Tutorial

Joomy Korkut edited this page yesterday · 2 revisions

Hello! Kip is an experimental programming language that combines Turkish grammar rules with a type system. Case endings, vowel harmony, and other Turkish morphological features are an integral part of Kip's type-checking process.

This project explores the interaction between programming language design and linguistics. It is not intended for everyday production use; it is a playground for experimentation, tinkering, and discussion.

> Note
>
> Kip is experimental; syntax and behavior may change from version to version.

Table of Contents

Pages 3

Find a page or section…

Home

Kılavuz

Tutorial

- Table of Contents
- Core Concepts
  - Case Endings
  - Vowel Harmony
  - Multi-word Names
- Data Types
  - Simple Enumeration Types
  - Constructor Syntax
  - Recursive Types
  - Types with No Constructors
- Constant Definitions
  - Consonant Softening
- Function Definitions
  - Basic Structure
  - Example: Unary Function
  - Example: Binary Function
- Pattern Matching
  - Basic Syntax
  - Binders
  - Hyphenated Binders

16. Important Notes

Core Concepts

Case Endings

In Kip, function arguments are distinguished by Turkish case endings. This means the order is less important than the case carried by the suffix.

Case	Turkish Name	Suffix	Example
Nominative	Yalin hal	(none)	`sıfır`
Accusative	-i hali	-i, -ı, -u, -ü	`sayıyı`
Dative	-e hali	-e, -a	`sayıya`
Locative	-de hali	-de, -da, -te, -ta	`listede`
Ablative	-den hali	-den, -dan, -ten, -tan	`listeden`
Genitive	Tamlayan eki	-in, -ın, -un, -ün	`sayının`
Instrumental	-le eki	-le, -la, ile	`sayıyla`
Possessive (3s)	Tamlanan eki	-i, -ı, -u, -ü, -si, -sı	`ardılı`
Conditional	Sart kipi	-sa, -se	`sıfırsa`

Vowel Harmony

Kip uses the Turkish morphological analyzer TRmorph to automatically recognize the correct form of suffixes. For example, the different genitive suffixes in `sayının` and `listesinin` (-nın vs -sinin) represent the same grammatical case. This keeps the written form natural.

Multi-word Names

Kip does not use names with spaces; instead, combine words with a hyphen (-):

```
tam-sayı
işaretli-sayı
öğe-listesi
```

Data Types

Simple Enumeration Types

You can define enum-like types as follows, yielding short and readable definitions:

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

Here, `doğruluk` is the type name, and `doğru` and `yanlış` are constructors (data constructors).

Constructor Syntax

The rule is simple:

- Every constructor starts with `ya`
- The last constructor may start with `ya da` (optional)
- The definition ends with `olabilir.`

```
Bir gün
ya pazartesi
ya salı
ya çarşamba
ya perşembe
ya cuma
ya cumartesi
ya pazar
olabilir.
```

Recursive Types

Types can refer to themselves; this lets us model trees, lists, and similar structures. It may look long at first, but the structure is quite regular.

```
Bir (öğe listesi)
ya boş
```


Clone this wiki locally

```
ya da bir öğenin bir (öğe listesine) eki
olabilir.
```

In this definition:

- `boş` : a constructor with no arguments (an empty leaf)
- `ek` : a constructor that takes an `öğe` and an `öğe listesi`

In `1'in boşa eki` :

- `boşa` : the argument type, in the dative case (-a)
- `eki` : the constructor name, with the possessive suffix (-i)

Types with No Constructors

The empty type (Haskell's `Void`) is defined like this:

```
Bir boşluk var olamaz.
```

Constant Definitions

Define constants with the `diyelim` form. Kip reads like a Turkish sentence here:

```
pazartesiye ilk-gün diyelim.
salıya ikinci-gün diyelim.
çarşambaya üçüncü-gün diyelim.
```

Things to note:

- The value being named takes the **dative case**: `pazartesi**ye**`
- The name is in the base form: `ilk-gün`
- The syntax reads like a sentence: "Pazartesiye ilk-gün diyelim."

