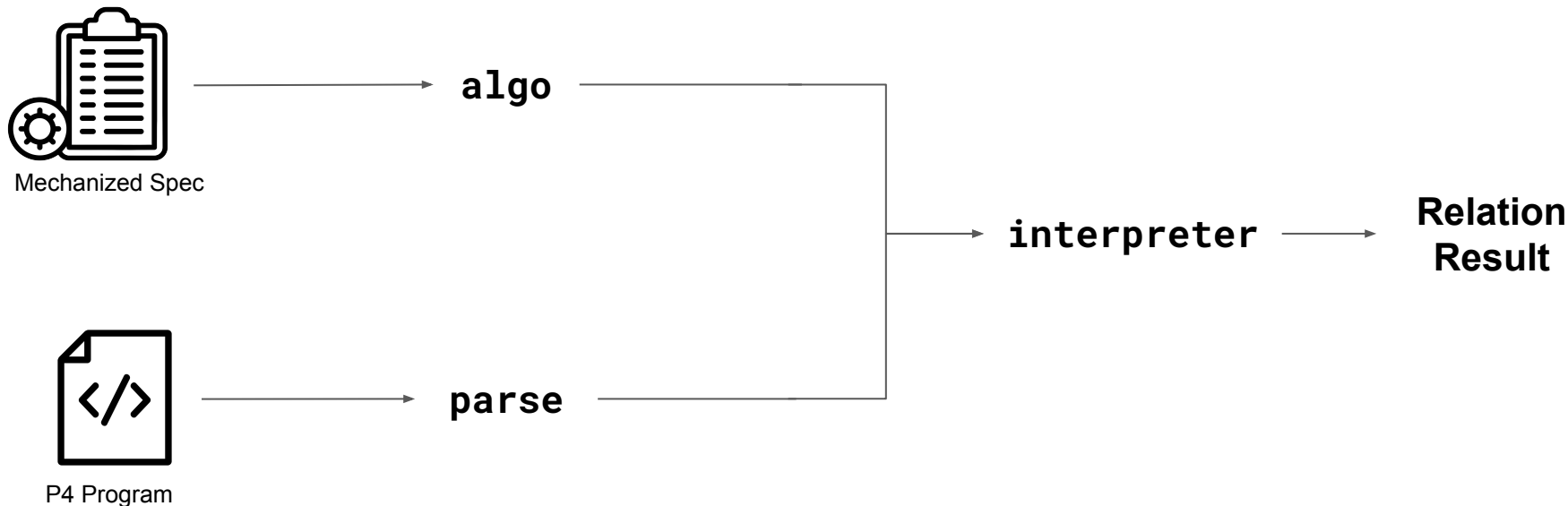
parser Parser(packet_in pkt, out Header hdr) {
    state start {
        transition accept;
    }
}

control Filter(inout Header hdr, out bool pass) {
    apply {
        pass = true;
    }
}

