Morphosyntactic Programming

Case, Mood, and Type-Directed Disambiguation for Turkish-Like Syntax

JOOMY KORKUT, Bloomberg, USA
ALPEREN KELES, University of Maryland, USA
ONUR AKDEMIR, Topsort, USA

We present Kip, a statically typed functional programming language with a syntactic purity discipline, whose surface syntax is in Turkish and whose type checking is guided by Turkish morphology. Kip uses case suffixes to determine which parameter each argument belongs to, so arguments can be given in any order when the cases uniquely identify them. Kip also carries morphological ambiguity through parsing and resolves it during type checking via type-directed constraint solving. Finally, Kip enforces a lightweight effect discipline via syntax: pure definitions are noun phrases, while effectful computations are infinitival verb phrases, and effectful calls appear in the imperative mood. We also describe the Rocq mechanization of Kip's calculus, which proves progress, preservation, and elaboration correctness.

CCS Concepts: • **Software and its engineering** → **Functional languages**; *Language features*; • **Human-centered computing** → Natural language interfaces.

Additional Key Words and Phrases: linguistics, morphosyntax, grammatical case, grammatical mood, type systems, type-directed disambiguation

ACM Reference Format:
Joomy Korkut, Alperen Keles, and Onur Akdemir. 2026. Morphosyntactic Programming: Case, Mood, and Type-Directed Disambiguation for Turkish-Like Syntax. *Proc. ACM Program. Lang.* 0, 0, Article 0 (March 2026), 24 pages.

1 INTRODUCTION

Most mainstream programming languages use English-oriented keywords and naming conventions, largely due to historical and ecosystem-related reasons rather than technical necessity. As a result, many surface-language design decisions are implicitly English-centric: argument order is primarily positional, meaning is carried by keywords and punctuation rather than by morphological structure. A number of projects have experimented with non-English syntax, but most translate keywords while leaving the underlying structure unchanged—an approach that broader localization research identifies as only one of several relevant design dimensions [25].

Morphologically rich languages such as Turkish, Finnish, Hungarian, German, Russian, and Latin mark grammatical roles directly on words through inflectional suffixes [5]. In Turkish, for instance, case suffixes indicate whether a noun phrase serves as a subject, direct object, recipient, location, source, modifier, or instrument. English typically encodes these relations through position and prepositions. This raises a design question for programming languages: can *morphosyntax*—the interplay of word form and syntactic function—participate directly in static reasoning, rather than being discarded after parsing as presentation-only syntax?

Kip explores this question.[1] It is a statically typed functional language whose surface syntax is based on Turkish and whose elaboration is morphology-aware. Rather than normalizing away inflection, Kip preserves case information after parsing and uses it operationally in name resolution,

[1]Kip is Turkish for "grammatical mood." Source code and artifacts are available at https://github.com/kip-dili/kip and https://github.com/kip-dili/metatheory.

Authors' addresses: Joomy Korkut, Bloomberg, New York, New York, USA, jkorkut@bloomberg.net; Alperen Keles, University of Maryland, College Park, Maryland, USA, akeles@umd.edu; Onur Akdemir, Topsort, San Francisco, California, USA, onur@topsort.com.

overload selection, and argument-role recovery. Case annotations are part of types, so role information participates directly in type-directed elaboration.

As a brief illustration, a binary function signature in Kip marks each argument's role with a case suffix on the type phrase:

Surface

```
(bu doğal-sayıyla) (şu doğal-sayının) toplamı,
  ...
```

Morphological Gloss

```
(this:N:NOM natural:N:NOM-number:N:INS)
(that:N:NOM natural:N:NOM-number:N:GEN) sum:N:P3S,
  ...
```

Here `doğal-sayıyla` carries the instrumental suffix *-(y)la* (roughly "with a natural number") and `doğal-sayının` carries the genitive suffix *-(n)ın* ("of a natural number"). Because the two cases are distinct, arguments are recoverable by role rather than position and can be supplied in either order when recovery is unique. Conversely, when argument roles are not distinguishable by case alone—for example, when multiple parameters bear the same case—Kip falls back to ordinary positional interpretation.

The central claim of this paper is that, in a morphologically rich language, inflection can participate directly in typed elaboration rather than being discarded after parsing. Kip can be read as a morphology-aware presentation of a mostly Haskell-like core, with a deliberately smaller surface: effectful computation exists without a user-denotable effect type constructor, and overloading is resolved through case/type-directed elaboration rather than through type classes. The language follows broad Turkish word-order tendencies—head-final noun phrases for pure definitions and predominantly SOV (subject–object–verb) verbal clauses for effectful computation. Operationally, the implementation parses surface forms into one or more morphological analyses, resolves them against scope and interface shape, uses case and type constraints to recover roles and prune overloads, and then evaluates or compiles the resolved program. Competing analyses are retained until typing evidence is sufficient or reported explicitly if ambiguity remains. When constraints are insufficient, the programmer can force a reading explicitly (e.g., with apostrophes).

These design choices yield four contributions:

(1) A case-structured interface discipline for functions and constructors, with elaboration rules that recover argument roles and permit non-positional application when recovery is unique.
(2) A type-directed account of morphological ambiguity in which analyses from the parser are carried forward and pruned by case and typing constraints, with explicit user-level disambiguation when needed.
(3) A syntactic purity boundary that maps noun-phrase forms to pure code and infinitive/imperative forms to effectful code, integrated into the static checker and exercised across interactive and file-based programs.
(4) A Rocq mechanization of the core calculus covering case-aligned application, effect rejection, small-step semantics, progress, preservation, and algorithmic overload elaboration with machine-checked correctness proofs.

The paper focuses primarily on the design, implementation, and formalization of these mechanisms rather than an empirical usability study. Section 2 introduces the surface fragment needed for the main technical story. Section 3 addresses morphological ambiguity and type-directed disambiguation. Section 4 discusses design trade-offs and practical limitations, Section 5 presents a Rocq mechanization of the core metatheory and elaboration, and Section 6 surveys related work.

2 SYNTAX THROUGH EXAMPLES

This section presents the core language forms needed for the rest of the paper.[2] Throughout the section, each example is shown in surface Kip syntax alongside a morphological gloss version of the same syntax, where every word in the original is translated to English, and annotated with the morphological parts. Where useful, we also give a brief functional programming comparison for readers more familiar with that tradition. At the level of computation, Kip behaves much like a conventional functional language; what differs is the way interfaces and calls expose grammatical role information.

2.1 Basics

We begin with the simplest forms: constants, naming conventions, and basic types.

2.1.1 Constants. A constant definition is a copular sentence:

Surface
```
başlangıç-sayısı, 0'dır.
cevap, 42'dir.
```

Morphological Gloss
```
start:N:NOM-number:N:P3s, 0:NOM:COP.
answer:N:NOM, 42:NOM:COP.
```

That is, `başlangıç-sayısı` is 0 and `cevap` is 42. In functional terms, these are ordinary top-level value bindings.

Names are generally hyphenated multi-word identifiers,[3] for example `başlangıç-sayısı` and `tam-sayı-hali`. In the analyzed column, we use a fixed tag vocabulary: :NOM, :ACC, :DAT, :LOC, :ABL, :GEN, :INS, :P3S, :COND, :COP, :INF, :IMP, :CVIP. We also include coarse part-of-speech tags in the gloss: :N marks nouns and :V marks verbs. The analyzed examples use a compact gloss style, with morpheme/tag information appended after the token (e.g., `42:NOM:COP`). For PL readers, these tags can be read as lightweight role/mode annotations used by elaboration:

- :NOM nominative (subject-like), :ACC accusative (direct-object-like), :DAT dative (to/for-like), :LOC locative (at/in/on-like), :ABL ablative (from/out-of-like), :GEN genitive (of/owner-like), :INS instrumental (with/by-like).
- :P3S third-person singular possessive—a linker that ties nominal compounds (common in function/type names).
- :COND conditional copula ("if…"), :COP present-tense copula ("is"), :INF infinitive (action definitions), :IMP imperative (calls), :CVIP *-ip* converb ("having done…, then…").

2.2 Functions

With basic definitions in place, we turn to functions—their definitions, call-site syntax, argument reordering, partial application, and higher-order use.

2.2.1 Function Definitions and Calls. A function definition has case-marked arguments, then a head noun:

Surface
```
(bu tam-sayının) karesi,
  bunla bunun çarpımıdır.
```

Morphological Gloss
```
(this:N:NOM integer:N:GEN) square:N:P3s,
  this:N:INS this:N:GEN product:N:P3S:COP.
```

This defines the square of an integer. Here the genitive case on the argument type makes the parameter role explicit—something positional syntax leaves unmarked.

[2] Quick guide to syntax highlighting in this paper: **keywords**, `types` or `type constructors`, `data constructors` or `literals`, `functions`, *`bound names`*.

[3] A variable naming style also known as *kebab-case* in the programming community, which feels appropriate here.

At call sites, arguments appear before the nominal head, following Turkish noun-phrase structure, so that function application reads as a noun phrase rather than a keyword-delimited form.

Surface
```
(5'in karesini) yaz.
```

Morphological Gloss
```
(5:GEN square:N:P3S:ACC) write:V:IMP.
```

The expression applies `kare` to 5 and prints the result. This is ordinary application followed by printing.

