Percent as a First-Class Type with Explicit Operator Semantics

DANIEL DONCHE JR, Independent Researcher, USA

The percent symbol (%) is widely taught as "divide by 100," yet calculators, spreadsheets, and programming languages implement it in mutually incompatible ways. Consumer calculators and spreadsheets interpret % context-dependently (e.g., 8+25% → 10 but 8×25% → 2), while most programming languages reserve % for modulus and force explicit division by 100. We present a design used in the Goblin programming language that reconciles these worlds by: (1) introducing *percent* as a first-class, dimensionless type with literal and cast forms; (2) defining an explicit family of percent operators that eliminates context magic while preserving familiar use-cases; and (3) retaining standard modulus via whitespace disambiguation. Our design follows long-standing guidance in programming language semantics regarding operator well-definedness and context independence [Pierce 2002; Harper 2016] and aligns with findings on error patterns in spreadsheet percentage usage [Panko 2008; Hermans et al. 2012]. We formalize the semantics, give worked examples for auditability, compare behavior across systems, and prove a simple well-definedness property.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Theory of computation → Semantics and reasoning**.

Additional Key Words and Phrases: operator semantics, percent operator, formal definition, programming languages, calculators, spreadsheets, Goblin

ACM Reference Format:
Daniel Donche Jr. 2025. Percent as a First-Class Type with Explicit Operator Semantics. , (August 2025), 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In everyday arithmetic, % is colloquially understood as "divide by 100." In practice, computational systems disagree on its meaning. Consumer calculators and spreadsheets use context-dependent rules for % (e.g., $a+p\%$ adds $p\%$ of a, while $a\times p\%$ multiplies by $p/100$), creating hidden semantics that many users learn only by rote. By contrast, most programming languages avoid the ambiguity by reserving % for modulus and requiring explicit division by 100 for percentage math.

This paper presents a semantics used in the Goblin programming language that makes percent both *convenient* and *sound*: percent is a *first-class type* with clear, explicit operators. The design is intentionally conservative with respect to programming language principles of well-definedness and compositionality [Pierce 2002; Harper 2016], while offering calculator-style behavior on demand.

Contributions.

- We introduce **percent as a type**, with literal form `25%` and cast/constructor `pct(25)` (arguments in percentage points).
- We define an explicit operator family:

(1) **%** (postfix, no spaces): *percent of 1* (programmer-style default).
(2) **%s** (postfix): *percent of the left operand* (calculator-style).

Author's address: Daniel Donche Jr, Independent Researcher, Colorado Springs, CO, USA, dan@goblinlang.org.

ACM XXXX-XXXX/2025/8-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(3) `%` **of** E: *percent of explicit base* E.
(4) `%` (with spaces): standard *modulus*.

- We formalize parsing, desugaring, and precedence; prove a simple well-definedness property; and provide step-by-step worked examples for auditability.

2 BACKGROUND AND PROBLEM

Calculators and spreadsheets implement % with context-sensitive rules: $a{+}p\% \to a + (p/100) \cdot a$; $a{-}p\% \to a - (p/100) \cdot a$; $a{\times}p\% \to a \cdot (p/100)$; $a \div p\% \to a \div (p/100)$. The identical token $p\%$ thus has different denotations depending solely on the surrounding operator, which violates the spirit of operator well-definedness and compositionality emphasized in mainstream PL texts [Pierce 2002; Harper 2016]. Spreadsheet error literature further documents that percentages are a frequent source of user mistakes, often due to unclear bases and implicit context [Panko 2008; Hermans et al. 2012].

Programming languages typically dodge the issue by reserving % for modulus, forcing explicit division by 100 (e.g., `p/100`). While consistent, this is verbose and discourages natural expressions that users expect when working with percentages.

3 DESIGN OVERVIEW

3.1 Percent as a First-Class Type

Goblin introduces a dimensionless numeric type **percent**. It has:

- **Literal**: `25%` (coerces to 0.25 in numeric contexts).
- **Cast/constructor**: `pct(25)` (arguments in percentage points). Thus `pct(5)` ≡ `5%` and `pct(0.5)` ≡ `0.5%`.

Percent values coerce to their fractional numeric value when passed to math functions or mixed with numeric types:

$$\texttt{sin(50\%)} \equiv \sin(0.5), \qquad \texttt{sqrt(25\%)} \equiv \sqrt{0.25} = 0.5.$$

When combined with `of`, the result inherits the unit of the base (e.g., money).