NanoSwitch(Parser(), Filter()) main;
```



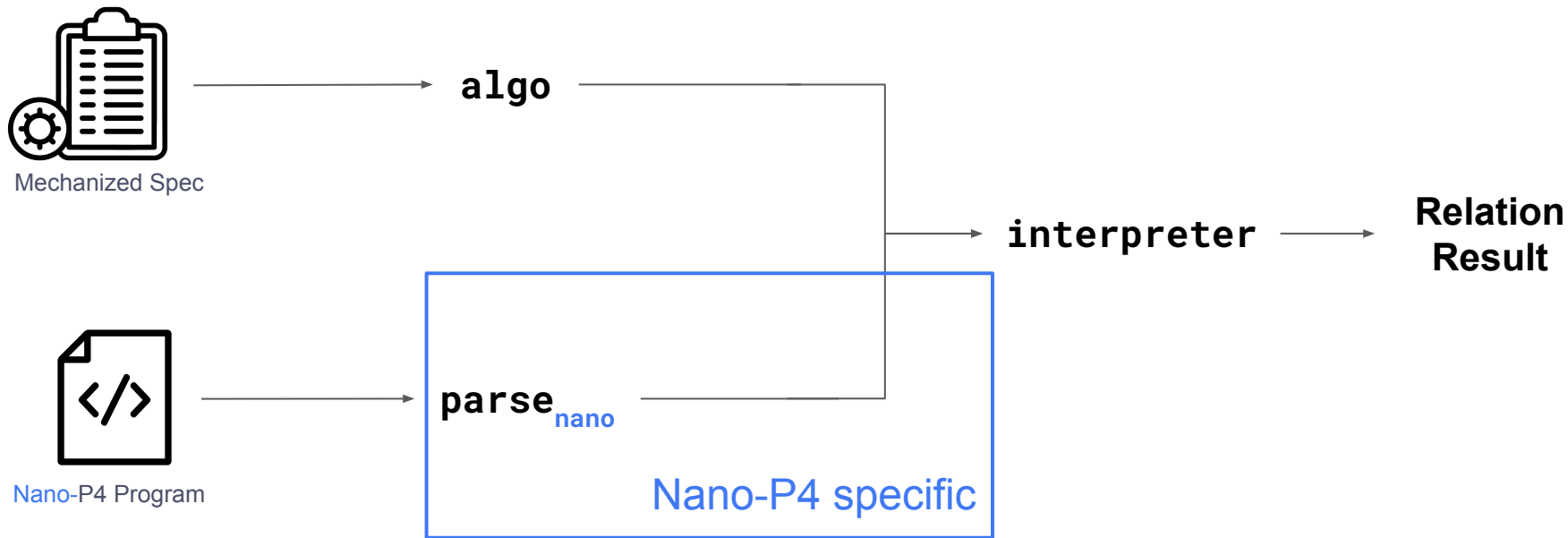
P4-SpecTec Toolchain Pipeline



algo: checks that every rule in EL is algorithmically executable

parse: parses and transforms the P4 source to a meta-value

Nano-P4-SpecTec Toolchain Pipeline



algo: checks that every rule in EL is algorithmically executable

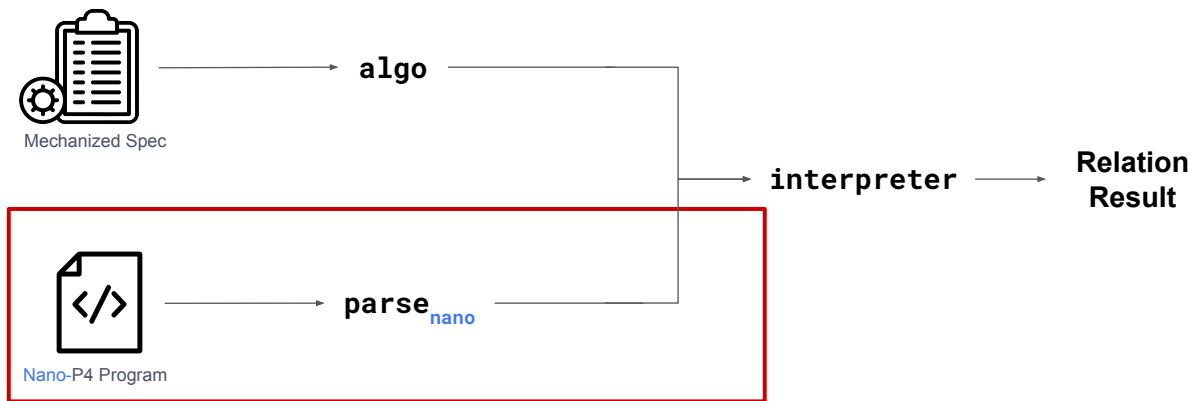
parse_{nano}: parses and transforms the P4 source to a meta-value

Hands-on

The Nano-P4 spec can be found under nano-p4/spec in [kaist-plrg/p4-spectec](https://github.com/kaist-plrg/p4-spectec)
(gsoc-nano-spec branch).

P4-SpecTec CLI: parse