Kip also permits effectful definitions, marked here by the infinitive and invoked in the imperative:

Surface
```
selamlamak,
  isim için okuyup,
  ("Merhaba "yla ismin birleşimini) yazmaktır.

selamla.
```

Morphological Gloss
```
greet:V:INF,
  name:N:NOM for read:V:CVIP,
  ("Hello ":INS name:N:GEN
   union:N:P3S:ACC) write:V:INF:COP.

greet:V:IMP.
```

This definition reads a name and writes a greeting. The next line invokes it imperatively. The verbal morphology is what marks it as effectful. Because effectful Kip programs have the same sequential structure as monadic Haskell, we show the corresponding Haskell program for comparison:

Haskell
```haskell
greet = do
  name <- readLn
  putStrLn ("Hello " ++ name)

main = greet
```

Converb sequencing (**`-ip`**) corresponds to **`do`**-notation lines, **`için`** binding corresponds to `<-`, and the imperative invocation `selamla` corresponds to `main` = `greet`.

2.2.2 Case-Marked Function Signatures. Function headers declare argument roles through case-marked type phrases. Returning to the sum signature from Section 1, the instrumental and genitive suffixes serve as role annotations: each parameter carries an annotation κ_i and type τ_i. A call site supplies arguments with observed cases κ_i', and elaboration attempts to align them with the expected signature.

2.2.3 Case-Directed Application and Elaboration.

Unordered Application When Roles Are Recoverable. Kip allows arguments to appear in non-positional order when case roles are sufficient for recovery. Operationally, call resolution proceeds in four stages:

(1) Filter candidate function signatures by arity.
(2) Filter by case compatibility.
(3) Filter by type compatibility (allowing unknowns during inference).
(4) Reorder surviving arguments into signature order and elaborate the call.

Within step (2), case matching is strict by default for exact applications. The implementation then makes two limited exceptions. Pattern-bound variables and function-typed arguments may by accepted at a wider range of case positions. Constructor-origin arguments remain strictly case-matched. This keeps reordering useful without making case matching overly permissive. When multiple parameters share the same case, case information alone cannot determine a permutation. In such cases, matching is effectively positional within the repeated-case segment.

Partial Application. When fewer arguments than a signature's arity are supplied, Kip matches supplied argument cases to unique parameter positions and returns a residual function type for the remaining positions.

The idea is easiest to see in a one-argument section:

Surface
```
(3'le (1'in toplamını)) yaz.
```

Morphological Gloss
```
(3:INS (1:GEN sum:N:P3S:ACC)) write:V:IMP.
```

The inner expression partially applies `toplam` to one argument; the outer call supplies the remaining one. As in curried functional languages, each partial application returns a closure over the remaining parameters; case information determines which positions have already been filled.

Higher-order use follows the same pattern:

Surface
```
(bu a listesinin) (şey a'nın b'siyle) eşlemi,
  bu boşsa, boş;
  ilkin devama ekiyse,
    (ilkin şeyinin)
      (devamın şeyle eşlemine) ekidir.
```

Morphological Gloss
```
(this:N:NOM a:N:NOM list:N:P3S:GEN)
(thing:N:NOM a:N:GEN b:N:P3S:INS) map:N:P3S,
  this:N:NOM empty:N:COND, empty:N:NOM;
  head:N:GEN rest:N:DAT addition:N:P3S:COND,
    (head:N:GEN thing:N:P3S:GEN)
    (rest:N:GEN thing:N:INS map:N:P3S:DAT)
    addition:N:P3S:COP.
```

This function applies a given function to each element of a list. Its overall type is the familiar polymorphic `map`: from functions (`a -> b`) and `a` lists to `b` lists. The phrase `a'nın b'si` (*a:N:GEN b:N:P3S*, literally: "*a*'s *b*") names the function type `a -> b`.

At the call site, the same mechanism supports compact higher-order composition:

Surface
```
[1, 2, 3]'ün (2'yle çarpımıyla)
  eşleminin toplamını yaz.
```

Morphological Gloss
```
[1, 2, 3]:GEN (2:INS product:N:P3S:INS)
  map:N:P3S:GEN sum:N:P3S:ACC write:V:IMP.
```

Here `2'yle çarpımı` is partially applied and then passed to `eşlem`.

This is equivalent to the Haskell expression `print (sum (map (* 2) [1, 2, 3]))`.

Name Resolution and Candidate Sets. Identifiers are parsed into candidate sets rather than single names. The checker resolves these sets against scope, arity, and case. If exactly one candidate remains, resolution succeeds. If none remain, the checker reports unknown-name or no-type errors. If multiple remain, ambiguity is reported explicitly.

These rules define the precise boundary of ambiguity in Kip, parser ambiguity is acceptable; unresolved checker ambiguity is not.

2.3 Data

Kip's algebraic data types follow the same noun-phrase pattern, with constructors and type parameters declared through case-marked interfaces.

2.3.1 *Simple Algebraic Data Types.* Enumeration-style types use **`Bir ... olabilir.`**

Surface
```
Bir doğruluk
ya doğru
ya da yanlış
olabilir.
```

Morphological Gloss
```
An boolean:N:NOM
either true:N:NOM
or false:N:NOM
can-be.
```

The type `doğruluk` has two alternatives: true and false.

In standard ADT terms, this is a two-constructor enumeration.

Recursive variants follow the same style. For example, natural numbers:

Surface

```
Bir doğal-sayı
ya sıfır
ya bir doğal-sayının ardılı
olabilir.
```

Morphological Gloss

```
A natural:N:NOM-number:N:NOM
either zero:N:NOM
either a natural:N:NOM-number:N:GEN
        successor:N:P3S
can-be.
```

The natural-number declaration provides constructors for zero and successor.

This is a standard Peano-style natural-number ADT.

2.3.2 Data Constructors and Function Signatures as Noun Phrases. Kip declares algebraic data types using Turkish copular declaration clauses (**`Bir ... olabilir.`**). The noun-phrase structure holds at the constructor and signature level: constructor heads are nominal, their arguments are case-marked dependents, and function signatures follow the same nominal interface style. ADT declarations as a whole are not themselves noun phrases; the noun-phrase structure operates at the interface level.

In the natural-number example above, the constructor `ardıl` takes one argument typed as `doğal-sayı` in genitive case. Because constructors appear in possessed form, nested values are also nested noun phrases (e.g., `sıfırın ardılı`, then `(sıfırın ardılının) ardılı`).

Polymorphic types use the same surface pattern. For example, a list type introduces a type variable alongside its constructors:

Surface

```
Bir öğe listesi
ya boş
ya da bir öğenin bir öğe listesine eki
olabilir.
```

Morphological Gloss

```
An element:N:NOM list:N:P3S
either empty:N:NOM
or a element:N:GEN a element:N:NOM
   list:N:P3S:DAT addition:N:P3S
can-be.
```

The polymorphic list declaration has empty and cons alternatives.

In standard ADT terms, this is a parametric list-like type.

Here `öğe` is a type variable, and `ek` takes two arguments with distinct roles: head in genitive and tail in dative.

2.3.3 Single-Constructor and Multi-Parameter Types. The same pattern extends smoothly to single-constructor wrappers and to multi-parameter types such as pairs and result-like encodings. These variants do not introduce new static machinery; they reuse the same case-marked nominal interface discipline shown above.

2.4 Patterns

Pattern matching in Kip covers constructor patterns as well as literal and list patterns, all expressed through Turkish conditional morphology.

Pattern branches are expressed with the Turkish conditional suffix *-sa/-se*, with **`değilse`** as catch-all. A typical definition uses constructor-shaped branches:

Surface

```
bu sıfırsa, ...; öncülün ardılıysa, ...
```

Morphological Gloss

```
this:N:NOM zero:N:COND, ...;
predecessor:N:GEN successor:N:P3S:COND, ...
```

The branch forms illustrate constructor-guarded clauses: the conditional suffix *-sa/-se* marks the pattern-match branching point. The scrutinee is *bu*, and each branch pattern is matched against that value.

Functionally, this corresponds to constructor-pattern clauses in **case** or equation style.

A complete example shows how these conditional branches compose in a recursive list-length definition:

Surface

```
(bu öğe listesinin) uzunluğu,
  bu boşsa,
    0;
  ilkin devama ekiyse,
    (devamın uzunluğuyla) 1'in toplamıdır.
```

Morphological Gloss

```
(this:N:NOM element:N:NOM list:N:P3S:GEN)
length:N:P3S,
  this:N:NOM empty:N:COND,
    0:NOM;
  head:N:GEN rest:N:DAT addition:N:P3S:COND,
    (rest:N:GEN length:N:P3S:INS) 1:GEN
    sum:N:P3S:COP.
```

The function computes list length by returning 0 for the empty case and adding 1 in the cons branch. Here again, the scrutinee is *bu* (the list argument), matched first against `boş` and then against `ek`.

This is a recursive `length`-style list function.

Checking a clause unifies the scrutinee type with the constructor result type and propagates the resulting substitution to bound pattern variables. Because patterns can nest, constructor arguments can themselves be matched by further constructor, literal, or list structure.

The checker also performs exhaustiveness analysis, reporting missing patterns unless an explicit wildcard branch is present. Beyond constructor patterns, the implementation supports literal patterns for integers, floats, strings, and list literals. Repeated binder names within a single branch are rejected.

2.4.1 Literal and List Patterns. Literal patterns use the same conditional morphology:

Surface

```
(bu tam-sayının) sınıfı,
  bu, 0'ysa, "sıfır";
  1'se, "bir";
  değilse, "diğer".
```

Morphological Gloss

```
(this:N:NOM integer:N:GEN) class:N:P3S,
  this:N:NOM, 0:COND, "zero":NOM;
  1:COND, "one":NOM;
  otherwise, "other":NOM.
```

The function classifies an integer using literal branches for 0 and 1, with a fallback.