3.2 Operator Family and Intuition

We make all interpretations explicit:

% (percent of 1). Postfix without spaces means "of 1": $p\% \rightsquigarrow p/100$. This matches programmer expectations: `8 * 25% = 2`, `100 / 25% = 400`, `8 + 25% = 8.25`.

%s (percent of self). Postfix `%s` means "of the left operand" (calculator-style):

$$A \circ p\%\texttt{s} \equiv A \circ ((p/100) \cdot A), \quad \circ \in \{+, -, \times, \div\}.$$

Examples: `8 + 25%s = 10`, `8 - 25%s = 6`, `8 * 25%s = 16`, `8 / 25%s = 4`.

% of E (explicit base). `p% of E` means $(p/100) \cdot E$, e.g., `8 + (25% of 50) = 20.5`.

% (with spaces): modulus. `8 % 3 = 2`.

3.3 Why This Mapping?

This design decouples concerns:

- **Type-level clarity:** percentages are values (`25%`, `pct(25)`), not mere notation.
- **Operator-level clarity:** the "percent of what?" question is always answered explicitly by `%s` or `of`, never implicitly by outer syntax.
- **Compatibility:** the default `%` keeps programmer-style semantics; `%s` recovers calculator-style addition/subtraction when desired; modulus is preserved via whitespace.

This adheres to established PL principles about operator meaning, context independence, and compositionality [Pierce 2002; Harper 2016].

4 FORMALIZATION

4.1 Grammar (EBNF Sketch)

```
expr         ::= term (("+" | "-") term)*
term         ::= factor (("*" | "/") factor)*
factor       ::= primary | percent_form | modulus
primary      ::= number | "(" expr ")" | pct_call

pct_call     ::= "pct" "(" expr ")"   # returns a percent
percent_form::= pct_lit ["of" expr]  # p% [of E] or p%s
pct_lit      ::= number "%" ["s"]     # 's' denotes self

modulus      ::= factor " % " factor  # spaces required around %
```

Tokenization notes. `25%s` is a single postfix token. `%` with spaces is modulus; `%` without spaces is a percent literal.

4.2 Desugaring (Operational)

```
desugar(E):
  case A     (p%s):           return A     ((p/100) * A)
  case p%:                  return (p / 100)
  case (p% of B):           return (p / 100) * desugar(B)
  case pct(X):              return (desugar(X) / 100)   # a percent value
  case (A % B) [spaced]:    return Mod(A, B)
  otherwise:                recurse
```

4.3 Precedence and Associativity

Highest: parentheses, function calls.
Then: primaries (`n%`, `n%s`, `pct(E)`, `n% of E`) as atomic numeric factors.
Then: unary `+`/`−`.
Then: `*`, `/`, and spaced modulus `%`.
Lowest: `+`, `−`.

```
x + 25%s * 2
= x + ((25% of x) * 2)
```

4.4 Well-Definedness

Definition 1 (Percent Base Mapping). *Let $p \in \mathbb{R}$ be percentage points. Define three base mappings:*

$$\text{of1}(p) = p/100, \quad \text{ofself}(p, a) = (p/100) \cdot a, \quad \text{of}(p, b) = (p/100) \cdot b.$$

Theorem 1 (Operator Well-Definedness). *For any $a, b \in \mathbb{R}$ and $p \in \mathbb{R}$, each of the forms p%, p%s, and p% of b desugars to a unique numeric value prior to participation in any outer arithmetic operator.*

Proof. By cases on the operator form using the desugaring rules above:

$$p\% \rightsquigarrow p/100; \quad p\%\mathtt{s} \rightsquigarrow (p/100) \cdot a; \quad p\% \ \mathtt{of} \ b \rightsquigarrow (p/100) \cdot b.$$

Each is a closed numeric term independent of the outer operator. Hence well-definedness holds. □

