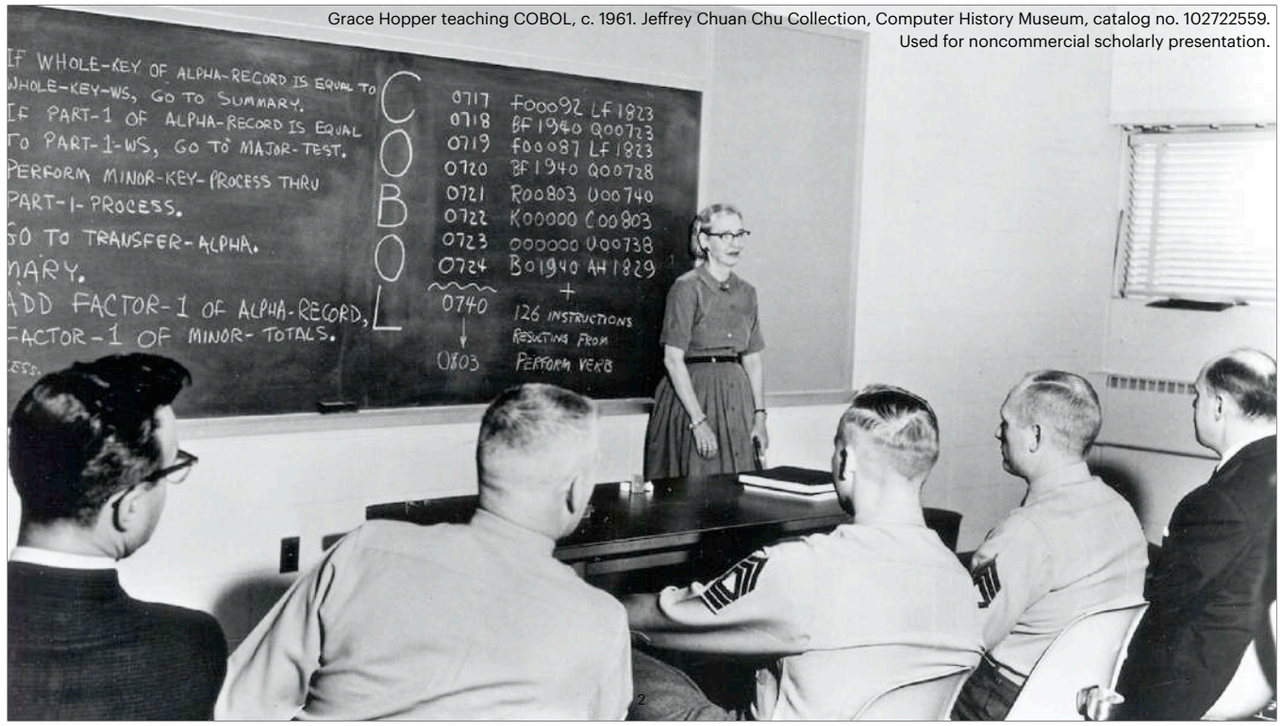
Morphosyntactic Programming

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

Joomy Korkut, Bloomberg
Alperen Keles, University of Maryland, College Park
Onur Akdemir, Topsort

New Jersey Programming Languages and Systems Seminar
May 22, 2026

Hi all,
I’m Joomy. Today I’ll talk to you about a passion project of mine I’ve been working on in my free time. And that is, a programming language in Turkish, where we bring the grammatical cases and moods of Turkish to the type system. This may be a niche topic for this audience but I’ll do my best for everyone to take away something from it.



Grace Hopper teaching COBOL, c. 1961. Jeffrey Chuan Chu Collection, Computer History Museum, catalog no. 102722559. Used for noncommercial scholarly presentation.

