Abstract Prefix Notation (APN): A Type-Encoding Naming Methodology for Programming

Jonas Immanuel Frey
Independent Researcher
First published: 2026

Abstract

This paper introduces **Abstract Prefix Notation (APN)**, a systematic naming methodology that encodes logical type information directly into identifiers across all programming languages. APN addresses a fundamental challenge in software development: the gap between how programmers think during active coding and how they (or others) understand code months later. By embedding abstract type information—numbers, strings, booleans, objects, arrays, functions—directly into variable and function names, APN creates self-documenting code that remains comprehensible in any context: plain text editors, printed paper, code reviews, or terminal output.

Keywords: naming convention, coding methodology, type notation, code readability, software engineering, self-documenting code, variable naming, programming pedagogy

---

1. Introduction

1.1 The Problem

Programming is a human cognitive activity before it is machine execution. During active coding, a programmer's short-term memory holds the complete mental model: the types of all variables, the return values of functions, the structure of data. After a break of days or months, this mental model degrades. Reading the same code becomes a process of reconstruction: *What type is this variable? What does this function return? Is this an array or a single object?*

The traditional solution—comments—has fundamental flaws. Comments are not enforced by the compiler or interpreter; they drift out of sync with code; they are often viewed as secondary to "working code." Developers, being human, prioritize making code work over maintaining comments.

1.2 The Solution: Abstract Prefix Notation

APN embeds crucial type information directly into identifier names. Instead of asking "what is this variable?" the name itself answers: `n_age` is a number, `a_o_user` is an array of objects (users), `f_n_sum` is a function returning a number. This information is always visible, never out of sync, and requires no tooling to access.

1.3 Scope and Applicability

APN is language-agnostic. It applies to:

- **Dynamically typed languages** (JavaScript, Python, Ruby, PHP, Lua) where type information is otherwise invisible
- **Statically typed languages** (Rust, Go, Java, C#) as a readability layer on top of the type system
- **Database schemas** through relational naming conventions
- **Mathematical and shader code** through normalized iteration patterns
- **Configuration files** and data serialization formats

While APN benefits all programming contexts, it provides the greatest value in dynamically typed languages where type information would otherwise be absent.

1.4 Key Insight: Logical vs. Concrete Types

APN does not encode concrete implementation types (e.g., `u32`, `i8`, `Vec<u8>`, `Uint8Array`). It encodes **logical types**—the small set of abstractions that appear in virtually all programming:

Logical Type	Prefix	Examples
Number	`n_`	0, 1, 6.2831, -1234
String	`s_`	"c", "hello", JSON
Boolean	`b_`	true, false
Object	`o_`	`{n: 6.2831, s: "Tau"}`
Array	`a_`	`[1, "Tau", true, []]`
Function	`f_`	`(a,b) => a + b`
Unknown	`v_`	Value of any type

These seven prefixes cover the vast majority of programming constructs across all languages. A Rust `Vec<u8>` is logically an array of numbers (`a_n_`). A Python dictionary is an object (`o_`). A UUID is a string (`s_`). APN operates at this abstract level, making it universally applicable.

---

2. Related Work and Context

2.1 Hungarian Notation

Charles Simonyi's Hungarian Notation (1970s) was the first systematic attempt to encode information in variable names. The original "Apps Hungarian" encoded semantic intent (e.g., `us` for unsafe string, `s` for safe string). The later "Systems Hungarian" (e.g., `lpszName` for "long pointer to null-terminated string") was widely criticized for encoding implementation details that change during refactoring. Joel Spolsky's essay "Making Wrong Code Look Wrong" (2005) rehabilitated Simonyi's original intent: encoding semantic invariants, not implementation types.

APN differs from Hungarian Notation in five key ways:

1. **Abstract type focus**: APN encodes logical types, not implementation details
2. **Compound prefixes**: `a_o_`, `a_n_` encode nested container types
3. **Function return types**: The second prefix in function names encodes return type
4. **Generic-first naming order**: `n_id__frame` (not `frame_id`)
5. **No plural words**: Cardinality expressed through `a_` prefix only

2.2 Static Type Systems

TypeScript, Flow, Python type hints, and MyPy add type information through annotations and tooling. They are powerful but require:

- Compilation or transpilation steps
- IDE support for full benefit
- Adoption of language supersets or extensions
- Runtime stripping (in JavaScript/TypeScript)

APN requires none of these. Type information is in the names themselves, visible in any context. The two approaches are complementary: a codebase can use both static typing and APN naming conventions.