This corresponds to literal-pattern matching on `Integer`.

String and list literals are handled similarly; they add no new control-flow form beyond the conditional branching already shown here.

2.5 Effects

So far, all definitions have been nominal. We now turn to effectful computation, where Kip uses verbal morphology to mark the boundary explicitly.

Here the purity boundary is grammatical rather than annotation-driven: nominal forms denote pure definitions, while infinitival verb forms denote computations whose use is restricted by the checker.

Effectful sequencing uses converbs (e.g., *-ip*) and binding with **`için`**: the verbal head appears at the end of the clause, so effectful forms remain SOV-like in the same head-final sense.

Surface

```
selamlamak,
  isim için okuyup,
  ("Merhaba "yla ismin birleşimini) yazmaktır.
```

Morphological Gloss

```
greet:V:INF,
  name:N:NOM for read:V:CVIP,
  ("Hello ":INS name:N:GEN
   union:N:P3S:ACC) write:V:INF:COP.
```

Returning to the greeting example from Section 2.2, the effectful definition reads a name and writes a greeting in sequence.

One can read this as a sequential `IO ()` action. Grammatical mood, rather than an explicit source-level effect constructor, marks that distinction on the surface.

This design is intentionally lightweight: Kip does not expose an explicit source-level monad, but it still enforces a pure/effectful boundary through grammar and checking.

2.5.1 Sequencing and Local Binding. When an intermediate result is needed, the **`için`** keyword binds it to a local name. The `selamlamak` example above already illustrates this: **`için`** binds the result of `oku` to *`isim`*, which is then used in the output expression. In more familiar functional notation, this plays the role of monadic bind.

File I/O follows the same pattern, combining sequencing with pattern matching on the result:

Surface

```
çalıştırmak,
  "./yazı.tmp"ye "selam kip"i yazıp,
  sonuç için "./yazı.tmp"'den okuyup,
  sonuç yokluksa,
    durmaktır;
  metnin varlığıysa,
    metni yazmaktır.
```

Morphological Gloss

```
run:V:INF,
  "./text.tmp":DAT "hello kip":ACC write:V:CVIP,
  result:N:NOM for "./text.tmp":ABL read:V:CVIP,
  result:N:NOM absence:N:COND,
    stop:V:INF:COP;
  text:N:GEN existence:N:P3S:COND,
    text:N:ACC write:V:INF:COP.
```

The program writes a file, reads it back, and branches on the result. In Haskell:[4]

Haskell

```haskell
run = do
  writeFile "./text.tmp" "hello kip"
  result <- readFile "./text.tmp"
  case result of
    Nothing  -> pure ()
    Just text -> putStrLn text
```

Note that `writeFile` takes two `String` arguments in a fixed order that the programmer must memorize. In the Kip version, the dative suffix *-ye* ("to") marks the destination and the accusative *-i* marks the content, making each argument's role explicit regardless of order.

The examples in this section illustrate the core surface mechanisms—case-marked arguments, verbal mood for effects, pattern matching, and partial application—all of which rely on morphological information that a conventional language would discard after parsing. The next section examines what happens when that morphological information is ambiguous.

3 MORPHOLOGICAL AMBIGUITY AS TYPE-DIRECTED CHOICE

Having introduced Kip's surface syntax through examples, we now turn to the question of how morphological ambiguity interacts with type checking. Rather than committing to a single morphological analysis at parse time, Kip defers that commitment to type checking, where richer context is available.

[4]Haskell snippet here uses an idealized interface to keep the correspondence with Kip clear.

To see why this matters, consider a concrete example. Turkish morphology produces genuine lexical ambiguity: a single surface form can correspond to distinct root+suffix analyses. The form *kümesi*, for instance, may be read as:

(1) *küme-si*: root *küme* ("set"), plus third-person possessive (:P3S), roughly "its set" / "the set of it".
(2) *kümes-i*: root *kümes* ("chicken coop"), plus accusative (:ACC) or possessive-marked nominal form depending on context, yielding a different lexical base and a potentially different role in the sentence.

Whether a program is manipulating sets or chicken coops is a distinction best not left to parser intuition; Kip therefore treats this as a typed disambiguation problem.

The resolution process has two phases. In the first phase, the parser queries TRmorph, a finite-state Turkish morphological analyzer/generator [27, 28], and builds an unranked candidate set for each surface token (§3.1). In the second phase, the type checker narrows each set to a unique reading using scope, case-role compatibility, and type constraints (§3.3). When the checker cannot resolve a candidate set automatically, the programmer can force a reading with apostrophes (e.g., `küme'si`). In effect, morphological ambiguity becomes a disciplined form of overloading whose resolution is context-dependent and type-checked.

3.1 Parser-Side Candidate Construction

Given a surface token t and the current scope Γ, the parser computes a candidate set $C(t)$ as follows:

(1) **Scope lookup.** If t matches a unique name in Γ whose case is unambiguous, return $C(t) = \{t\}$ immediately.
(2) **Morphological analysis.** Query the morphological analyzer for all analyses of t (and, where applicable, a copula-stripped variant). Let $A(t)$ be the resulting set of (root, case, part-of-speech) triples.
(3) **Heuristic completion.** Extend $A(t)$ with systematically predictable case readings that the analyzer's lexicon may lack (e.g., :P3S/:ACC alternations on nominal stems).
(4) **Scope filtering.** Remove from $A(t)$ every analysis whose root is not visible in Γ. If multiple analyses survive, retain all of them.
(5) **Deferral.** Store the surviving candidate set in the AST node for t and hand it to elaboration.

The output is an unranked set: the parser makes no commitment among surviving candidates. Ambiguity that remains after scope filtering is resolved during type checking, where case-role compatibility and typing constraints prune the set further. This "defer then prune" policy lets later phases exploit both grammatical and typing evidence instead of locking in a reading too early.

3.2 Unknown Surface Forms

Verbification, especially for borrowed words from foreign languages such as English, is very common in Turkish. For instance, *googling* is translated to *googlelamak*, *mailing* is translated to *maillemek*, an arbitrary foreign word such as the English placeholder *foo* can be extended with the suffix *-la* to turn the stem into a verb *foola-* ("to do/apply foo"), the infinitive suffix *-mak* produces the citation form *foolamak* ("to foo"), and the converb suffix *-(y)ıp* produces a sequential form *foolayıp* ("having fooed, ..."). The current morphological analyzer uses a finite lexicon. When a surface form cannot be analyzed—because its root is absent from that lexicon—Kip reports unknown-word diagnostics, often with suggestions. In principle, the lexicon could be extended by adding new stems and rebuilding the analyzer's transducer artifacts, but the current implementation does not provide a user-facing mechanism for this. The limitation is one of lexical coverage, not of the type system or grammar: the parser and type checker require no changes.

3.3 Type-Directed Resolution

Given a parsed AST whose nodes carry candidate sets $C(t)$ from §3.1, the type checker resolves each node to a unique reading as follows:

(1) **Name resolution.** For each candidate set $C(t)$, remove analyses whose root is not in scope. If $C(t)$ becomes empty, report an unknown-name error.
(2) **Overload selection.** At each call site $f\{e_1\kappa'_1, \ldots, e_n\kappa'_n\}$, filter candidate signatures by arity, then by case compatibility, then by type compatibility. Exact arity matches take priority over partial matches. If argument types are not yet known, defer the choice.
(3) **Role recovery.** For surviving signatures, align the supplied argument cases κ'_i against the signature's declared cases κ_i. If the alignment is unique, reorder arguments into signature order; otherwise retain positional order within repeated-case segments.
(4) **Effect checking.** Verify that no effectful expression appears in a context whose mode is pure.
(5) **Exhaustiveness.** Check that every function definition and match expression covers all values of the scrutinee type.
(6) **Commitment.** If every candidate set has been reduced to a single reading, type checking succeeds and the AST is fully resolved. Otherwise, report the first ambiguity or type mismatch.

After this pass, every surface form has a unique morphological reading, every call has a unique overload with arguments in signature order, and no effectful expression appears in a pure context.

3.4 Head-Final Syntax and Parenthesization

Because Turkish is a head-final language, Kip's syntax places the function name—the head—after its arguments. In pure definitions, this follows noun-phrase structure (e.g., `5'in karesi`: "the square of 5"); in effectful calls, the SOV clause structure likewise puts the verbal head last (e.g., `selamla`: "greet!"). In both cases, arguments precede their head, so the resulting expressions have a *reverse Polish notation* (RPN)-like character.

RPN is well-known for being unambiguous without parentheses when every operator has a fixed arity [9]. If Kip's function names each had a single, fixed arity, the same property would hold here and explicit parentheses would be unnecessary. However, Kip permits overloading: the same surface name can be defined at multiple types and, critically, at different arities. The parser therefore cannot determine how many arguments a given head consumes from the name alone. This arity ambiguity can cause structural attachment ambiguity (analogous to *[American history] teacher* vs. *American [history teacher]* in English)—it is not a morphological problem but a grouping problem, and the standard remedy is the same as in arithmetic: explicit parentheses delineate constituent structure. In the current implementation, the user-facing parenthesis-free ambiguity check is explicit in the REPL path for overloaded-head chains; ordinary file parsing remains deterministic and relies on later elaboration/type checking to reject incompatible readings.