But first, a little history. Even as early on as 1960, PL folks have been interested in converging their programming languages to natural languages. Making an English-like programming language and thus appealing less experienced programmers and even non-programmers has been a goal in the development of COBOL, for example.

```
247 ПРОЦ ВПЕРЕД();
248  ЕСЛИ ПЕРЕДСОСТАВОМ()="ТУПИК" ТО
249    ВЫВОД:"??НЕ МОГУ, НЕКУДА?";
250    ВЫХОД
251  ИНАЧЕ
252    <**>->ВОЗВСТР;
253    ОКНОДАЙ(КАРТ);
254    ШАГСОСТАВА();
255    ПОКА ПЕРЕДСОСТАВОМ()="СВОБОДНО"::
256      ШАГСОСТАВА()
257    ВСЕ;
258    ДЛЯ И ИЗ ВОЗВСТР::
259      ПУТИ[И,9]->ПУТИ[И,8];
260    ВСЕ;
261    ДЛЯ И ОТ 2 ДО #ПУТИ::
262      ЕСЛИ ПУТИ[И,1]="СТРЕЛКА" ТО
263        КРАСЬСТРЕЛК(И)
264      ИНАЧЕ
265        ПОЗ(0,ПУТИ[И,5]*4);
266        ВЫВОД БПС:"?",ФТЕКСТ(3*ПУТИ[И,4]-1,"="),"<"
267      ВСЕ
268    ВСЕ;
269    ДЛЯ И ОТ 1 ДО #ВАГОНЫ::
270      РИСВАГ(И,0)
271    ВСЕ;
272  ВСЕ;
273  ОКНОДАЙ(ДИАЛ);
274 КНЦ;
275
276 ФУНК ПЕРЕДСОСТАВОМ();
277 ИМЕНА:ПУТЬ,МЕСТО,Р;
278 _ВАГОНЫ[СОСТАВ[1],4]->ПУТЬ;
```

Rapira (1982)

Of course, the Anglosphere wasn’t alone in its search for natural language-like programming languages. The Soviets also had their own Russian language systems and programming languages. Here you see the Rapira programming language from 1982, a dynamically typed, ALGOL-like language.

(This program implements some kind of train simulation.)

Qalb (2013)

This fascination of approximating programming languages to natural language has not ceased since then. Here's the REPL for the Qalb programming language from 2013. It's a Lisp dialect with Levantine Arabic keywords. Here we see a Fibonacci implementation in Qalb.

Something you probably have not noticed here unless you're a Russian or Arabic speaker is that these programming languages are linguistically not all that interesting. Their keywords are in Russian and Arabic but they don't use any other grammatical features from those natural languages.

And I think that’s a huge missed opportunity. There is especially one linguistic concept that I think would enrich the way we program, and that is grammatical cases. If you don’t speak a language in which cases are very prominent, let’s have a quick look at this sentence: “Give him a book.”

Here, why do we say “him” and not “he”? It is because this is one of the few places in English where we still have grammatical cases! But in my situation, the natural language I care about is Turkish.

Case	Ending	Example		Translation	
Nominative	-∅-	*ev* ("house")	*adam* ("man")	"(the) house"	"(the) man"
Accusative	*-(y)ı-, -(y)i-, -(y)u-, -(y)ü-*	*evi*	*adamı*	"the house"	"the man"
Dative	*-(y)a-, -(y)e-*	*eve*	*adama*	"to the house"	"to the man"
Locative	*-da-, -de-, -ta-, -te-*	*evde*	*adamda*	"at home"	"in/on the man"
Ablative	*-dan-, -den-, -tan-, -ten-*	*evden*	*adamdan*	"from the house"	"from the man"
Genitive	*-(n)ın-, -(n)in-, -(n)un-, -(n)ün-*	*evin*	*adamın*	"the house's"	"the man's"
Instrumental	*-(y)le-, -(y)la-*	*evle*	*adamla*	"with the house"	"with the man"

Turkish has at least 7 cases. The declensions happen through suffixes. These suffixes can take different forms according to the last vowel before the suffix, or whether there was a vowel or consonant before the suffix. Obviously we’re not going to spend our time learning this. But if you ever tried to learn Latin, German, Russian, Hungarian, Finnish etc., this type of declension tables should look very familiar to you.

difference of 5 and 3

Here’s one way I think cases can enrich our programs. Suppose we want to write an expression that says “difference of 5 and 3”.

difference of 5 and 3

5’le 3’ün farkı

In Turkish, you’d say “5’le 3’ün farkı”. Notice how the word that corresponds to “difference” is at the end here. Noun phrases in Turkish have their head at the end.

difference of 5 and 3

5'le

instrumental case

3'ün

genitive case

farkı

nominative case with possessive suffix

Here, the first part takes an instrumental case suffix, the second part takes a genitive case suffix, and the last part, the part that actually says difference, takes a possessive suffix.

difference of 5 and 3

3'ün — genitive case

5'le — instrumental case

farkı — nominative case with possessive suffix

The thing is, the specific grammatical cases required in how you say difference, allows you to switch the order in which you say the numbers in the subtraction. In either order, the listener will understand that this phrase means 2 and not -2.