2.3 JSDoc and Documentation Comments

JSDoc and similar systems embed type information in comments. This information is invisible without tooling, drifts out of sync, and requires maintenance effort. APN makes type information part of the code itself, eliminating the sync problem.

2.4 Existing Naming Conventions

Style guides (Google, Airbnb, MDN) address casing (camelCase, snake_case) and basic rules (no single-letter variables). They provide no systematic grammar for encoding type information. APN fills this gap with a composable, learnable system.

---

3. The APN Specification

3.1 Core Principles

APN rests on four foundational principles:

1. **Information visibility**: Crucial type information belongs in the name, not in comments or external tooling
2. **Logical abstraction**: Encode logical types, not implementation details
3. **Composability**: Prefixes combine to express complex structures
4. **Global consistency**: One standard across languages and contexts

3.2 Type Prefixes

Every variable name begins with a type prefix and an underscore:

Prefix	Type	Example
`v_`	Unknown / mixed	`v_result` (could be anything)
`n_`	Number	`n_age`, `n_tau`, `n_count`
`b_`	Boolean	`b_done`, `b_visible` (never `b_is_done`)
`s_`	String	`s_name`, `s_json`, `s_uuid`
`o_`	Object	`o_config`, `o_person`
`a_`	Array	`a_items`, `a_points`
`f_`	Function	`f_callback`, `f_map`

The `v_` prefix signals polymorphism—variables whose type intentionally varies or whose value may be null or undefined. It alerts readers that this identifier requires careful handling.

The `b_` prefix inherently denotes a boolean, making verbs such as `is_`, `has_`, or `can_` redundant. Write `b_done` rather than `b_is_done`—the prefix already communicates that the variable holds a truth value.

3.3 Compound Prefixes

When a container has a homogeneous inner type, prefixes compound:

Compound Prefix	Meaning	Example Value
`a_n_`	Array of numbers	`[1, 2, 3]`
`a_s_`	Array of strings	`["a", "b", "c"]`
`a_b_`	Array of booleans	`[true, false, true]`
`a_o_`	Array of objects	`[{n:1}, {n:2}]`
`a_a_`	Array of arrays	`[[1,2], [3,4]]`
`a_f_`	Array of functions	`[f_add, f_multiply]`

This creates a type algebra read left-to-right. `a_o_user` is parsed as "array of object-type user." The pattern extends to any depth: `a_a_o_point` is "array of arrays of point objects" (e.g., a list of polygons).

3.4 Return Type Encoding in Functions

Functions encode their return type as the second prefix:

Function Prefix	Return Type	Example
`f_n_`	Returns number	`f_n_sum = (a,b) => a + b`
`f_b_`	Returns boolean	`f_b_contains = (a, v) => a.includes(v)`
`f_s_`	Returns string	`f_s_label = (o) => o.name`
`f_o_`	Returns object	`f_o_person = (n,s) => ({n_age:n, s_name:s})`
`f_a_`	Returns array	`f_a_n_range = (n) => [...Array(n).keys()]`
`f_` (no second prefix)	Returns nothing (void)	`f_render = () => { draw(); }`

The presence or absence of a return type prefix is itself information: `f_render` is a procedure; `f_n_render` returns a number. This distinction is invisible in conventional naming.

Extensibility. APN prefixes operate at the abstract level, but the system is extensible. A simple `a_n` denotes an array of numbers with no further qualification. When additional specificity is useful, it can be appended without breaking the convention: `a_n_u8__image` communicates an array of unsigned 8-bit numbers representing image data. The abstract prefix (`a_n_`) remains intact and readable to any APN user, while the additional detail (`u8`, `__image`) provides context for those who need it. This makes APN backward-compatible by design—adding specificity never invalidates the base prefix.