Consonant Softening

Turkish consonant softening rules apply; Kip accepts them naturally:

```
kitabın kapağına önkapak diyelim.
```

`kitap + -ın -> kitabın` (p -> b softening)

Function Definitions

Basic Structure

```
(argüman1) (argüman2) fonksiyon-adı,
  gövde.
```

Example: Unary Function

```
(bu doğruluğun) tersi,
  bu doğruysa, yanlış,
  yanlışsa, doğrudur.
```

In this definition:

- `(bu doğruluğun)` : the argument is bound to `bu` (any name works)
- `tersi` : the function name (takes the possessive suffix)
- Body: pattern matching

Example: Binary Function

```
(bu öğe listesiyle) (şu öğe listesinin) birleşimi,
  bu boşsa,
    şu,
  ilkin devama ekiyse,
    ilkin (devamla şunun birleşimine) ekidir.
```

In this definition:

- The first argument takes the `-le` suffix: `bu öğe listesiyle`
- The second argument takes the genitive suffix: `şu öğe listesinin`
- The function name takes the possessive suffix: `birleşimi`

Pattern Matching

Basic Syntax

Pattern matching uses the conditional (-sa/-se). It reads and writes naturally:

```
değer yapkıysa,
  sonuç,
...
```

Binders

You can bind constructor arguments to names:

```
(bu öğe listesinin) kopyası,
  bu boşsa,
    boş,
  ilkin devama ekiyse,
    ilkin (devamın kopyasıyla) birleşimidir.
```

In `ilkin devama ekiyse`, `ilk` and `devam` bind the constructor arguments. This lets you use those names in the body.

Binder names cannot be repeated within the same pattern branch; this avoids confusion.

Hyphenated Binders

Hyphenated names can be used to avoid collisions:

```
bu-öncülün ardılıysa,
  bu-öncülün ardılıdır.
```

Nested Pattern Matching

Pattern matches can be nested:

```
(bu işaretin) çift-öncülü,
  bu boşsa,
    boş,
  bu-öncülün ardılıysa,
    (bu-öncül boşsa,
      boş,
     bu-öncül-öncülün ardılıysa,
      bu-öncül-öncülün ardılıdır).
```

Otherwise (Wildcard)

Use `değilse` to cover all remaining cases:

```
(bu doğruluğun) tersi,
  bu doğruysa, yanlış,
  değilse, doğru.
```

Match Expressions

A match can also be used as an expression, and the scrutinee does not need to be a simple variable. This helps keep transformations fluent:

```
(bu dizgenin) sayının-bir-fazlası,
  ((bunun tam-sayı-hali)
    yokluksa, 0,
    n'nin varlığıysa, (n'nin 1'le toplamıdır)).
```

Matching Built-in Types

Built-in (primitive) types like integers cannot be matched with constructors; you can only match on their own constructors.

Function Calls

Basic Call

```
(ikiyle üçün toplamını) yaz.
```

Flexible Argument Order

One nice feature of Kip is that argument order is flexible. The two calls below are equivalent:

```
(5'le 3'ün farkını) yaz.
(3'ün 5'le farkını) yaz.
```

This flexibility is possible because **arguments take different cases**. Kip determines which argument is which by looking at the case suffix.

> Important
>
> If more than one argument takes the same case, order matters.

Imperative Calls

Effectful functions defined in the infinitive (-mak/-mek) can be called in the imperative. This makes calls look more natural:

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

selamla.
```

Built-in Types

Integers

```
Bir yerleşik tam-sayı olsun.
```

Integer operations:

```
(bu tam-sayıyla) (şu tam-sayının) toplamı,
yerleşiktir.
(bu tam-sayıyla) (şu tam-sayının) farkı, yerleşiktir.
(bu tam-sayıyla) (şu tam-sayının) çarpımı,
yerleşiktir.
(bu tam-sayının) öncülü, yerleşiktir.
(bu tam-sayının) sıfırlığı, yerleşiktir.
(bu tam-sayıyla) (şu tam-sayının) eşitliği,
yerleşiktir.
(bu tam-sayının) (şu tam-sayıdan) küçüklüğü,
yerleşiktir.
(bu tam-sayının) (şu tam-sayıdan) küçük-eşitliği,
yerleşiktir.
(bu tam-sayının) (şu tam-sayıdan) büyüklüğü,
yerleşiktir.
(bu tam-sayıyla) (şu tam-sayının) büyük-eşitliği,
```