So, we made a programming language, called Kip, in which that phrase is a function call… +


```
Kip> 5'le 3'ün farkı
2

Kip> 3'ün 5'le farkı
2
```

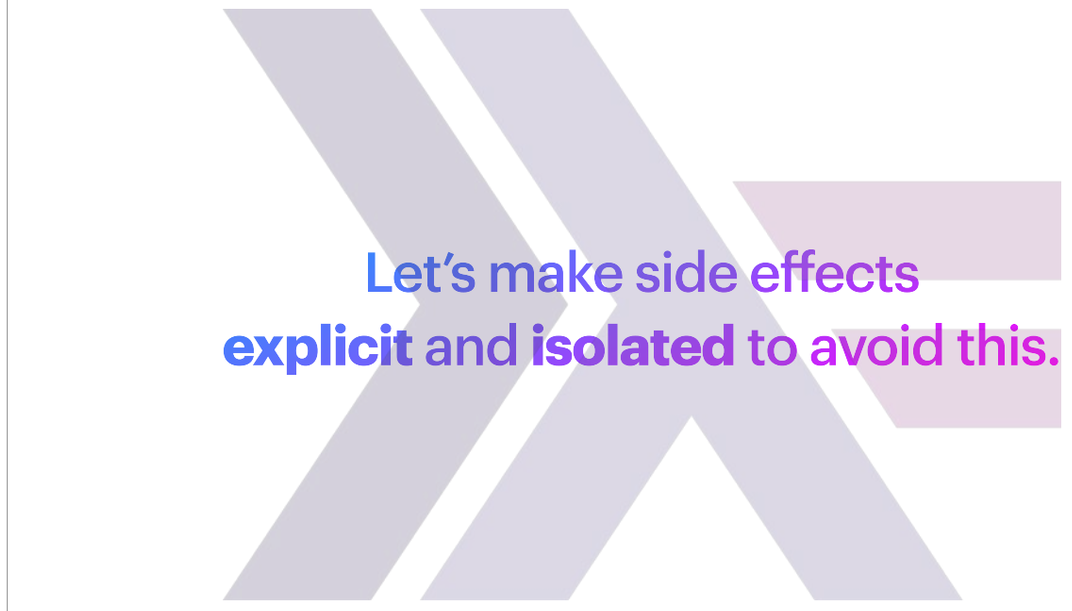

+ and changing the order of the arguments doesn’t change the result as long as the cases of the arguments are unique.