3.5 Factory Functions Over Classes

APN recommends factory functions instead of classes:

```
// Conventional class (discouraged)
class Person {
  constructor(s_name, n_age) {
    this.s_name = s_name;
    this.n_age = n_age;
  }
}

// APN factory function
let f_o_person = function(s_name, n_age) {
  return { s_name, n_age };
};
```

Factory functions: - Clearly signal return type (`f_o_` = returns object) - Avoid `this` binding pitfalls - Are first-class values (can be passed, stored, composed) - Eliminate prototype chain complexity

3.6 No Plural Words

APN eliminates most plural forms from identifiers. The `a_` prefix already communicates multiplicity, making English plurals redundant:

Conventional	APN
users	a_o_user
numbers	a_n
functions	a_f
indices	a_n_idx

A variable named `a_s_name` is "an array containing string-type name values." The array itself is singular—it is *one* array. The `a_` prefix already communicates that it holds multiple elements.

In rare edge cases, a plural form may remain as part of a domain-specific term—for example, `o_file.n_bytes` is an acceptable property name where "bytes" denotes a unit of measurement rather than a collection of individual items.

The computational argument. English pluralization is irregular and unpredictable: `user` becomes `users`, but `index` becomes `indices`, `datum` becomes `data`, and `mouse` becomes `mice`. These irregularities make plurals unreliable for programmatic manipulation—a naive `{singular}s` transformation produces nonsense like `indexs` or `mouses`. The `a_` prefix provides a consistent, machine-parseable alternative: `a_n_idx` is always constructed the same way regardless of the English word involved. By removing plural forms from identifiers, APN establishes a uniform convention that works reliably across both human readers and automated tooling.

3.7 Generic-First Naming Order

Conventional naming places qualifiers first: `filteredUsers`, `startIndex`, `frameId`. This scatters related variables alphabetically.

APN aims to reverse this: the base concept comes first, qualifiers follow after a double underscore:

Conventional	APN
filtered_users	a_o_user__filtered
start_index	n_idx__start
end_index	n_idx__end
frame_id	n_id__frame
timestamp_ms	n_ms__timestamp

This pattern is particularly valuable when creating multiple instances of the same type:

```
let o_person__gretel = f_o_person('gretel', 20);
let o_person__olaf   = f_o_person('olaf', 19);
let o_person__daria  = f_o_person('daria', 34);
let o_person__fromdb = f_o__from_db(n_id);
```

Because all four variables share the `o_person__` prefix, they group together in sorted lists and are immediately recognizable as instances of the same model.

Benefits: - Related concepts group together in sorted lists - Qualifier is visually distinct (double underscore) - Reading order matches thinking: "user (filtered)" not "filtered user"

A note on double underscore usage. The double underscore convention is intentionally flexible. In many cases, a single underscore is sufficient and readable—`n_idx_start` is perfectly acceptable. The double underscore becomes important where ambiguity would otherwise arise. For example, `o_person_gretel` could be mistaken for a model or class name ("person_gretel"), whereas `o_person__gretel` makes clear that `gretel` is a qualifier on an `o_person` instance. Similarly, `a_o_person_filtered` could be misread as an array of "person_filtered" objects, while `a_o_person__filtered` unambiguously marks `filtered` as a qualifier. As a guideline: use the double underscore wherever a single underscore would create confusion between the base concept and its qualifier.

3.8 Wrapping and Unwrapping

APN prefixes are, by their nature, a wrapping system. Each prefix layer represents a container, and removing one layer of containment—through array access, function invocation, or parsing—corresponds to removing the outermost prefix. This relationship between prefix structure and data access is one of APN's most powerful properties.

Basic wrapping. At the simplest level, every APN prefix wraps a value in a type declaration:

```
let n_id = 2;  // n_ wraps the identification value as a number
```

Array unwrapping. An array adds one prefix layer. Accessing an element removes it:

```
let a_n_id = [2, 3, 4];   // a_ wraps n_id values into an array
let n_id = a_n_id[0];     // accessing an index removes the a_ layer
```

Function unwrapping. A function wraps its return value. Calling the function unwraps it:

```
let f_n_sum = function(n_a, n_b) {
  return n_a + n_b;
};
let n_sum = f_n_sum(1, 2);  // invocation removes the f_ layer
```