Consider this expression computing the product of two differences:

Surface
```
4'le 1'in farkıyla 5'le 3'ün farkının çarpımı
```

Morphological Gloss
```
4:INS 1:GEN difference:N:P3S:INS
5:INS 3:GEN difference:N:P3S:GEN product:N:P3S
```

Because each function here has a unique arity, the token stream can be parsed unambiguously without parentheses, exactly as in RPN.

However, the programmer may write the parenthesized form to make grouping explicit:

Surface
```
(4'le 1'in farkıyla)
(5'le 3'ün farkının) çarpımı
```

Morphological Gloss
```
(4:INS 1:GEN difference:N:P3S:INS)
(5:INS 3:GEN difference:N:P3S:GEN) product:N:P3S
```

This is the same expression with explicit grouping.

In practice, parentheses are required only where overloading introduces genuine arity uncertainty. When a name has a unique definition, or all overloads share the same arity, Kip can resolve grouping without parentheses, preserving the concise, RPN-like reading that head-final syntax affords.

3.5 Overloading and Context Selection

Besides morphological disambiguation, Kip supports ordinary type-driven overloading: the same head noun can be defined at multiple types.

Surface
```
(bu tam-sayıyla) (şu tam-sayının) birleşimi,
  (bu tam-sayıyla) (şu tam-sayının) toplamıdır.

(bu doğrulukla) (şu doğruluğun) birleşimi,
  bu doğruysa, doğru;
  değilse, şudur.
```

Morphological Gloss
```
(this:N:NOM integer:N:INS)
(that:N:NOM integer:N:GEN) union:N:P3S,
  (this:N:INS integer:N:INS)
  (that:N:GEN integer:N:GEN) sum:N:P3S:COP.

(this:N:NOM boolean:N:INS)
(that:N:NOM boolean:N:GEN) union:N:P3S,
  this:N:NOM true:N:COND, true:N:NOM;
  otherwise, that:N:NOM:COP.
```

Here the same surface function name is defined at integer and boolean types with different branch behavior. The closest Haskell analogue is a typeclass method with separate instances. Kip resolves the overload at each call site by case and type context rather than by typeclass dispatch. The call site selects the intended overload by type context:

Surface
```
(5'le 2'nin birleşimini) yaz.
(doğruyla yanlışın birleşimini) yaz.
```

Morphological Gloss
```
(5:INS 2:GEN union:N:P3S:ACC) write:V:IMP.
(true:INS false:GEN union:N:P3S:ACC) write:V:IMP.
```

Both calls invoke the same surface name; argument types select the intended interpretation.

This is type-directed overload resolution at call sites.

4 DESIGN TRADE-OFFS AND LIMITATIONS

Kip's approach to role assignment in function application—using grammatical case—is one of several possible strategies. This section examines the trade-offs behind that choice and several related design decisions: when to resolve morphological ambiguity (§4.2), how to enforce a purity boundary without an explicit effect language (§4.3), why partial application can replace most uses of lambdas (§4.4), and where the gap between Kip and ordinary Turkish creates friction (§4.5). We close with implementation status and current limitations.

4.1 Argument Role Strategies

The simplest alternative is purely positional role assignment, where argument roles are determined entirely by order. This yields simple parsing and predictable elaboration, but it weakens one of Kip's core linguistic affordances: Turkish-like reordering for emphasis and local readability. It also makes morphological case mostly ornamental, since role information is already fixed by position. More subtly, it suppresses focus-sensitive alternatives in Kip interfaces. In Kip's polymorphic set interface, membership is defined as:

Surface

```
(bu öğe kümesine) (şu öğenin)
  (üyelik doğruluğu),
  ...
```

Morphological Gloss

```
(this:N:NOM element:N:NOM set:N:P3S:DAT)
(that:N:NOM element:N:GEN)
  (membership:N:NOM boolean:N:P3S),
  ...
```

At call sites, the same query can be phrased with different emphasis:

Surface

```
(bir-kümeye bir-öğenin üyeliği)
(bir-öğenin bir-kümeye üyeliği)
```

Morphological Gloss

```
(a-set:DAT an-element:GEN membership:N:P3S)
(an-element:GEN a-set:DAT membership:N:P3S)
```

Both forms compute the same membership query.

The first foregrounds the set, while the second foregrounds the element. A strictly positional interface would force one canonical wording and erase this distinction.

Another option is to keep fixed function names but attach explicit role labels (keyword arguments or preposition-like markers) to arguments. This improves clarity and avoids some ambiguity, but introduces a parallel role language separate from ordinary case morphology. For Kip, that would reduce the central claim that natural inflection itself can carry role information through static checking.

Kip instead uses grammatical case as the primary cue for role assignment in function and constructor interfaces. At call sites, arguments may be reordered when cases and types jointly recover a unique assignment. This aligns with Turkish morphosyntax and makes case markings computationally relevant rather than merely decorative. The trade-off is increased checker complexity, since role recovery, overload selection, and ambiguity handling become coupled—but studying that coupling is precisely the research goal.

In practice, reordering is most useful when argument roles are distinct and semantically salient (e.g., source vs. destination, accumulator vs. current value). It is less useful when multiple parameters share the same case or when role distinctions are already obvious from local context. Accordingly, Kip's implementation keeps conservative matching heuristics and falls back to more positional behavior in repeated-case situations.

4.2 Early vs. Late Disambiguation

Independent of argument style, language designers must choose when morphological ambiguity is resolved: at lex/parse time (early commitment, simpler later phases) or at elaboration/type-check time (deferred commitment, richer context). Kip chooses deferred commitment. This enables type-directed disambiguation but requires diagnostics that explain why one reading survives and another fails. From a localization perspective, this is a structural choice rather than a surface-lexicon choice [25].

4.3 Purity Boundary Design

Kip also chooses a syntactic purity boundary (noun-phrase-like pure definitions vs. infinitive/imperative effectful forms) rather than an explicit effect annotation language in source syntax. This choice follows the language's broader head-final tendencies: pure code appears in nominal structures, while effectful code appears in verbal ones.

There is also a semantic reason for this split. Haskell's strong separation between pure expressions and effectful computations is often justified by the fact that lazy evaluation makes timing of effects difficult to reason about unless effects are isolated [19]. Kip faces a related problem for a different reason: case-marked role recovery allows call-site order to diverge from interface order. In a language that allowed arbitrary effectful arguments in such calls, users would immediately face a sequencing question: should those effects occur in the order written at the call site, or in the order induced by the original function interface after elaboration?

OCaml provides a cautionary example. Its labeled arguments [7] decouple call-site order from parameter order, but argument evaluation still follows the language's ordinary evaluation strategy rather than the visible order at the call site. For example:

```
# let f ~x ~y = x - y;;
val f : x:int -> y:int -> int = <fun>
# f ~x:(printf "x"; 5) ~y:(printf "y"; 3);;
yx- : int = 2
# f ~y:(printf "y"; 3) ~x:(printf "x"; 5);;
yx- : int = 2
```

The observable evaluation order does not match the surface order of the labeled arguments. Partial application makes this even less intuitive:

```
# f ~y:(printf "y"; 3) ~x:(printf "x"; 5);;
yx- : int = 2
# (f ~y:(printf "y"; 3)) ~x:(printf "x"; 5);;
xy- : int = 2
```

The result value is the same, but the effect order changes once one argument is supplied earlier than the other. Kip avoids this class of confusion by pairing its flexible nominal application syntax with a pure/effectful distinction. Case-directed reordering and partial application operate over pure arguments, for which evaluation order is observationally irrelevant. Effectful computation remains in verbal syntax, where sequencing is explicit. Non-positional application is safe precisely because it is restricted to pure expressions, where evaluation order does not affect the result.

4.4 Partial Application Instead of Lambdas

In a positional language, passing a multi-argument function to a higher-order combinator like `map` typically requires an explicit lambda to route the mapped element to the correct parameter position. Case-labeled arguments eliminate this problem: because each argument is identified by its case role rather than its position, the programmer can supply any subset of arguments and the compiler determines which roles remain open.

Consider a three-argument clamping function that takes a target integer (Genitive, *-in*), a lower bound (Ablative, *-den*), and an upper bound (Dative, *-e*). To clamp every element of `[1, 2, 3, 4]` between 0 and 2, the programmer simply supplies the two bounds:

Surface

```
(0'dan 2'ye kısıtlanmışıyla)
  [1, 2, 3, 4]'ün eşlemi
```

Morphological Gloss

```
(0:ABL 2:DAT clamp:N:P3S:INS)
  [1, 2, 3, 4]:GEN map:N:P3S
```

The Ablative and Dative arguments are supplied, leaving the Genitive role open. The partially applied function then has exactly the type that `map` expects. No lambda, no argument reordering, no adapter function.

This works precisely because case labels make partial application position-independent. The closest analogue in Haskell is operator sections: `(+ 1)` and `(2 *)` partially apply a binary operator by filling in one argument. But sections only work for infix operators and only for two arguments. Kip's case-directed partial application generalizes this to any function, any arity, and any subset of arguments—the programmer supplies whichever roles are known and the compiler infers the rest from the signature. The result is that most uses of anonymous functions in conventional functional code—sectioning, argument reordering, filling in fixed parameters—are covered by ordinary partial application.

Kip does not currently support explicit lambdas. A natural-language lambda would require converb chains ("take … and give") that are considerably more verbose than the partial-application form above, and a symbolic λ notation would break the linguistic surface design. Since case-directed partial application handles the common cases, the trade-off favors omitting lambdas for now.