Now, this design choice comes with other considerations.

```
# let difference ~x ~y = x - y;;
val difference : x:int -> y:int -> int = <fun>

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

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

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

Currying changes the order of side effects!

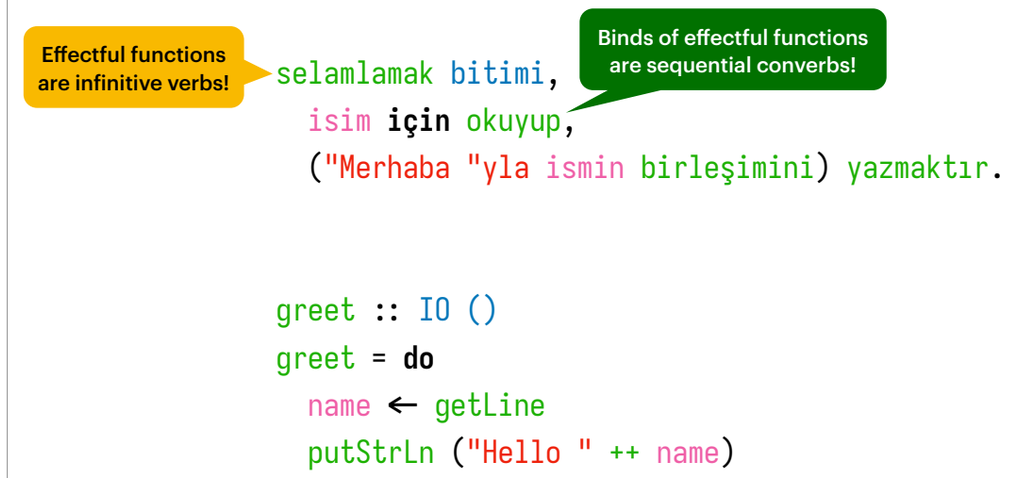
Here’s a quick OCaml refresher. OCaml has labelled arguments, which also allow flexible order of argument application.
<click> But what happens when the argument expressions have side effects? Well, it turns out, they are evaluated from right to left. (not specified in OCaml today, it was specified in ZINC back in the day, but OCaml does it right to left in practice)
<click> Let’s put the flexible order of application in action! I flipped the order here. The result is still 2, but somehow the side effects are not happening from right to left. They are evaluated right to left according to the order given in the original function definition. Let’s pretend I’m ok with that for now.
<click> Ok, but what if we curry it? We put parentheses around the first application, and then passed the second argument separately. But that changed the order in which the side effects appeared!



I don’t like this uncertainty. Let’s make side effects explicit and isolated to avoid this. Just like in Haskell, except without ever mentioning monads.

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

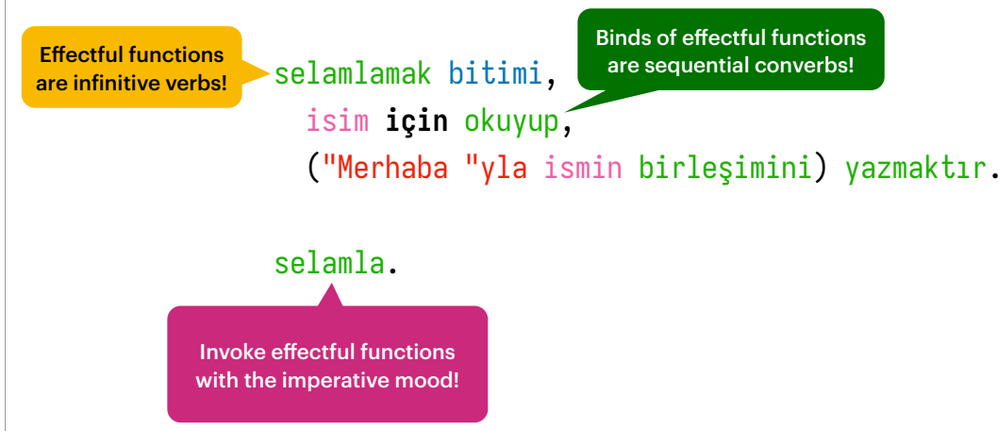

Here’s what an effectful function looks like in Kip. The equivalent Haskell program looks almost the same…



I don’t have time to go over the morphological analysis of this entire function, but there’s some interesting stuff going on here!