Multi-dimensional unwrapping. Compound prefixes unwrap one layer at a time, left to right:

```
let a_a_n_num = [[1, 2, 3], [4, 5, 6]];
let a_n_num = a_a_n_num[0];   // first a_ removed → a_n_num
let n_num = a_n_num[0];       // second a_ removed → n_num
```

A practical example—image data as nested arrays:

```
let a_a_a_n_rgba__image = [...];                    // 3D: rows of columns of RGBA channels
let a_n_rgba__pixel = a_a_a_n_rgba__image[20][50];  // pixel at row 20, column 50
let n_r = a_n_rgba__pixel[0];                       // red channel value
```

String unwrapping. Strings can encode serialized data. The prefix names both the current form (string) and the contained type:

```
// String containing a serialized function
let s_f_n_prod = '(n_a, n_b) => { return n_a * n_b }';
let f_n_prod = eval(s_f_n_prod);

// String containing JSON-encoded object
let s_json__o_person = '{"s_name":"hans","n_age":20}';
let o_person = JSON.parse(s_json__o_person);

// String containing HTML element markup
let s_o_el_html = '<h1>Hello</h1>';
let o_el_html = parser.parseFromString(s_o_el_html);
```

In each case, the variable name describes the transformation chain: reading the prefix from left to right shows how the data is currently packaged, and unwrapping operations peel away layers until the target type is reached.

3.9 Loop and Normalization Pattern

In mathematical and iterative contexts, APN defines a consistent pattern:

- `n_its_{domain}`: total iterations in a domain
- `n_it_{domain}`: current iteration (loop variable)
- `n_it_nor_{domain}`: normalized iteration [0.0, 1.0]

```javascript
// Generate polygons with normalized coordinates
let n_its__polygon = 5;
let n_its__corner = 3;
let n_tau = Math.PI * 2;

for (let n_it_polygon = 0; n_it_polygon < n_its__polygon; n_it_polygon++) {
  let n_it_nor_polygon = n_it_polygon / n_its__polygon;  // [0, 1)

  for (let n_it_corner = 0; n_it_corner < n_its__corner; n_it_corner++) {
    let n_it_nor_corner = n_it_corner / n_its__corner;
    // Use normalized values for parametric calculations
  }
}
```

The normalized variable `n_it_nor_polygon` is structurally distinct from the raw index, and its prefix communicates that it is a unitless ratio.

3.10 Scope-Aware Naming and Redundancy Prevention

APN encourages adapting identifier specificity to the current scope. When context already establishes what type of object is being handled, names can be shortened to avoid redundancy:

```javascript
// In a narrow scope, 'o' suffices-the context makes the type unambiguous
let a_o_person__filtered = a_o_person.filter(
  o => o.s_name.length > 10
);

// In a broader scope, disambiguation becomes necessary
let f_a_o_person__filtered = function(a_o_person, n_len_name__min) {
  let a_o = [];  // 'a_o' rather than 'a_o_person'-no ambiguity within this scope
  for (let o_person of a_o_person) {
    // 'o_person' distinguishes loop variable from objects already in a_o
    if (o_person.s_name.length >= n_len_name__min) {
      a_o.push(o_person);
    }
  }
  return a_o;
};
```

The principle is straightforward: include only as much specificity as the scope requires. In a single-line callback where only one object exists, `o` is sufficient. In a function body where multiple object references coexist, qualified names prevent confusion.

3.11 Relational Database Naming

APN extends to database schemas:

Element	APN Convention	Example
Table	`a_o_{entity}`	`a_o_person`
Integer primary key	`n_id`	`n_id`
UUID primary key	`s_id`	`s_id`
Foreign key	`n_o_{entity}_n_id`	`n_o_person_n_id`

The foreign key convention derives from how properties are accessed in code. A person's identifier is accessed as `o_person.n_id`. To construct a foreign key column name, this access path is flattened into a single

identifier: `n_o_person_n_id`. The leading `n_` prefix is required because APN mandates that every identifier reflect its abstract type—in this case, the foreign key column holds a number. The general pattern is `n_{entity}_{property}`, yielding names that are both self-describing and traceable back to their source model.