```
yerleşiktir.
(bu tam-sayının) faktöriyeli, yerleşiktir.
```

Integer Literals

Numeric literals can be used directly; when taking suffixes, use an apostrophe:

```
5'i yaz.
(5'le 3'ün toplamını) yaz.
-1'i yaz.
```

Strings

```
Bir yerleşik dizge olsun.
```

String operations:

```
(bu dizgenin) uzunluğu, yerleşiktir.
(bu dizgeyle) (şu dizgenin) birleşimi, yerleşiktir.
(bu dizgenin) tam-sayı-hali, yerleşiktir.
```

The `tam-sayı-hali` conversion may fail; the result is an `olasılık` value.

String Literals

```
"merhaba"'yı yaz.
"merhaba"'yla "dünya"'nın birleşimini yaz.
```

Escape Sequences

Common escape sequences can be used in strings:

```
"a\nb\tc\\\"d"'yi yaz.
```

Effectful Functions

Printing

```
(bu şeyi) yazmak, yerleşiktir.
(bu dizgeyi) yazmak, yerleşiktir.

5'i yaz.
"merhaba"'yı yaz.
```

Reading

```
okumak, yerleşiktir.
```

Reads a line from standard input and returns a string. It is ideal for REPL experiments.

File I/O

```
(bu dosyadan) (okumak dizge olasılığı), yerleşiktir.
(bu dosyaya) (şu dizgeyi) (yazmak doğruluğu),
yerleşiktir.
```

Reading from a file returns an `olasılık` type:

```
çalıştırmak,
  ("./tests/yazı.tmp"'den okumak)
    yokluksa,
      "Okunamadı." yazmaktır,
    metnin varlığıysa,
      metni yazmaktır.

çalıştır.
```

Sequencing

You can sequence multiple operations with the converb suffixes `-ıp/-ıp/-up/-üp`:

```
(bu tam-sayıyı) denemek,
  bunu yazıp,
  (bunla 1'in toplamını) yazmaktır.
```

```
5'i dene.
```

Binding

Use `olarak` to bind the result of an expression to a name:

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

selamla.
```

In this example:

1. The `okumak` function is called
2. The result is bound to the variable `isim`
3. `isim` is used in the next expression

Unit Type

Effectful operations typically return the `bitim` type (OCaml's `unit`, C-like languages' `void`). `durmak` is a simple value of this type:

```
(durmak bitimi),
  bitimliktir.
```

Modules and Loading

Kip files are imported with `yükle`. The `yükle` command expects the accusative case (e.g. `girişi yükle.`). By default, `lib/giriş.kip` is loaded automatically, so you start with basic modules. You can disable this with `--no-prelude`.

> **Important**
>
> If automatic loading is enabled, you do not need to load basic modules yourself.

To load manually:

```
girişi yükle.
```

Polymorphic Types

Type Variables

Polymorphic types can be defined:

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

Here `öğe` is a type variable. This type can be instantiated as `tam-sayı listesi`, `dizge listesi`, `doğruluk listesi`, and so on.

Building Lists

```
doğrunun (yanlışın boşa ekine) ekini yaz.
```

This can be thought of like `[true, false]` in other languages.

List Functions

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

(bu öğe listesiyle) (şu öğe listesinin) birleşimi,
  bu boşsa,
    şu,
  ilkin devama ekiyse,
    ilkin (devamla şunun birleşimine) ekidir.

(bu öğe listesinin) tersi,
  bu boşsa,
    boş,
  ilkin devama ekiyse,
    (devamın tersiyle) (ilkin boşa ekinin)
birleşimidir.