<click> We use infinitive verbs for names of effectful functions! That means you don’t have to write in the type that a function is effectful! Kip’s morphological analyzer can figure out that this word is an infinite verb and therefore must be a function definition.
<click> We use a sequential converbs, a grammatical construct in Turkic languages, to bind the result of an effective function call to a pure variable.

But of course, in Haskell, you don’t have a way of invoking `greet` without calling it from `main`, unless you’re in the Haskell REPL. Kip is a bit more relaxed than that and allows top level effectful function invocations, so everywhere is like the REPL.

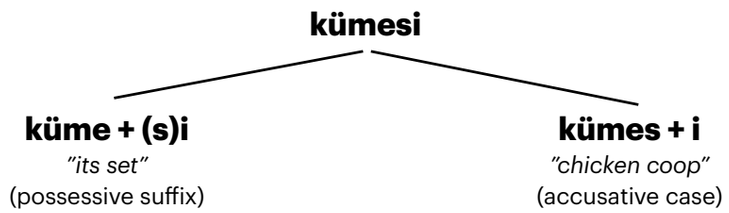


Like this. Notice that the invocation of the effectful function is in imperative mood!

When we were defining the effectful function, we were saying “to greet is to read the name and print hello name”.
When we are calling the effectful function, we are saying “greet!” as a command.

Ok, now that we have some understanding of what Kip programs are like, you might be asking: Why is this even technically interesting, why does this language have a place in a PL venue?

Ambiguity Is Deferred to Elaboration



The main reason is how we deal with ambiguity. Like most other natural languages, Turkish morphology is ambiguous. A surface form may have several possible analyses:

kümesi can mean
  küme-si = its set, with the possessive suffix
  kümes-i = the chicken coop, with the accusative case

A parser alone often cannot know which reading is intended. Kip, using an external morphological analyzer, parses identifiers into candidate sets of possible roots and possible cases.

Kip's type system tries to prune the candidate sets, using scopes, function arities, case compatibility, type compatibility, and effect mode. This is essentially type-directed constraint solving. After the solving, if there are no candidates left, then you have a good old fashioned type error, if you have multiple candidates left, then you have an ambiguity error. You want to end up with a single correct candidate.