5 WORKED EXAMPLES (SHOW YOUR WORK)

We expand each example to eliminate ambiguity.

```
8 * 25%
= 8 * 0.25
= 2

100 / 25%
= 100 / 0.25
= 400

8 + 25%
= 8 + 0.25
= 8.25

8 - 25%
= 8 - 0.25
= 7.75



8 + 25%s
= 8 + (0.25 * 8)
= 8 + 2
= 10

8 - 25%s
= 8 - (0.25 * 8)
= 8 - 2
= 6

8 * 25%s
= 8 * (0.25 * 8)
= 8 * 2
= 16

8 / 25%s
= 8 / (0.25 * 8)
= 8 / 2
= 4
```

```
8 + (25% of 50)
= 8 + (0.25 * 50)
= 8 + 12.5
= 20.5


8 % 3
= remainder of 8    3
= 2


sqrt(25%)
= sqrt(0.25)
= 0.5

(25%)^2
= (0.25)^2
= 0.0625

sin(50%)
= sin(0.5)
```

6 COMPARATIVE ANALYSIS

6.1 "Percent of what?" for 8 + 25%

Table 1. Cross-system behavior for 8 + 25%.

System	Percent of what?	Result
Calculator/Sheets	8 (self)	10
Python/JS (no %)	n/a (must use 25/100)	8.25
Goblin %	1	8.25
Goblin %s	8 (self)	10

6.2 Full operator comparison (base used)

Table 2. Bases used by different systems for 8 ∘ 25%.

Operation	Calc	Sheets	Python/JS	Goblin %	Goblin %s
8 + 25%	8	8	(n/a)	1	8
8 − 25%	8	8	(n/a)	1	8
8 × 25%	1	1	(n/a)	1	8
8 ÷ 25%	1	1	(n/a)	1	8

7 PRACTICAL SYNTAX EXAMPLES

```
# Pseudocode illustrating Goblin-like behavior

price = $80

# Percent as type/literal
tip = 15%          # equals 0.15 (dimensionless)
also_tip = pct(15) # identical to 15%

# Default: percent of 1
two_dollars = 8 * 25%      # 2

# Calculator-style: percent of self (left)
with_tip = price + 15%s    # $80 + (0.15 * $80) = $92

# Explicit base
fee = 2% of price          # (0.02 * $80) = $1.60

# Functions
half = sqrt(25%)           # 0.5

# Modulus (spaced)
r = 8 % 3                  # 2
```

8 RELATED WORK

Our approach follows mainstream PL guidance that operators should be context independent and semantically well-defined [Pierce 2002; Harper 2016]. Pierce stresses the importance of clear typing and compositional meaning for language constructs, while Harper warns against context-sensitive overloading that obscures denotation. We treat percent as a proper value with explicit operators, rather than an overloaded token whose meaning varies by surrounding syntax.

On the end-user side, spreadsheet research repeatedly documents percentage-related errors and ambiguous formula intent [Panko 2008; Hermans et al. 2012]. By making the base explicit (`%s` or `of`) and by showing algebraic desugarings, Goblin's design targets these pitfalls directly: it provides the convenience that users expect, while retaining the formal clarity that languages require.

9 DISCUSSION AND LIMITATIONS

Our choice of defaulting bare `%` to "percent of 1" favors programming language conventions over calculator expectations. We recover calculator-style semantics explicitly via `%s` and `of`. This split keeps the type/operator model simple, avoids implicit context rules, and preserves standard modulus via whitespace disambiguation. Alternative defaults are possible, but they tend to reintroduce context dependence or sacrifice programmer expectations.

10 CONCLUSION

We presented a principled semantics for percent in Goblin: a first-class *percent* type (`n%`, `pct(n)`) and an explicit operator family (`%`, `%s`, `% of E`, and spaced `%` for modulus). The design is simple, compositional, and audit-friendly (with worked examples), and it aligns with established principles

for operator design [Pierce 2002; Harper 2016] while acknowledging real-world user expectations documented in spreadsheet research [Panko 2008; Hermans et al. 2012]. We believe this strikes a practical balance between clarity and convenience, and we offer it as a template for languages that want unambiguous percentage semantics without giving up usability.

REFERENCES

[Harper 2016] Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). Cambridge University Press.

[Hermans et al. 2012] Felienne Hermans, Michael Pinzger, and Arie van Deursen. 2012. Detecting code smells in spreadsheet formulas. In *Proceedings of the 34th International Conference on Software Engineering*, 441–450.

[Panko 2008] Raymond R. Panko. 2008. What we know about spreadsheet errors. *Journal of Organizational and End User Computing* 10, 2, 15–21.

[Pierce 2002] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.