```
nano-p4spectec parse -p <nano-p4 program> -i <p4include>
```

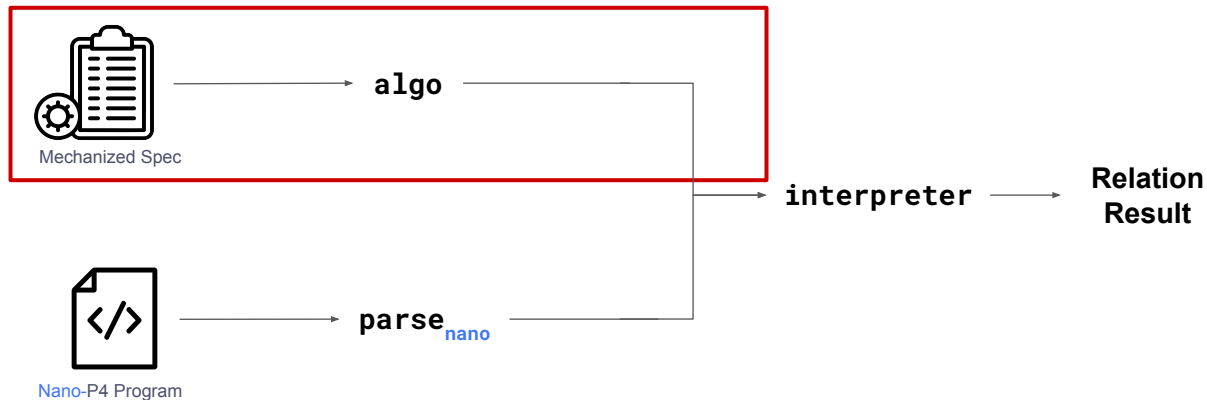


algo: checks that every rule in the spec is algorithmically executable

parse_{nano}: parses and transforms the P4 source to a meta-value

P4-SpecTec CLI: algo

`nano-p4spectec algo <spec-directory>`

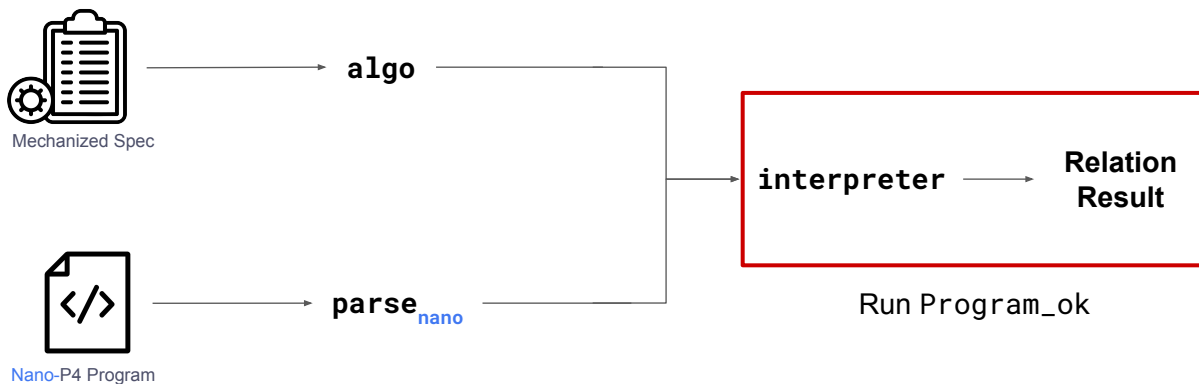


algo: checks that every rule in the spec is algorithmically executable

parse_{nano}: parses and transforms the P4 source to a meta-value

P4-SpecTec CLI: check

```
nano-p4spectec check <spec-directory>  
  -i <include-directory>  
  -p <nano-p4 program>
```

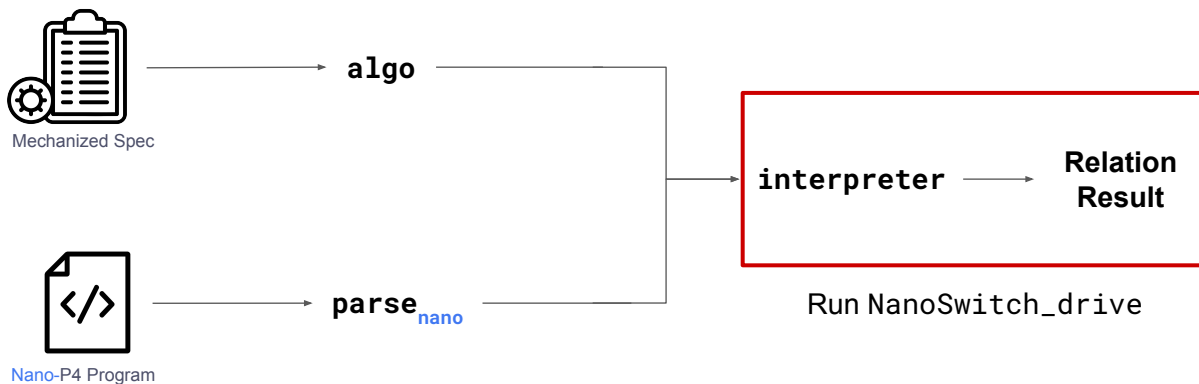


algo: checks that every rule in the spec is algorithmically executable

parse_{nano}: parses and transforms the P4 source to a meta-value

P4-SpecTec CLI: eval