This is kind of a novel approach because other natural language based programming languages usually treat such ambiguities as parse errors. Instead, we solve those ambiguities in elaboration, which allows us to accept more well-formed programs.

Flexible Application Order Everywhere

κ	::=	:NOM | :ACC | :DAT | :LOC | :ABL | :GEN | :INS | :COND | :P3S	*grammatical case labels*
κ_r	::=	:NOM | :P3S	*return-case labels*
κ_{res}	::=	$\varnothing \mid \kappa_r$	*result-case marker*
μ	::=	pure | eff	*effect modes*
c	::=	$n \mid fl \mid s \mid ch$	*integer, float, string, and character literals*
τ	::=	α | Int | Float | String | Char | $D\{\tau_1\kappa_1, \ldots, \tau_m\kappa_m\} \mid \tau_1 \to \tau_2$	*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*
	|	$f\{e_1\kappa_1, \ldots, e_n\kappa_n\} \mid C\{e_1\kappa_1, \ldots, e_n\kappa_n\}$	*case-labeled application*
	|	match e with $p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$	*match*
	|	let $x = e_1$ in e_2	*pure local binding*
	|	$e_1; e_2 \mid x \leftarrow e_1; e_2$	*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	::=	data $D\{\alpha_1\kappa_1, \ldots, \alpha_m\kappa_m\}$ 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.

Kip has an ML-like type system, so it has predicative polymorphism and algebraic data types. But keep in mind that the flexible application orders are everywhere in Kip. <click> Parameters of data types, arguments of data type constructors, patterns on data type constructors, they all support flexible application order. Especially combined with the ambiguities we just talked about, this gets tricky to get right.