4.5 Uncanny Valley between Kip and Turkish

Kip's Turkish surface syntax invites speakers to rely on native grammatical expectations, which creates a distinctive failure mode: users may write perfectly valid Turkish that is nevertheless invalid Kip. For example, if the standard library defines subtraction (`fark`) as "subtraction of a and b" (`a'yla b'nin farkı`), a Turkish speaker may naturally attempt the equally grammatical "subtraction of a from b" (`a'nın b'den farkı`).

This is not an idiosyncratic choice but a systematic *case-frame* alternation in Turkish. Kip's stricter grammar turns that flexibility into parse errors. The resulting experience has an "uncanny valley" quality in language design: the text looks close enough to ordinary Turkish to invite fluent interpretation, yet the system rejects it because of small rule-level mismatches [17].

A practical response can be to embrace these predictable alternations by defining standard-library synonyms (and, where necessary, explicit argument-role mappings), so that common Turkish paraphrases are accepted as equivalent usage modes rather than treated as errors.

Linguistically speaking, Kip's case-label vocabulary deliberately flattens a heterogeneous set of suffixes into a uniform notion of "case." In Turkish linguistics, not all of these suffixes are best described as cases in the same sense. Some are syntactic (nominative, accusative), assigned by structural position rather than by semantic role; some are semantic (instrumental, dative, ablative), carrying transparent role information such as instrument or goal; and some are lexically idiosyncratic, selected by particular verb stems regardless of meaning [26]. The instrumental suffix *-(y)la*, for instance, can function as a semantic case marker ("with a knife"), but it also serves as a conjunction ("Deniz and I")—a distinction that Kip does not currently represent. By treating all of these uniformly as case labels κ, Kip gains a simple, uniform alignment mechanism at the cost of erasing distinctions that a linguistically richer treatment would preserve. In practice, the design works best for semantic cases, where the suffix genuinely signals an argument role, and is weaker for structural cases, where the suffix is determined by syntactic position rather than by the kind of role information that Kip's elaboration exploits.

4.6 Evaluation Strategy and Implementation

Once type checking succeeds, the disambiguated program is a fully resolved AST: overloads have been selected, arguments have been reordered into signature order, and morphological candidate sets have been reduced to single readings. No separate core language or elaborated intermediate representation is produced; the type checker modifies the parsed AST in place. The rest of this section describes the resulting runtime architecture and then the current prototype status.

The primary backend is a tree-walking interpreter. To avoid stack overflow on deeply recursive programs, the evaluator uses a *trampoline*: expressions in tail position return a continuation rather than a value, and a driver loop iterates these continuations without growing the Haskell call stack. Effectful primitives (console I/O, file operations, random number generation) are implemented as monadic operations in the interpreter's state monad, which threads an environment of value bindings, function clause families, constructor signatures, and primitive implementations.

An alternative backend emits JavaScript (targeting Node.js or browser ESM). This backend lowers algebraic data types to tagged objects, compiles pattern matching to conditional chains, and translates effectful sequencing to **`await`** expressions. Both backends consume the same post-type-checking AST, so all case-directed reordering, overload selection, and purity enforcement are resolved before code generation begins.

Implementation. Kip is implemented in approximately 16,000 lines of Haskell, plus a small C wrapper for morphological analysis. Parsing uses the Megaparsec combinator library; morphological analysis is provided by TRmorph [27], a finite-state Turkish analyzer built on the Foma transducer toolkit [11], accessed via Haskell's foreign-function interface [19]. The type checker, evaluator, and a JavaScript code-generation backend all operate on a single annotated AST rather than a separate intermediate representation. An LSP server provides editor integration (hover, go-to-definition, and diagnostics), and a Haskeline-based REPL supports interactive exploration with step-by-step evaluation traces. A binary caching layer (keyed by SHA-256 hashes of source files) allows the parser and type checker to be skipped on unchanged modules. The standard library, test suites, and build infrastructure are included in the artifact repository. The current artifact's positive suite (`tests/succeed`) contains 100 passing programs (1178 lines of Kip code) spanning:

(1) Case-structured arithmetic and interface use, including reordered application when case/type evidence is sufficient.
(2) Polymorphic algebraic data types and pattern matching (e.g., pairs, sums/options, days, trees), with nested and multi-line matches.
(3) Higher-order list programming (map, filter, and fold) and explicit type ascription forms.
(4) Effectful programs in verbal syntax, including stdin/stdout interaction, environment-variable access, file I/O, and command-line argument handling.
(5) Larger end-to-end code such as a 262-line DPLL SAT solver that parses CNF text, performs unit propagation/pure-literal elimination/branching, and reports `SAT`/`UNSAT`.

4.7 Limits and Non-Goals

Promoting morphology from presentation layer to type-checking signal introduces real costs. Diagnostic design becomes central, because users must understand both type and morphological mismatches simultaneously when a program fails. Unrestricted case flexibility can make overload behavior difficult to predict, requiring conservative matching heuristics in practice. And the design transfers most directly to languages with overt role marking; the usability gains depend on the programmer's fluency in the target language's inflectional (or declensional) system.

Kip does not aim to be a full formalization of Turkish grammar, nor a free-form natural-language programming interface. Its surface language is deliberately constrained: linguistic material is kept where it improves compositional parsing and static checking, and set aside where it would reduce predictability. Kip also does not currently target principal global disambiguation across all possible parses and overloads; the implementation uses typed elaboration with local, deterministic resolution steps. Finally, the current effect discipline is intentionally lightweight—it enforces a practical purity boundary in syntax, but it is not a full effect typing framework with polymorphic effect inference.

5 FORMALIZATION

This section summarizes a Rocq mechanization[5] of an explicitly case-labeled, overloaded core of Kip covering case-aware interfaces, overloaded application, pattern matching, the purity boundary, small-step semantics with case alignment, and machine-checked proofs of effect rejection, progress,

[5]The development comprises seven files under the `KipCore` namespace: `Syntax.v`, `Static.v`, `StaticFacts.v`, `Dynamic.v`, `DynamicFacts.v`, `Soundness.v`, and `Elaboration.v`.

and preservation. The core language includes pure let-binding, floating-point literals, general pattern matching, and semantic exhaustiveness conditions for functions and matches. A separate elaboration layer formalizes call-site overload resolution from an abstract morphology oracle and already elaborated, already typed arguments. The development does not model the full parser, full-expression elaboration, or the implementation's end-to-end heuristic search strategy.

5.1 Core Calculus

The core language (Figure 1) makes case labels explicit on type arguments, signature parameters, function and constructor arguments, and constructor subpatterns.

κ	$::=$	$\text{:NOM} \mid \text{:ACC} \mid \text{:DAT} \mid \text{:LOC} \mid \text{:ABL} \mid \text{:GEN} \mid \text{:INS} \mid \text{:COND} \mid \text{:P3S}$	*grammatical case labels*
κ_r	$::=$	$\text{:NOM} \mid \text{:P3S}$	*return-case labels*
κ_{res}	$::=$	$\varnothing \mid \kappa_r$	*result-case marker*
μ	$::=$	$\mathsf{pure} \mid \mathsf{eff}$	*effect modes*
c	$::=$	$n \mid fl \mid s \mid ch$	*integer, float, string, and character literals*
τ	$::=$	$\alpha \mid \mathsf{Int} \mid \mathsf{Float} \mid \mathsf{String} \mid \mathsf{Char} \mid D\{\tau_1\kappa_1, \ldots, \tau_m\kappa_m\} \mid \tau_1 \to \tau_2$	*types*
υ	$::=$	(τ, κ_{res})	*result types*
σ	$::=$	$\forall\overline{\alpha}.\ [\tau_1\kappa_1, \ldots, \tau_n\kappa_n] \Rightarrow \tau_r\kappa_r$	*signature*
p	$::=$	$\_ \mid x \mid c \mid C\{p_1\kappa_1, \ldots, p_n\kappa_n\}$	*patterns*
e	$::=$	$x \mid c \mid (e : \tau)$	*variables, literals, ascription*
	$\mid$	$f\{e_1\kappa_1, \ldots, e_n\kappa_n\} \mid C\{e_1\kappa_1, \ldots, e_n\kappa_n\}$	*case-labeled application*
	$\mid$	$\mathsf{match}\ e\ \mathsf{with}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$	*match*
	$\mid$	$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	*pure local binding*
	$\mid$	$e_1; e_2 \mid x \leftarrow e_1; e_2$	*effect sequencing/binding*
cl	$::=$	$f\{p_1\kappa_1, \ldots, p_n\kappa_n\} = e$	*function clause*
ϕ	$::=$	$[cl_1, \ldots, cl_m]$	*function definition family*
δ	$::=$	$\mathsf{data}\ D\{\alpha_1\kappa_1, \ldots, \alpha_m\kappa_m\}\ \mathsf{where}\ C_1 : \sigma_1 \mid \cdots \mid C_k : \sigma_k$	*ADT declaration*
P	$::=$	$\delta_1; \cdots; \delta_m; \phi_1; \cdots; \phi_\ell; e$	*programs*

Fig. 1. Syntax of the core calculus.