(bu tam-sayı listesinin) toplamı,
```

```
bu boşsa,
  0,
ilkin devama ekiyse,
  ilkin (devamın toplamıyla) toplamıdır.
```

Optional Type (Maybe)

```
Bir (öğenin olasılığı)
ya yokluğu
ya da bir öğenin varlığı olabilir.
```

By matching on `yokluk` and `varlık`, you can write safe conversions:

```
(bu dizgenin) sayıya-çevrimi,
  ((bunun tam-sayı-hali)
    yokluksa, yokluğudur,
    n'nin varlığıysa, n'nin varlığıdır).
```

Ambiguities and Disambiguation

Morphological Ambiguity

In Turkish, some words can be analyzed in more than one way. For example "takası":

- `taka` + buffer consonant + possessive suffix
- `takas` + accusative case

Kip keeps all such possibilities and chooses the correct one during type checking, based on context.

> Tip
>
> If Kip cannot resolve a type because of ambiguity, but you know the expression should be valid, you can use an apostrophe to make your intent explicit.

Disambiguation with Apostrophes

Use apostrophes to disambiguate:

```
taka'sı   (* taka'nın tamlananı *)
takas'ı   (* takası -i haliyle *)
```

Function Overloading

You can define functions with the same name but different types:

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

(bu doğrulukla) (şu doğruluğun) birleşimi,
  bu doğruysa,
    doğru,
  yanlışsa,
    şudur.

(5'le 2'nin birleşimini) yaz.           (* 7 *)
(doğruyla yanlışın birleşimini) yaz.    (* doğru *)
```

Kip selects the correct function based on argument types. For readability, the return type can also be included in the name.

In the standard library, some function names include the return type (e.g. `toplam tam-sayısı` , `uzunluk tam-sayısı`). This is used for overloading and readability.

Syntax Summary

Comments

```
(* Bu bir yorumdur *)
```

Loading

```
modül-adını yükle.
```

Type Definitions

```
Bir tip-adı ya yapkı1 ya yapkı2 ya da yapkı3 olabili
Bir tip-adı ya yapkı1 ya da bir argüman-tipinin yapkı2
```

```
olabilir.
Bir yerleşik tip-adı olsun.
Bir boş-tip var olamaz.
```

Constant Definition

```
değere isim diyelim.
```

Function Definitions

```
(argüman1) fonksiyon-adı,
  gövde.

(argüman1) (argüman2) fonksiyon-adı,
  gövde.

fonksiyon-adı, yerleşiktir.
```

Pattern Matching

```
değer yapkıysa, sonuç, değilse, varsayılan-sonuç.
değer yapkı1ysa, sonuç1, yapkı2ysa, sonuç2.
değer bağlayıcının yapkısıysa, bağlayıcıyı-kullanan-
sonuç.
```

Effectful Expressions

```
ifade1ip, ifade2.
isim olarak ifadeyi, ismi-kullanan-ifade.
değeri yaz.
```

Compiler Cache

Kip stores the parsed and type-checked form of each `.kip` file in a `.iz` cache next to it. (Think of it like Python bytecode.) If the source file and its `yükle` dependencies have not changed, Kip uses the `.iz` file.

The `.iz` file also stores hashes for the compiler and the file, so the cache is automatically invalidated when the compiler changes. If you want to re-parse and re-type-check, delete the corresponding `.iz` file.

> **Note**
>
> The `kip --build` command helps you pre-generate `.iz` caches.

Standard Library

`lib/temel.kip` contains the basic types; standard functions live in related modules:

- Basic types: `doğruluk`, `boşluk`, `olasılık`, `liste`, `yerleşik tam-sayı`, `yerleşik dizge`
- Extra basic type: `bitim`
- `temel-doğruluk.kip`: `tersi`, `birleşimi`, `kesişimi`
- `temel-tam-sayı.kip`: `toplamı`, `farkı`, `çarpımı`, `eşitliği`, `küçüklüğü`, `küçük-eşitliği`, `büyüklüğü`, `büyük-eşitliği`, `öncülü`, `sıfırlığı`, `faktöriyeli`
- `temel-dizge.kip`: `uzunluğu`, `birleşimi`, `tam-sayı-hali` (result: `olasılık`)
- `temel-liste.kip`: `uzunluğu`, `birleşimi`, `tersi`, `toplamı` (integer list)
- `temel-etki.kip`: `durmak`, `yazmak` (thing/string), `okumak`, file `okumak`, file `yazmak`

Important Notes

1. Kip does not care about line starts or indentation; you can format freely.
2. Single-letter names require an apostrophe: `a'nın`, `b'ye`
3. Functions and constructors end with -dır/-dir (optional but recommended)
4. This is a research project; it is not designed for production use.

> **Caution**

Because it is experimental, we do not recommend using Kip for long-term projects.