By the way, you can find this formalization of core Kip in the paper. We also have a Rocq formalization and proofs of progress and preservation, among other things.

Head-Final Syntax is like Reverse Polish Notation

difference of 5 and 3

5'le 3'ün farkı

A less consequential point that I find quite cool is that head-final syntax is like reverse Polish notation!

Remember this expression? This is how you say the difference of 5 and 3 in Turkish. The word for difference is the last word. It is also a valid Kip expression.

Head-Final Syntax is like Reverse Polish Notation

the product of the difference of 5 and 3 and the difference of 4 and 1

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

Now, let's do a more complicated expression. If you were writing this expression in say, Python, you'd have to put the subexpression in parentheses. But interestingly, you don't have to do that in Kip! Since the operation is at the end, it is like parsing reverse Polish notation. And famously, reverse Polish notation does NOT require parentheses.

Head-Final Syntax is like Reverse Polish Notation*

the product of the difference of 5 and 3 and the difference of 4 and 1

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

*** if you don't take into account function overloading with different arities.**

So we can just write it this way. This both reads and looks like natural language, and it has no ambiguity.

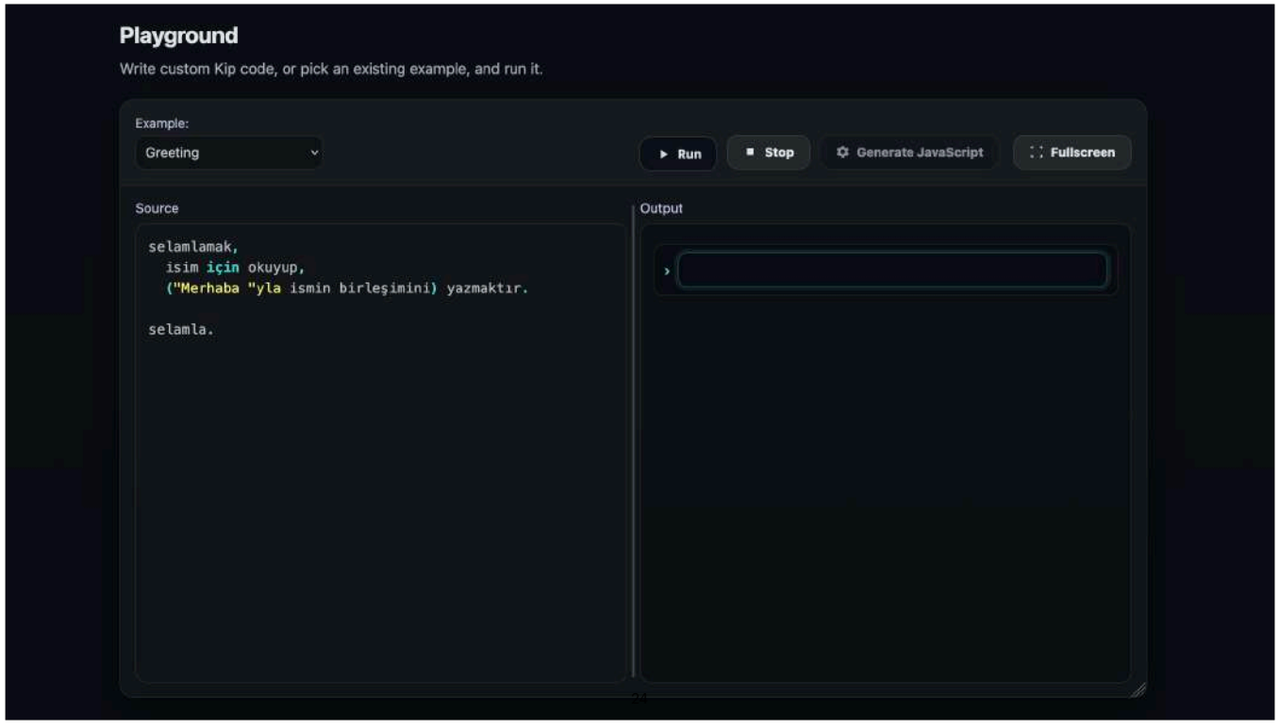
Does that mean we never need to use parentheses to delineate subexpressions? No. There should be an asterisk here.
<click> If you have function overloading where you have functions with different arities sharing the same name, then you cannot depend on the notation and leave out the parentheses.

On the foolishness of "natural language programming".