Signatures declare a return-case label κ_r: either :NOM for bare nouns such as nullary constructors (`boş`, `doğru`), or :P3S for possessive noun phrases such as `toplamı` ("the sum of") and `eki` ("the cons of"). The typing judgment, however, must also handle expressions that do not originate from a declared signature—variables, literals, matches, let-bindings, and partial applications—which have no declared return case. The result-case marker κ_{res} extends κ_r with $\varnothing$ for these cases. The typing judgment pairs an ordinary type with this marker, yielding a *result type* $\upsilon = (\tau, \kappa_{res})$. Signatures always declare a return case—κ_r is never empty—but the typing context Γ stores only plain types, so variables and pattern-bound names have no return case to report; $\varnothing$ fills that role. Elaboration uses κ_{res} to reconstruct how surface morphology decomposes, but soundness does not depend on it: preservation erases κ_{res} and tracks only the underlying type τ.

Turning to how labeled arguments are processed, the key auxiliary operation is

$$\text{AlignByCases}([\kappa_1, \ldots, \kappa_n], [a_1\kappa'_1, \ldots, a_m\kappa'_m]).$$

We call the element paired with a case label (i.e. each a_i above) a *payload*. This operation repeatedly selects the first remaining payload with the requested case label. To illustrate, recall the sum function from §1, whose signature expects two parameters in the order [:INS, :GEN]. A caller may write the arguments in the opposite order:

Surface	Morphological Gloss
`1'in 3'le toplamı`	1:GEN 3:INS sum:N:P3S

Here the written argument list is [1:GEN, 3:INS], and alignment against [:INS, :GEN] yields [3, 1]. Distinct labels therefore admit permutation,[6] while repeated labels remain sensitive to written order. The same alignment discipline is used for full function application, constructor application, constructor patterns, ADT type arguments, and function clause patterns.

5.2 Static Semantics

The judgments are parameterized by several contexts: a datatype declaration context Σ_d, an effectfulness context Σ_e recording which function names are effectful, a function signature context Σ_f, a constructor signature context Σ_c, a type-variable context Δ, a variable typing context Γ, and an effect mode μ. The main static judgments are:

$$\begin{array}{rl} \Sigma_d; \Delta \vdash \tau \text{ ok} & \text{type } \tau \text{ is well formed} \\ \Sigma_d; \Delta \vdash \sigma \text{ ok} & \text{signature } \sigma \text{ is well formed} \\ \Sigma_d \vdash \delta \text{ ok} & \text{ADT declaration } \delta \text{ is well formed} \\ \Sigma_e; \Sigma_f; \Sigma_c; \Gamma; \mu \vdash e : v & \text{expression } e \text{ has result type } v \end{array}$$

The compatibility predicate used by ascription is just equality in the current development.

Well-formedness. The well-formedness judgments $\Sigma_d; \Delta \vdash \tau$ ok, $\Sigma_d; \Delta \vdash \sigma$ ok, and $\Sigma_d \vdash \delta$ ok check types, signatures, and ADT declarations, respectively. Primitive types and arrows are standard. An ADT type $D\{\tau_1\kappa_1, \ldots, \tau_m\kappa_m\}$ is well formed when its labeled arguments align with the declared case labels of D and the aligned payload types are themselves well formed. Because ADT declarations require distinct parameter case labels, writing those arguments in a different order is harmless.

Signatures check their parameter types and return type in a context extended with the quantified type variables. ADT declarations require distinct constructor names, distinct case labels on the ADT parameters, well-formed constructor signatures, and constructor return types of the form

$$D\{\tau_1\kappa_1, \ldots, \tau_m\kappa_m\}$$

whose labeled arguments align with the declaration's parameter cases. This means constructor return types may write the ADT's case-labeled type arguments in any aligned order.

To support soundness for general constructor-pattern matching, the mechanization also imposes a regularity condition on constructor signatures. Informally, the result type of a constructor must determine the instantiated payload types: there are no hidden constructor-only type variables affecting pattern matching. The final preservation theorem assumes this regularity invariant for all constructors in Σ_c.

Typing. The central rule is full application (Figure 2), where θ is a substitution instantiating the quantified type variables $\overline{\alpha}$, with the side condition $\Sigma_e(f) = \text{true} \implies \mu = \text{eff}$.[7]

In surface Kip, this is the purity boundary described in §2.5: nominal forms such as `toplamı` (`sum`:N:P3S, "the sum of") define pure functions and are never recorded in Σ_e, while verbal forms such as `yazmak` (`write`:V:INF, "to write") are effectful and require $\mu = \text{eff}$ at every call site. Constructor application is identical except that the signature comes from Σ_c. Partial application identifies which

[6] `align_cases_Permutation` in `StaticFacts.v`.
[7] Effect rejection is proved as `effect_rejection_sound` in `StaticFacts.v`.

$$\dfrac{\begin{array}{c}\forall\overline{\alpha}.[\tau_1\kappa_1,\ldots,\tau_n\kappa_n] \Rightarrow \tau_r\kappa_r \in \Sigma_f(f)\\ \text{AlignByCases}([\kappa_1,\ldots,\kappa_n],[e_1\kappa'_1,\ldots,e_n\kappa'_n]) = [e^\star_1,\ldots,e^\star_n]\\ \forall i\in[n].\ \Sigma_e;\Sigma_f;\Sigma_c;\Gamma;\mu\vdash e^\star_i : (\theta(\tau_i),\kappa_{res,i})\end{array}}{\Sigma_e;\Sigma_f;\Sigma_c;\Gamma;\mu\vdash f\{e_1\kappa'_1,\ldots,e_n\kappa'_n\} : (\theta(\tau_r),\kappa_r)}\ \text{T-App}$$

Fig. 2. Full application typing rule.

signature positions have already been filled and returns an arrow over the remaining parameter types in signature order.

Pattern binding mirrors application. For constructor patterns, the constructor signature is instantiated, the subpatterns are aligned by case, and each aligned subpattern is checked against the corresponding parameter type. The resulting binding contexts are concatenated. For example, in the list-length function from §2.4, the pattern `ilkin devama ekiyse` (`head:N:GEN rest:N:DAT addition:N:P3S:COND`) destructures a cons cell. The constructor `ek` expects [:GEN, :DAT], so the subpatterns *`ilk`*:GEN and *`devam`*:DAT are aligned against that signature exactly as function arguments would be. Since the two cases are distinct, writing `devama ilkin ekiyse` (`rest:N:DAT head:N:GEN addition:N:P3S:COND`) would produce the same binding.

Match typing uses an exhaustiveness condition: every $v \in \mathcal{V}[\![\tau]\!]$ (the semantic-value relation defined in §5.3) must match some branch. Function-definition families are checked the same way. A family chooses one declared signature, aligns every clause's labeled patterns against that signature, records the resulting pattern-binding obligations, types each clause body, and requires the same semantic exhaustiveness condition for the instantiated signature.

5.3 Dynamic Semantics

For soundness, the development uses an erased typing judgment

$$\Sigma_e;\Sigma_f;\Sigma_c;\Gamma;\mu\vdash_{plain} e:\tau \iff \exists\kappa_{res}.\ \Sigma_e;\Sigma_f;\Sigma_c;\Gamma;\mu\vdash e:(\tau,\kappa_{res}).$$

Preservation tracks only the plain type, not the result-case marker.

Evaluation is parameterized by a dynamic function-entry context

$$F : f \mapsto [entry_1,\ldots,entry_k]$$

whose entries package a monomorphic signature together with its clause family. Two resolution judgments look up the function name in F and return a matching entry. Full resolution $F \vdash f\{\overline{e\kappa}\} \hookrightarrow_{\text{full}} entry$ applies when all parameter positions are supplied; partial resolution $F \vdash f\{\overline{e\kappa}\} \hookrightarrow_{\text{partial}} entry$ applies when only a strict subset is supplied. (We write the hook arrow $\hookrightarrow$ to distinguish these lookups from the step relation $\longrightarrow$ introduced below.) Both judgments must select a unique entry for the call. The well-formedness invariant $\Sigma_e;\Sigma_f;\Sigma_c \vdash F$ ok ties F to the static contexts: every stored entry has a declared monomorphic signature, its clauses satisfy the local clause-typing invariant needed by beta-reduction, and those clauses are semantically exhaustive in every compatible mode; conversely, every declared signature has a dynamic entry; and full matches are unique, partial matches are unique, with a full match ruling out a simultaneous partial one.

Values are literals, fully applied constructor applications whose arguments are all values, and partial applications whose supplied arguments are already values. In addition, the development defines a pattern-matching judgment $p \vdash_{\Sigma_c} v \rightsquigarrow \rho$, which holds when pattern p matches value v and produces substitution ρ, and a semantic-value relation $v \in \mathcal{V}[\![\tau]\!]$, which classifies the closed values used in exhaustiveness statements. Progress uses a lemma showing that every ordinary value of a plain type is also a semantic value of that type.[8]

The small-step semantics is given by three mutually inductive relations:

$$\overline{e} \longrightarrow_{F,\Sigma_c} \overline{e}' \qquad \overline{e\kappa} \longrightarrow_{F,\Sigma_c} \overline{e'\kappa} \qquad e \longrightarrow_{F,\Sigma_c} e'.$$

The list relations step one payload inside unlabeled or labeled lists; the expression relation uses them to evaluate payloads from left to right after case alignment.

We say a labeled argument list is *canonical* with respect to a signature σ when every argument is a value and the labels appear in the parameter order declared by σ. Function and constructor calls reduce in phases that produce a canonical argument list and then consume it:

(1) canonicalize the written labeled argument list into parameter order by applying AlignByCases,
(2) step argument payloads left to right,
(3) for fully saturated function calls, beta-reduce against a semantically matching clause.