3.12 Standard Abbreviation Glossary

Establishing a universal abbreviation standard is inherently difficult—many common abbreviations collide across domains, and no single glossary can satisfy all contexts. Nevertheless, certain abbreviations are widely shared across programming languages and communities. APN suggests the following as a recommended baseline:

Abbreviation	Meaning
`idx`	index
`pos`	position
`off`	offset
`el`	element
`evt`	event
`val`	value
`len`	length
`sz`	size
`cnt`	count
`cur`	cursor
`ptr`	pointer
`db`	database
`ms`	milliseconds
`us`	microseconds
`ns`	nanoseconds
`sec`	seconds
`ts`	timestamp
`dt`	delta time
`ttl`	time to live
`its`	iterations (count)
`it`	iteration (current)
`nor`	normalized [0,1]
`trn`	translation (geometry)
`scl`	scale (geometry)
`rot`	rotation (geometry)
`fps`	frames per second
`raf`	request animation frame

A long variable name is always preferable to an ambiguous abbreviation. Domain-specific projects may extend this glossary as needed, provided extensions are documented and used consistently within the codebase.

These abbreviations compose naturally with APN prefixes:

Expression	Meaning
`n_ts_ms__created`	Millisecond timestamp of when an instance was created
`n_ts_ms__updated`	Millisecond timestamp of when an instance was last updated
`n_trn_x`	X component of a translation vector
`n_scl_x`	X component of a scale vector

---

4. Examples

4.1 User Management Example

```
// Factory function for user objects
let f_o_user = function(s_name, n_age) {
  return { s_name, n_age };
};

// Array of users
let a_o_user = [
  f_o_user('gretel', 20),
  f_o_user('olaf', 19),
  f_o_user('daria', 34)
];

// Function returning boolean
let f_b__adult = function(n_age, n_age__min) {
  return n_age > n_age__min;
};

// Function returning filtered array
let f_a_o_user__adult = function(a_o_user, n_age__min) {
  return a_o_user.filter(o => f_b__adult(o.n_age, n_age__min));
};


// Function returning formatted string
let f_s_userinfo = function(o_user) {
  return `${o_user.s_name} (age ${o_user.n_age})`;
};

// Usage
let a_o_user__adult = f_a_o_user__adult(a_o_user, 18);
let a_s_userinfo__adult = a_o_user__adult.map(f_s_userinfo);
let v_o_user__ludwig = a_o_user.filter(o=>o.s_name == 'ludwig')?.at(0);
let b_ludwig_existing = v_o_user__ludwig !== undefined
```

4.2 Geometry Example

```
// Configuration
let n_its_point = 100;
let n_radius = 50;
let n_tau = Math.PI * 2;
```

```javascript
// Generate circle points
let a_o_point = [];

for (let n_it_point = 0; n_it_point < n_its_point; n_it_point++) {
  let n_it_nor_point = n_it_point / n_its_point;

  a_o_point.push({
    n_x: Math.cos(n_it_nor_point * n_tau) * n_radius,
    n_y: Math.sin(n_it_nor_point * n_tau) * n_radius
  });
}

// Transform: translate all points
let o_trn = { n_x: 100, n_y: 100 };
let a_o_point__translated = a_o_point.map(o_point => ({
  n_x: o_point.n_x + o_trn.n_x,
  n_y: o_point.n_y + o_trn.n_y
}));
```

4.3 Data Transformation Example

```javascript
// Raw data (from API)
let s_json__a_o_product = await fetch('/api/products').then(r => r.text());

// Parsed data
let a_o_product = JSON.parse(s_json__a_o_product);

// Derived data
let a_n_price = a_o_product.map(o_product => o_product.n_price);
let n_price__max = Math.max(...a_n_price);
let n_price__min = Math.min(...a_n_price);
let n_price__avg = a_n_price.reduce((a,b) => a + b, 0) / a_n_price.length;

// Filtered view
let a_o_product__expensive = a_o_product.filter(o => o.n_price > n_price__avg);
```