```
nano-p4spectec eval <spec-directory> -i <include-directory>  
-p <nano-p4 program> -stf <packet input>
```



algo: checks that every rule in the spec is algorithmically executable

parse_{nano}: parses and transforms the P4 source to a meta-value

Demo: Typing Expressions in Nano-P4

Demo: Typing Expressions in Nano-P4

```
#include <nano_model.p4>

parser Parser(packet_in pkt, out Header hdr) {
    state start {
        transition accept;
    }
}

control Filter(inout Header hdr, out bool pass) {
    apply {
        bit<8> a = 8w1;
        bit<8> b = (8w2);
        bit<8> c = -a;
        bit<8> d = a + b;

        pass = false;
    }
}

NanoSwitch(Parser(), Filter()) main;
```

[expression-demo.p4](#)

Demo: Generating Prose Specification

Mechanizing the Nano-P4 specification with P4-SpecTec

Generating Prose Specification

The chapters up to this point have walked through every part of the Nano-P4 specification as it lives in `.watsup` files. P4-SpecTec can do more than run programs against a spec: it can also *render* the spec into human-readable English prose, formatted as AsciiDoc, ready to be compiled into HTML or PDF. As a concrete example of what the output looks like, the P4-SpecTec team publishes a [spliced HTML document](#) generated from the mechanized P4 spec.

This chapter introduces the `splice` workflow and shows it in action on the Nano-P4 conditional statement.

[Tutorial Page](#)

Conditional Statement

The conditional statement is Nano-P4's sole branching construct. Unlike C or Java, where any non-zero integer is truthy, P4 (and Nano-P4) require the condition to be a Boolean expression. Both branches are always required; there is no optional `else`.

```
conditionalStatement
: IF ( expression ) blockStatement ELSE blockStatement
;
```

Type Checking

▼ Click to view the specification source

```
rule Statement_ok/conditionalStatement:
  scope TC
  |- IF `( expression `) blockStatement_then ELSE blockStatement_else -| TC
-- Expr_ok: scope TC |- expression : BOOL
-- Block_ok: TC |- blockStatement_then
-- Block_ok: TC |- blockStatement_else
```

▼ Click to view the mathematical definition

$$\frac{\begin{array}{l} \text{Statement_ok-conditionalStatement} \\ (1) \text{ scope TC} \vdash \text{expression} : \text{BOOL} \\ (2) \text{ TC} \vdash \text{blockStatement}_{\text{then}} \\ (3) \text{ TC} \vdash \text{blockStatement}_{\text{else}} \end{array}}{\text{scope TC} \vdash \text{IF (expression) blockStatement}_{\text{then}} \text{ ELSE blockStatement}_{\text{else}} \rightarrow \text{TC}}$$

Typing `IF `(expression `) blockStatementthen ELSE blockStatementelse`, under context `TC` at scope :

1. Let `typeIR` be the result of typing `expression`, under context `TC` at scope.
2. Let `baseTypeIR` be `typeIR` _[FAIL].
3. Check that `baseTypeIR` is `BOOL`.
4. If `blockStatementthen` is well-typed under context `TC` _[FAIL]
 - a. If `blockStatementelse` is well-typed under context `TC` _[FAIL] the result is context `TC`.

Conditional Statement

The conditional statement is Nano-P4's sole branching construct. Unlike C or Java, where any non-zero integer is truthy, P4 (and Nano-P4) require the condition to be a Boolean expression. Both branches are always required; there is no optional `else`.

```
conditionalStatement
: IF ( expression ) blockStatement ELSE blockStatement
;
```

Spec definitions,
inserted via *splicing*

Syntax Definition

Type Checking

▼ Click to view the specification source