Constructors have the same canonicalization and payload-stepping phases as functions, but no beta rule. Matches step the scrutinee until it is a value, then reduce by selecting a branch whose pattern matches that value (i.e. $p \vdash_{\Sigma_c} v \rightsquigarrow \rho$ holds). Pure let first evaluates its bound expression and then substitutes the resulting value into the body.

The characteristic function-call beta rule is shown in Figure 3. The resolution premise looks up the function name in the dynamic context

$$\frac{F \vdash f\{\overline{v\kappa}\} \hookrightarrow_{\mathsf{full}} (\sigma, [f_1, \ldots, f_n]) \qquad f_i = f\{\overline{p\kappa}\} = e_{body} \qquad \mathsf{canonical}(\sigma, \overline{v\kappa}) \qquad \overline{p} \vdash_{\Sigma_c,\sigma} \overline{v\kappa} \rightsquigarrow \rho}{f\{\overline{v\kappa}\} \longrightarrow_{F,\Sigma_c} \rho(e_{body})}\ \textsc{ST-App-Beta}$$

Fig. 3. Function-call beta reduction.

The resolution premise looks up the function name in the dynamic context F and returns a monomorphic signature σ together with the clause family $[f_1, \ldots, f_n]$. The premise $\mathsf{canonical}(\sigma, \overline{v\kappa})$ asserts that the arguments are canonical with respect to σ. The rule then picks a clause f_i, destructures it into patterns $\overline{p}$ and body e_{body}, matches the value arguments against the patterns to produce a substitution ρ, and reduces to $\rho(e_{body})$.

5.4 Elaboration of Calls

The elaboration layer isolates the compile-time overload selection performed by Kip's type checker. Its input is not raw source syntax, but a smaller interface tailored to the overload-resolution phase:

- an abstract morphology oracle mapping a surface string to possible base names and case information;
- a surface call head together with arguments that already carry an elaborated core expression and its inferred type;
- the declarative function-signature context from the core formalization.

A SurfaceCall packages these three components together. From it, the elaboration development enumerates candidate function names, candidate argument case-labelings, and then exact or partial overload choices. Exact choices are prioritized over partial ones, and the top-level classifier

$$\mathrm{ElaborateCall} : \mathrm{SurfaceCall} \rightarrow \{\mathrm{NoMatch}, \mathrm{Unique}(\mathit{choice}), \mathrm{Ambiguous}\}$$

[8] `value_has_semantic_value` in `Soundness.v`.

never picks an arbitrary first match: ambiguity is reported explicitly.

The key algorithmic theorems show that the computed exact, partial, and prioritized choice lists are sound and complete with respect to relational specifications of overload matching. From this, the mechanization proves that elaboration is deterministic up to explicit ambiguity, that unique results are sound, and that elaboration is stable under irrelevant context growth. These results, along with theorems connecting elaboration back to the core formalization, are listed in §5.5. Unique exact elaboration yields a prioritized semantic match, a core full-application term, and a plain typing derivation for it. Unique partial elaboration yields the analogous prioritized partial match, partial-application candidate, and residual arrow type. Under $\Sigma_e; \Sigma_f; \Sigma_c \vdash F$ ok, unique exact elaboration also agrees with the core dynamic resolution judgment: the overload selected statically is exactly the one selected by the declarative operational semantics.

There is an important modeling distinction here. The core language still represents calls as unresolved application nodes, so those theorems are coherence results between compile-time elaboration and an overloaded core semantics. To capture the implementation's intuition more directly, the elaboration layer also defines a small resolved call-head layer whose states store the chosen dynamic entry explicitly. Unique exact and partial elaboration results produce such resolved call heads, they erase to the corresponding core application term, they are well typed, and their head-local execution relation performs no overload search. A separate erasure theorem shows that every resolved call-head step corresponds to an ordinary core step after forgetting the stored entry.

This completes the three layers of the mechanization: a core calculus with case-aligned application and small-step semantics (§5.1–§5.3), and an elaboration layer that connects compile-time overload resolution to the core's declarative semantics (§5.4).

5.5 Mechanized Results

We now state the top-level theorems that these layers support. Each result is machine-checked in Rocq; we give the relevant lemma names and source files for reference.

- **Effect rejection.** Any effectful expression is untypeable in pure mode (`effect_safety` and `effect_rejection_sound` in `StaticFacts.v`).
- **Algorithmic overload elaboration.** The elaboration layer proves that compile-time call elaboration is a correct decision procedure for prioritized overload resolution: it is deterministic up to explicit ambiguity (`elaborate_call_deterministic_or_ambiguous`), sound (`elaborate_call_overload_sound`), unique when it returns a unique answer (`elaborate_call_overload_unique`), all in `Elaboration.v`.
- **Elaboration produces well-typed core terms.** When elaboration returns a unique exact choice, the resulting core application term has a plain typing derivation (`elaborate_call_unique_exact_correct`); when it returns a unique partial choice, the resulting partial application is well typed with the expected residual arrow type (`elaborate_call_unique_partial_correct`), both in `Elaboration.v`.
- **Compile-time resolved call heads.** Unique elaboration also produces resolved call-head states (`elaborate_call_unique_exact_produces_resolved_call` and `elaborate_call_unique_partial_produces_resolved_call`) whose head-local execution does not perform overload search, together with well-formedness and typing facts for those states and an erasure theorem back to the ordinary overloaded core semantics (`resolved_call_head_step_erases_to_core_step`), all in `Elaboration.v`.

- **Type soundness.** For closed well-typed expressions, regular constructor signatures, and well-formed dynamic function-entry contexts, both progress and preservation are proved for the plain typing judgment (progress and preservation in Soundness.v).

The mechanization is intentionally narrower than the full implementation. It does not yet give a single end-to-end theorem from raw source programs to fully elaborated resolved core terms. Parser-level ambiguity resolution, full-expression elaboration, and implementation heuristics remain abstract. But the formalization now covers both the core metatheory and a substantial algorithmic layer for compile-time overload resolution, together with the proofs connecting those two views.

6 RELATED WORK

6.1 Natural Language Programming

Kip is related to natural-language-inspired programming languages, but differs in where linguistic structure is operationalized. Dijkstra's critique [4] is often interpreted as an argument against unconstrained natural-language programming; Kip sidesteps this concern because it is not free-form Turkish but a formal language that adopts a narrow, typed fragment of Turkish morphosyntax.

This perspective also connects to work on natural programming languages and environments. Myers, Pane, and Ko argue that programming systems often ignore how users actually formulate tasks and expectations, and they treat expectation alignment as a legitimate design objective rather than an afterthought [18]. Kip shares the narrower premise that surface form matters, though it pursues that premise technically rather than through usability studies: its Turkish morphosyntax is part of the programming interface, not a superficial translation layer. At the same time, Kip does not claim the user-centered validation pursued in that line of work. Its contribution is more technical: it shows how linguistically motivated surface distinctions can participate in typed elaboration and static disambiguation.

Swidan et al. provide a 12-aspect framework for programming language localization [25]; Kip can be read as a deep implementation of structural aspects (morphology, word order, and role marking) rather than keyword-only localization.

6.2 Non-English Programming Languages

Several projects have explored programming languages with non-English syntax. Among these, Tampio is the closest precedent [10]. It is a Finnish natural-language programming language that compiles to JavaScript and uses a real morphological analyzer (libvoikko) for Finnish inflection. The key technical difference is where ambiguity is resolved. In Tampio, ambiguity is handled greedily during lexing and parsing. Candidate analyses are scored against the parser's expected syntactic classes, and a single best reading is chosen immediately without global backtracking. In Kip, multiple morphological analyses survive parsing and are pruned later by type-directed elaboration; unresolved ambiguities must be disambiguated explicitly (for example, with apostrophes).

Beyond the disambiguation strategy, the two languages differ in how deeply case participates in the type system. Tampio primarily uses case as a parsing/role-marking aid in an imperative, JavaScript-targeting setting. Kip uses case as part of typed dispatch for function application, including flexible-order argument matching when case roles are recoverable. It also enforces a static purity boundary—noun-phrase forms for pure code, infinitive/imperative forms for effectful code—checked by the compiler. Where Tampio demonstrates that morphology can make programs readable as natural language, Kip investigates what changes when morphology participates directly in static reasoning.

Perligata [3] draws heavily on Latin morphology and vocabulary, but regularizes and invents forms to fit programming needs, making it Latin-inspired rather than philologically faithful. Kip's orientation differs: it targets consistent, typed program construction rather than a linguistic recasting of an existing language.

Ritter's "Noun Case" post [22] is a useful sketch for case-inspired non-positional notation, but it is explicitly exploratory and not tied to a full formalization or implementation. Its mechanism is mainly case-like sigils and keyword/preposition-style labels, whereas Kip uses actual Turkish inflectional morphology. Both Ritter's sketch and Kip are motivated by role-marked argument structure, but Kip goes further by integrating morphological roles into a typed elaboration pipeline with deferred ambiguity resolution.

6.3 Related Mechanisms in Other Programming Languages

Kip's case-directed argument recovery is related to labeled and named argument mechanisms in several programming languages, even when those mechanisms are not motivated by natural language.

Many languages allow arguments to be passed out of order via explicit naming or labeling—examples include OCaml's labeled arguments [7], Ada's named parameter association [13], Swift's argument labels [1], Smalltalk's keyword messages [8], and Common Lisp and Python keyword arguments [21, 24]. In all these cases, role markers are fixed identifiers—either in the function signature or in the calling convention—rather than inflectional features of the argument expressions themselves.