4.4 Object Maps

When an object is used as a key-value map rather than a structured record, APN encodes the types of both keys and values in the prefix: `o_{key_type}_{key_name}_{value_type}_{value_name}`.

```javascript
// Map from file extension (string) to MIME type (string)
let o_s_ext_s_mime = {
  '.jpg':  'image/jpeg',
  '.jpeg': 'image/jpeg',
  '.png':  'image/png',
  '.gif':  'image/gif',
  '.webp': 'image/webp',
  '.svg':  'image/svg+xml',
  '.pdf':  'application/pdf',
  '.json': 'application/json',
  '.txt':  'text/plain',
  '.html': 'text/html',
  '.css':  'text/css',
```

```
  '.js':   'application/javascript'
};

// A map can be converted to an array of structured objects
let a_o_mime = Object.keys(o_s_ext_s_mime).map(s => ({
  s_ext: s,
  s_mime: o_s_ext_s_mime[s]
}));

// Resulting structure:
// [{ s_ext: '.jpg', s_mime: 'image/jpeg' }, ...]
```

This convention distinguishes structured objects (`o_person` with known fields) from dictionary-like maps (`o_s_ext_s_mime` mapping strings to strings), resolving an ambiguity that a bare `o_` prefix would leave open.

---

5. Discussion

5.1 Cognitive Benefits

APN fundamentally changes how programmers read code. Instead of:

1. Encounter variable
2. Jump to definition or hover for type
3. Return to usage site
4. Interpret operation

The reader simply:

1. Read the prefix
2. Understand the type
3. Continue reading

This reduction in context switching reduces cognitive load.

5.2 Programmatic Usage of Prefixes

The underscore-delimited structure of APN identifiers carries a secondary advantage: prefix strings can be manipulated programmatically. Because type information follows a consistent, parseable pattern, it becomes possible to generate or inspect identifiers in code. The longer variable names are a deliberate trade-off—what is spent in characters is recovered in machine-readability.

For example, a database table name can be derived from a model name:

```
let s_prefix__array = 'a_';
let f_s_name__table = function(s_name__model) {
  return `${s_prefix__array}${s_name__model}`;
};
```

Similarly, a type-inference utility can extract the logical type from any APN-conformant identifier:

```
let f_s_type_from_name = function(s_name) {
  let o_s_prefix_s_type = {
    'n_': 'number',
    'b_': 'boolean',
    's_': 'string',
    'o_': 'object',
    'a_': 'array',
```

```
    'f_': 'function'
  };
  let s_prefix = s_name.slice(0, 2);
  return o_s_prefix_s_type[s_prefix] || 'unknown';
};
```

This programmatic accessibility is not an additional feature but an inherent property of encoding type information in a consistent, machine-readable format.

5.3 Error Prevention Through Visual Salience

When types are encoded in names, type mismatches become visually apparent:

- Passing `n_age` where `s_name` is expected looks wrong
- Iterating over `o_user` (an object) with `for...of` is visually flagged
- Assigning `f_b_something()` result to `n_count` is immediately incongruent

As Spolsky (2005) argued, the goal is to "make wrong code look wrong." APN achieves this through systematic type encoding.

5.4 Brain Reshaping

APN is not merely a notation—it reshapes how programmers mentally model code. The prefixes become pictograms: `a_` signals multiplicity, `n_` signals quantity, `b_` signals truth value. This mental mapping, reinforced through consistent use, creates a form of "programming literacy" that transcends individual languages.

5.5 Comments Become Optional

When crucial type information is in the names themselves, comments can focus on their proper role: explaining *why*, not *what*. Comments explaining what a variable contains become redundant and are naturally eliminated. The remaining comments provide true meta-information that cannot be encoded in names.

5.6 Universality Across Languages

APN works in any language that allows underscores in identifiers. This includes:

Language Family	Examples
C-family	JavaScript, TypeScript, C, C++, C#, Java, Go, Rust
Scripting	Python, Ruby, PHP, Lua, Perl
Functional	Elixir, Erlang, F#
Database	SQL (table/column names)
Configuration	JSON, YAML, TOML (key names)

A developer moving between languages carries APN with them—the notation is independent of syntax.

5.7 Limitations

Learning curve: Developers must learn the prefix system. This upfront cost is comparable to learning any team style guide.

Verbosity: APN names are sometimes longer. This is an intentional trade-off: more information in exchange for more characters. Storage is cheap; developer time is expensive.

No enforcement: APN relies on developer discipline. A linter could enforce conventions, but the base system is human-enforced.