```
rule Statement_ok/conditionalStatement:
  scope TC
  |- IF `( expression `) blockStatement_then ELSE blockStatement_else -| TC
-- Expr_ok: scope TC |- expression : BOOL
-- Block_ok: TC |- blockStatement_then
-- Block_ok: TC |- blockStatement_else
```

SpecTec Source

▼ Click to view the mathematical definition

$$\frac{\text{Statement_ok-conditionalStatement} \quad \begin{array}{l} (1) \quad \text{scope TC} \vdash \text{expression} : \text{BOOL} \\ (2) \quad \text{TC} \vdash \text{blockStatement}_{\text{then}} \\ (3) \quad \text{TC} \vdash \text{blockStatement}_{\text{else}} \end{array}}{\text{scope TC} \vdash \text{IF (expression) blockStatement}_{\text{then}} \text{ELSE blockStatement}_{\text{else}} \rightarrow \text{TC}}$$

Latex

Typing `IF `(expression `) blockStatementthen ELSE blockStatementelse`, under context `TC` at scope :

1. Let `typeIR` be the result of typing `expression`, under context `TC` at scope.
2. Let `baseTypeIR` be `typeIR` _[FAIL]
3. Check that `baseTypeIR` is `BOOL`.
4. If `blockStatementthen` is well-typed under context `TC` _[FAIL]
 - a. If `blockStatementelse` is well-typed under context `TC` _[FAIL] the result is context `TC`.

Prose

`[#sec-conditional-statement]`

`== Conditional Statement`

The conditional statement is Nano-P4's sole branching construct. Unlike C or Java, where any non-zero integer is truthy, P4 (and Nano-P4) require the condition to be a Boolean expression. Both branches are always required; there is no optional ``else``.

`${syntax: conditionalStatement}`

`=== Type Checking`

`${rulegroup-source: Statement_ok/conditionalStatement}`

`${rulegroup-latex: Statement_ok/conditionalStatement}`

`${rulegroup-prose: Statement_ok/conditionalStatement}`

`=== Runtime Evaluation`

`${rulegroup-source: Statement_eval/conditionalStatement}`

`${rulegroup-latex: Statement_eval/conditionalStatement}`

`${rulegroup-prose: Statement_eval/conditionalStatement}`

[conditional-statement.adoc](#)

```
[#sec-conditional-statement]
```

```
== Conditional Statement
```

The conditional statement is Nano-P4's sole branching construct. Unlike C or Java, where any non-zero integer is truthy, P4 (and Nano-P4) require the condition to be a Boolean expression. Both branches are always required; there is no optional ``else``.

```
${syntax: conditionalStatement}
```

```
=== Type Checking
```

```
${rulegroup-source: Statement_ok/conditionalStatement}
```

```
${rulegroup-latex: Statement_ok/conditionalStatement}
```

```
${rulegroup-prose: Statement_ok/conditionalStatement}
```

```
=== Runtime Evaluation
```

```
${rulegroup-source: Statement_eval/conditionalStatement}
```

```
${rulegroup-latex: Statement_eval/conditionalStatement}
```

```
${rulegroup-prose: Statement_eval/conditionalStatement}
```

[conditional-statement.adoc](#)

```
[#sec-conditional-statement]
== Conditional Statement
```

The conditional statement is Nano-P4's sole branching construct. Unlike C or Java, where any non-zero integer is truthy, P4 (and Nano-P4) require the condition to be a Boolean expression. Both branches are always required; there is no optional `else`.

```
$(syntax: conditionalStatement)
```

```
=== Type Checking
```

```
$(rulegroup-source: Statement_ok/conditionalStatement)
$(rulegroup-latex: Statement_ok/conditionalStatement)
$(rulegroup-prose: Statement_ok/conditionalStatement)
```

```
=== Runtime Evaluation
```

```
$(rulegroup-source: Statement_eval/conditionalStatement)
$(rulegroup-latex: Statement_eval/conditionalStatement)
$(rulegroup-prose: Statement_eval/conditionalStatement)
```

```
[#sec-conditional-statement]
== Conditional Statement
```

The conditional statement is Nano-P4's sole branching construct. Unlike C or Java, where any non-zero integer is truthy, P4 (and Nano-P4) require the condition to be a Boolean expression. Both branches are always required; there is no optional `else`.