Kip's nominal/verbal purity boundary is reminiscent of Meyer's command-query separation principle in Eiffel [16], which enforces a syntactic distinction between side-effecting procedures and pure queries. In a looser sense it also parallels Haskell's separation between pure expressions and `IO` actions, with grammatical mood taking on some of the surface role that **do**-notation and monadic types play there. Languages with richer effect systems, such as Koka [14], Eff [2], and Frank [15], instead make effect distinctions explicit through polymorphic effect types and handlers. Kip's version is deliberately lighter-weight and coarser than these richer effect systems: it uses grammatical mood to enforce a purity boundary without exposing an effect-type language in surface syntax.

Taken together, these comparisons clarify Kip's contribution: it reuses a natural language's existing role-marking system as part of typed elaboration, rather than introducing a separate labeling mechanism.

6.4 Controlled Natural Languages

Kuhn characterizes controlled natural languages (CNLs) as engineered varieties of a base natural language that restrict lexicon, syntax, and/or semantics to improve comprehensibility, translation quality, or formal processability [12]. Relative to that definition, Kip is adjacent to CNL work but not a canonical CNL: it does not seek broad natural-language well-formedness as a primary objective. Instead, it uses selected linguistic structure where it supports programming-language design goals.

6.4.1 CNLs as Executable Specification Languages. A useful distinction for Kip's positioning is between CNLs whose primary goal is improved human communication/translation and CNLs designed as executable or formally interpretable specification interfaces. In the latter branch, controlled input is mapped to logic-like representations that can be queried, checked, or executed. Kuhn's survey groups such systems under the formal-representation ("F") objective and traces this line across knowledge representation, querying, and proof-oriented settings [12]. An early representative of this executable-specification line is Fuchs and Schwitter's 1995 system, which translates controlled natural language into Prolog for querying and execution [6]. Later systems, such as PENG and E2V, continue this direction with tightly constrained syntax and explicit mappings to formal representations for automated reasoning [20, 23].

The methodological contrast reinforces this difference: many CNLs begin from ordinary-language-like sentences and constrain them to reduce ambiguity, whereas Kip starts from typed elaboration requirements and incorporates linguistic devices only when they improve static reasoning. That is, CNLs typically optimize for controlled interpretation of near-natural text, whereas Kip optimizes for predictable program meaning under a type system [12].

Kip is therefore best read as complementary to CNL research rather than competing with it. CNL work shows how natural-language constraints can be driven toward precision; Kip shows how linguistic intuitions can inform a programming language while typing criteria remain primary.

7 CONCLUSION

Kip demonstrates that natural-language morphology can do more than decorate a programming language surface: it can enter into the static semantics. In Kip, case marking contributes to argument-role recovery, word order can be relaxed when roles remain recoverable, and ambiguous analyses are settled during typing instead of being forced prematurely at parse time. Unlike keyword-localized translations of English syntax or free-form natural language interfaces, Kip keeps morphological structure operationally meaningful within a compiler-driven pipeline. The key technical claim is not that "programs can look natural," but that inflectional structure can be integrated into predictable static reasoning—with elaboration, type checking, and diagnostics all informed by morphology.

Concretely, case-marked interfaces recover argument roles without a separate labeling mechanism. Deferred morphological disambiguation lets type checking exploit evidence that the parser cannot. The noun/verb purity boundary pairs naturally with non-positional application, avoiding the effect-ordering confusion that labeled arguments create in strict languages. A Rocq mechanization backs these claims with machine-checked proofs of effect rejection, progress, preservation, and elaboration correctness for the core calculus.

More broadly, the project suggests a direction for morphology-aware language design: treat linguistic ambiguity as a managed typing problem, and evaluate syntax choices by their effect on elaboration determinism and diagnostic quality. In that sense, Kip suggests that localization can be a question of elaboration design rather than keyword translation alone. Although Kip targets Turkish, the underlying mechanisms—case-driven role recovery, head-final parsing, and morphological disambiguation—transfer most directly to other agglutinative, head-final languages such as Finnish, Hungarian, Korean, and Japanese, where suffixes decompose predictably; fusional languages like German or Russian have case but would require different strategies for morphological decomposition [5].

Going forward, we plan to conduct user studies on Kip, comparing case-marked and positional interfaces to assess whether the reordering and disambiguation mechanisms improve readability for Turkish-speaking programmers. We also plan to develop richer ambiguity diagnostics that rank likely readings and explain why alternatives fail.

ACKNOWLEDGMENTS

AI tools were used to reword sentences idiomatically, to check the grammar and spelling, and in the coding of both the Kip implementation and its Rocq formalization.

REFERENCES

[1] Apple Inc. 2026. Functions. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/functions/. In *The Swift Programming Language*. Accessed: 2026-02-25.

[2] Andrej Bauer and Matija Pretnar. 2015. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123.

[3] Damian Conway. [n.d.]. Lingua::Romana::Perligata – Perl for the XXI-imum Century. https://web.archive.org/web/20241217212902/https://users.monash.edu/~damian/papers/HTML/Perligata.html. Accessed: 2025-12-16.

[4] Edsger W Dijkstra. 2005. On the foolishness of "natural language programming". In *Program Construction: International Summer School*. Springer, 51–53.

[5] Matthew S. Dryer and Martin Haspelmath (Eds.). 2013. *WALS Online (v2020.4)*. Zenodo. https://doi.org/10.5281/zenodo.13950591

[6] Norbert E. Fuchs and Rolf Schwitter. 1995. Specifying Logic Programs in Controlled Natural Language. In *Proceedings of the Workshop on Computational Logic for Natural Language Processing (CLNLP 1995)*. Edinburgh, Scotland, UK. April 3–5, 1995.

[7] Jacques Garrigue. 2001. Labeled and optional arguments for Objective Caml. In *JSSST Workshop on Programming and Programming Languages, Kameoka, Japan*.

[8] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley.

[9] C. L. Hamblin. 1962. Translation to and from Polish Notation. *Comput. J.* 5, 3 (11 1962), 210–213. https://doi.org/10.1093/comjnl/5.3.210

[10] Iikka Hauhio. [n.d.]. Tampio Programming Language (GitHub Repository). https://github.com/fergusq/tampio. Repository and README. Accessed: 2026-02-26.

[11] Mans Hulden. 2009. Foma: a finite-state compiler and library. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, 29–32.

[12] Tobias Kuhn. 2014. A survey and classification of controlled natural languages. *Computational linguistics* 40, 1 (2014), 121–170.

[13] Henry Ledgard. 1981. *ADA: An Introduction, Ada Reference Manual (July 1980)*. Springer-Verlag.

[14] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science* 153 (2014), 100–126.

[15] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 500–514.

[16] Bertrand Meyer. 1988. *Object-Oriented Software Construction*. Prentice Hall.

[17] Masahiro Mori, Karl F. MacDorman, and Norri Kageki. 2012. The Uncanny Valley. *IEEE Robotics & Automation Magazine* 19, 2 (June 2012), 98–100. https://doi.org/10.1109/MRA.2012.2192811 English translation of Mori's (1970) essay.

[18] Brad A. Myers, John F. Pane, and Amy J. Ko. 2004. Natural programming languages and environments. *Commun. ACM* 47, 9 (Sept. 2004), 47–52. https://doi.org/10.1145/1015864.1015888

[19] Simon Peyton Jones. 2001. Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell. *NATO Science Series Sub Series III Computer and Systems Sciences* 180 (2001), 47–96. https://web.archive.org/web/20221212192523/https://simon.peytonjones.org/assets/pdfs/tackling-awkward-squad.pdf

[20] Ian Pratt-Hartmann. 2003. A two-variable fragment of English. *Journal of Logic, Language and Information* 12, 1 (2003), 13–45.

[21] Python Software Foundation. 1997. *Python Reference Manual* (1.5 ed.). Section 5.3.4 (Calls). Accessed: 2026-02-26.

[22] Getty Ritter. 2014. Noun case. https://web.archive.org/web/20250316160730/https://journal.infinitenegativeutility.com/noun-case. Accessed: 2025-12-16.

[23] Rolf Schwitter. 2002. English as a formal specification language. In *Proceedings. 13th International Workshop on Database and Expert Systems Applications*. IEEE, 228–232.

[24] Guy L. Steele. 1984. *Common LISP: The Language*. Digital Press.

[25] Alaaeddin Swidan and Felienne Hermans. 2023. A Framework for the Localization of Programming Languages. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E* (Cascais, Portugal) *(SPLASH-E 2023)*. Association for Computing Machinery, New York, NY, USA, 13–25. https://doi.org/10.1145/3622780.3623645

[26] Ellen Woolford. 2006. Lexical case, inherent case, and argument structure. *Linguistic inquiry* 37, 1 (2006), 111–130. https://doi.org/10.1162/002438906775321175

[27] Çağrı Çöltekin. 2010. A Freely Available Morphological Analyzer for Turkish. In *Proceedings of the 7th International Conference on Language Resources and Evaluation (LREC 2010)*. 820–827. http://www.lrec-conf.org/proceedings/lrec2010/summaries/109.html

[28] Çağrı Çöltekin. 2014. A Set of Open Source Tools for Turkish Natural Language Processing. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Hrafn Loftsson, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis (Eds.). 1079–1086. http://www.lrec-conf.org/proceedings/lrec2014/pdf/437_Paper.pdf