IDE friction: Some tools assume camelCase and may not optimally support snake_case prefixes. This is a minor inconvenience compared to the readability benefits.

Low-level contexts: In systems where all variables are numbers (e.g., WebGL shaders, embedded systems with only integers), the `n_` prefix becomes redundant. APN can be relaxed in such contexts.

---

6. Conclusion

Abstract Prefix Notation (APN) is a systematic naming methodology that encodes logical type information directly into identifiers. It is:

- **Universal**: Works in any programming language
- **Tooling-free**: Information is in the names, not in external systems
- **Composable**: Prefixes combine to express complex structures
- **Self-documenting**: Type information is always visible
- **Cognitively aligned**: Reshapes how programmers mentally model code

APN does not replace static type systems, linters, or good software engineering practices. It complements them by making type information visible in the one place programmers always look: the names of things.

The methodology establishes a global standard for identifier naming—a small step toward making code more readable and more maintainable.

6.1 Code Is Not Natural Language

APN does not attempt to make code read like prose. Code is fundamentally unlike natural language—it operates on formal logic, strict sequencing, and precise data transformations. Efforts to make code resemble English often obscure these structural realities rather than clarifying them.

APN takes the opposite approach: rather than adapting code to human language, it adapts human naming habits to the demands of code. Programmers who internalize the prefix system begin to think in terms of data types and transformations, which is precisely the mental model that effective programming requires. For those who prefer code that reads like natural language, modern large language models and conversational coding tools already serve that purpose.

6.2 APN Formalizes What Programmers Already Do

APN does not introduce a novel idea so much as it standardizes an existing, widespread practice. Programmers have always embedded type hints in their naming—they simply lacked a consistent grammar for doing so.

Consider `cv2.createCLAHE()` from OpenCV: the `create` prefix tells the reader that the function returns a CLAHE object. In APN, this becomes `f_o_clahe`—shorter, systematic, and unambiguous. The pattern is pervasive: `getUsers`, `fetchOrders`, `buildConfig`, `createSocket`—all use ad-hoc English verbs to communicate return types. Each library, each team, each developer invents their own flavor. A reader must learn that `get` in one codebase means the same as `fetch` in another, or that `create` and `build` and `make` all signify object construction.

APN replaces this fragmented landscape with a single, learnable standard. `f_o_` always means "function returning an object." `f_a_` always means "function returning an array." There is no ambiguity, no per-project dialect, and no reliance on the nuances of English vocabulary. The result is a naming system that transfers across codebases, teams, and programming languages without requiring relearning.

References

[1] Simonyi, C. (1999). "Hungarian Notation." MSDN Library, Microsoft Corporation.

[2] Spolsky, J. (2005). "Making Wrong Code Look Wrong." Joel on Software.

Appendix: Quick Reference

```
TYPE PREFIXES
  v_    unknown / mixed / null
  n_    number
  b_    boolean (never b_is_)
  s_    string
  o_    object
  a_    array
  f_    function

COMPOUND PREFIXES (arrays)
  a_n_    numbers
  a_s_    strings
  a_b_    booleans
  a_o_    objects
  a_a_    arrays
  a_f_    functions

FUNCTION RETURN TYPES
  f_n_    returns number
  f_b_    returns boolean
  f_s_    returns string
  f_o_    returns object (factory)
  f_a_    returns array
  f_      returns nothing (void)

QUALIFIER SEPARATOR (double underscore)
  a_o_user__filtered    (not: filtered_users)
  n_idx__start          (not: start_index)
  n_id__frame           (not: frame_id)

NAMING ORDER
  Generic → Specific
  n_id__frame      (not: frame_id)
  n_ms__timestamp  (not: timestamp_ms)

DATABASE
  Table:        a_o_{entity}
  Primary key:  n_id (integer) or s_id (UUID)
  Foreign key:  n_o_{entity}_n_id

NO PLURAL WORDS
  a_o_user   (not: users)
  a_n_score  (not: scores)

LOOP PATTERN
  n_its_{domain}  = total iterations
```

```
n_it_{domain}   = current iteration
n_it_nor_{domain} = normalized [0,1]
```

First published: 2026
Author: Jonas Immanuel Frey