```
++++
<span id="conditionalStatement"></span>
++++
```

```
[source,bison]
```

```
conditionalStatement
: IF ( expression ) blockStatement ELSE blockStatement
;
----
```

```
=== Type Checking
```

```
ifdef::backend-html5[]
```

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta name="generator" content="AsciiDoctor 2.0.26">
<title>Conditional Statement</title>
<link rel="stylesheet" href="">
<style>
/* AsciiDoctor default stylesheet | MIT License | https://asciidoctor.org */
/* Uncomment the following line when using as a custom stylesheet */
html{font-family:sans-serif;-webkit-text-size-adjust:100%}
a{background:none}
a:focus{outline:thin dotted}
a:active,a:hover{outline:0}
h1{font-size:2em;margin:.67em 0}
b,strong{font-weight:bold}
abbr{font-size:.9em}
abbr[title]{cursor:help;border-bottom:1px dotted #ddd;text-decoration:none}
```

conditional-statement.adoc

conditional-statement-spliced.adoc

conditional-statement-spliced.html

nano-p4spectec splice

asciidoctor

```

[#sec-conditional-statement]
== Conditional Statement

```

The conditional statement is Nano-P4's sole branching construct. Unlike C or Java, where any non-zero integer is truthy, P4 (and Nano-P4) require the condition to be a Boolean expression. Both branches are always required; there is no optional `else`.

```

$(syntax: conditionalStatement)

```

=== Type Checking

```

$(rulegroup-source: Statement_ok/conditionalStatement)
$(rulegroup-latex: Statement_ok/conditionalStatement)
$(rulegroup-prose: Statement_ok/conditionalStatement)

```

=== Runtime Evaluation

```

$(rulegroup-source: Statement_eval/conditionalStatement)
$(rulegroup-latex: Statement_eval/conditionalStatement)
$(rulegroup-prose: Statement_eval/conditionalStatement)

```

```

[#sec-conditional-statement]
== Conditional Statement

```

The conditional statement is Nano-P4's sole branching construct. Unlike C or Java, where any non-zero integer is truthy, P4 (and Nano-P4) require the condition to be a Boolean expression. Both branches are always required; there is no optional `else`.

```

++++
<span id="conditionalStatement"></span>
++++
[source,bison]
----
conditionalStatement
: IF ( expression ) blockStatement ELSE blockStatement
;
----

```

=== Type Checking

```

ifdef::backend-html5[]

```

```

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta name="generator" content="AsciiDoctor 2.0.26">
<title>Conditional Statement</title>
<link rel="stylesheet" href="">
<style>
/*! AsciiDoctor default stylesheet | MIT License | https://asciidoctor.org */
/* Uncomment the following line when using as a custom stylesheet */
html{font-family:sans-serif;-webkit-text-size-adjust:100%}
a{background:none}
a:focus{outline:thin dotted}
a:active,a:hover{outline:0}
h1{font-size:2em;margin:.67em 0}
b,strong{font-weight:bold}
abbr{font-size:.9em}
abbr[title]{cursor:help;border-bottom:1px dotted #ddd;text-decoration:none}

```

conditional-statement.adoc

conditional-statement-spliced.adoc

conditional-statement-spliced.html

nano-p4spectec splice

asciidoctor

Below is a prose algorithm that implements `Program_ok`. Details of what each meta-variable means are provided in the subsequent sections.

▼ Click to view the specification source

```

rule Program_ok:
  |- p4program : (declarationIR* ' ');
  -- if declaration* = $flatten_p4program(p4program)
  -- if TC_0 = Empty_typingContext()
  -- Declr_ok: TC_0 |- declaration* : TC_1 declarationIR*

```

▼ Click to view the mathematical definition

```

[Program_ok]
(1) declaration* = flatten_p4program(p4program)
(2) TC_0 = empty_typingContext()
(3) TC_0 |- declaration* : TC_1 declarationIR*
-----
  |- p4program : (declarationIR* ' ');

```

Typing `p4program`:

1. Let `declaration*` be the list of declarations in `p4program`. [FAIL]
2. Let `TC_0` be an empty typing context. [FAIL]
3. Let context `TC_1` and `declarationIR*` be the result of typing `declaration*` under context `TC_0`.
4. The result is typed P4 program `declarationIR* ' ';`.

Auto-generated spec is [live!](#)

Goals For This Talk

Demystify P4-SpecTec and help you get started today:

- Understand the basics of the P4-SpecTec DSL
- Understand Nano-P4 and its purpose
- Get familiar with the P4-SpecTec CLI

Goals Going Forward

- Complete the [tutorial](#)
 - 7 Chapters + 16 Exercises
- Read the [mechanized P4 specification](#)
- Check out the auto-generated [specification document](#)

Inquiries about P4-SpecTec: 99jaehyunlee@kaist.ac.kr

P4-SpecTec Issue Tracker: [Issues - kaist-plrg/p4-spectec](https://issues-kaist-plrg.github.io/p4-spectec/)