Since the early days of automatic computing we have had people that have felt it as a shortcoming that programming required the care and accuracy that is characteristic for the use of any formal symbolism. They blamed the mechanical slave for its strict obedience with which it carried out its given instructions, even if a moment's thought would have revealed that those instructions contained an obvious mistake. "But a moment is a long time, and thought is a painful process." (A.E.Housman). They eagerly hoped and waited for more sensible machinery that would refuse to embark on such nonsensical activities as a trivial clerical error evoked at the time.

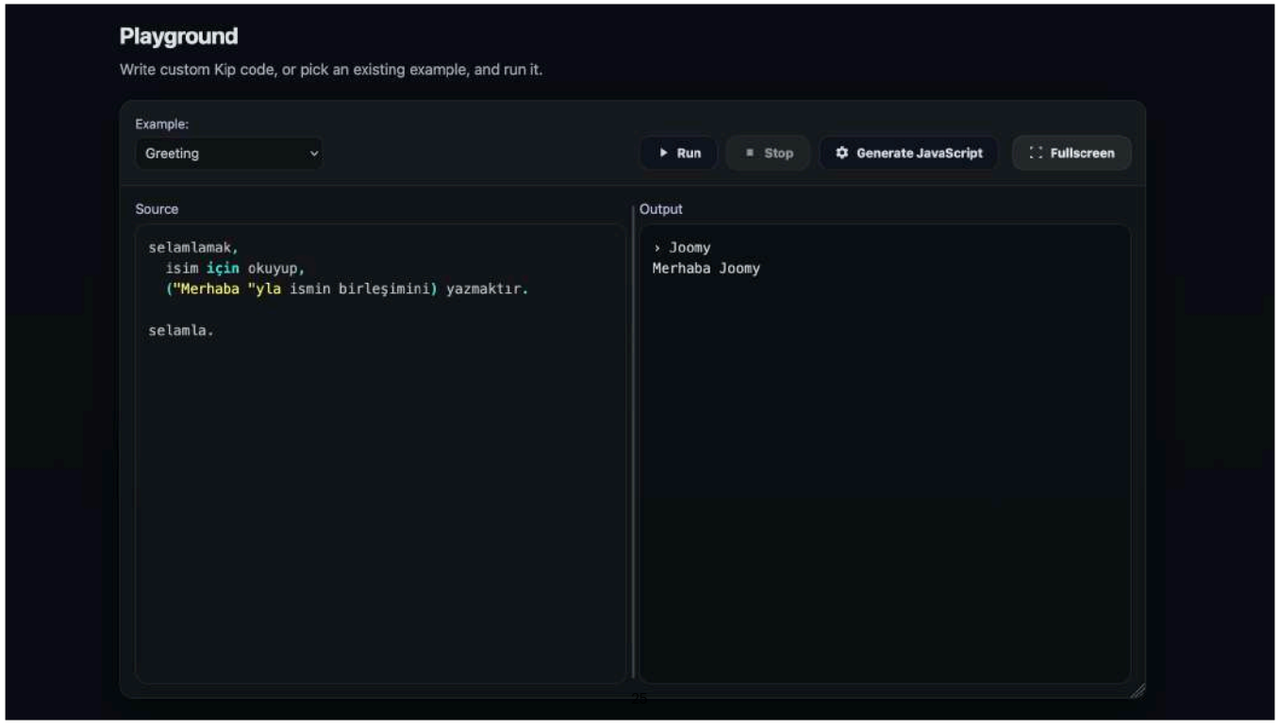
Perhaps this is a good moment to preempt some of the possible criticism (like this famous Dijkstra essay) and make our position clear. We don’t care about natural language programming in the sense that we should just write natural language prose and the computer should understand what we mean (though frankly, there’s enough discourse about that already these days)

We only care about natural language programming in the sense that there are ideas in linguistics that we can **borrow** to make our programming languages richer. These ideas may make our languages less or more usable, we don’t know that yet; it’ll be different for every idea. We haven’t yet done a user study for Kip. Our goal was just to conduct this experiment and see if it’s doable.

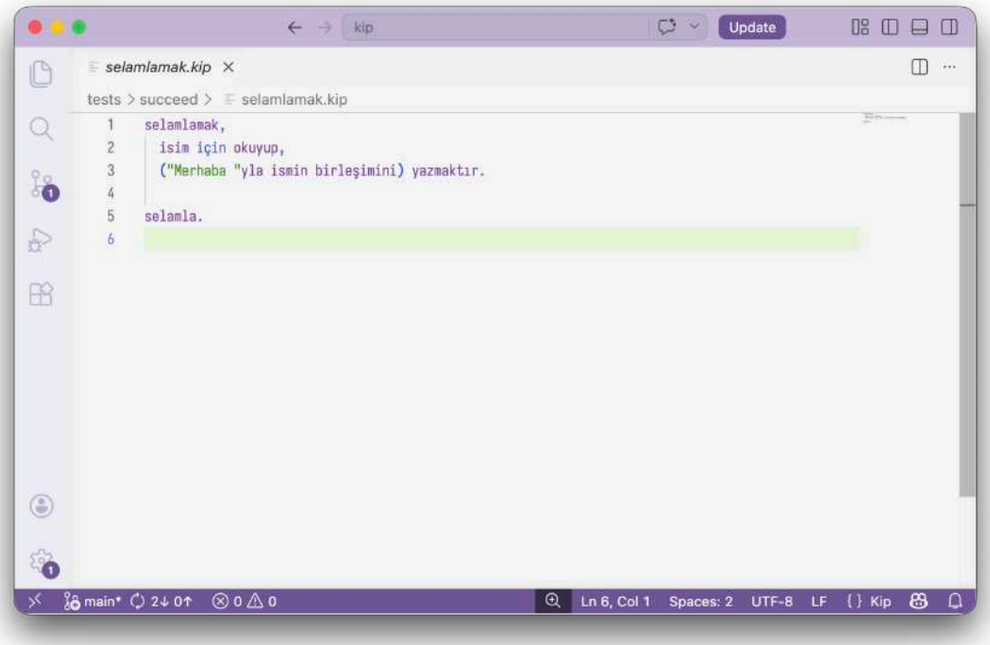


Before I finish, I should talk a bit more about the implementation as well. We implemented Kip in Haskell. It has a REPL that runs the way you would expect. But also, we compiled it to WebAssembly (thanks to GHCJS), and it can also run in your browser.

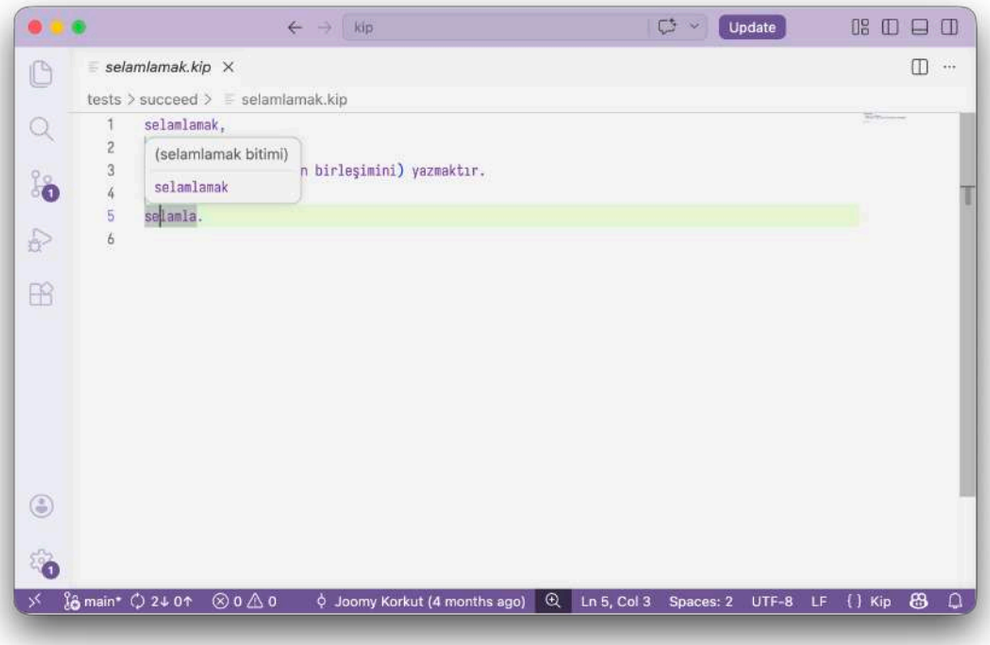
Here we run the program we say earlier and it prompts us for a name.



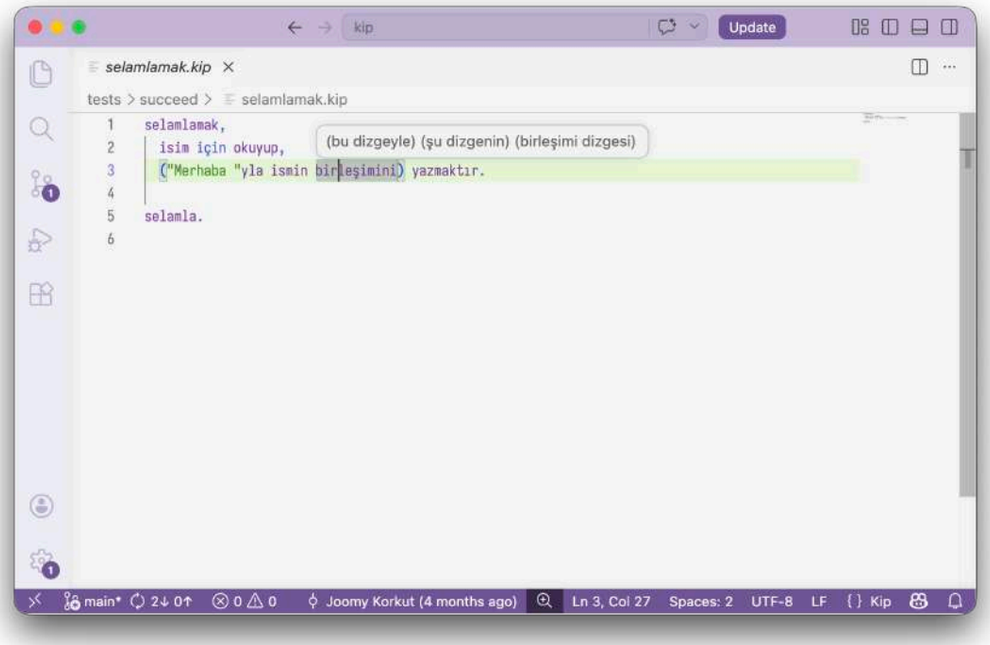
When we enter the name, it prints a hello message for us.



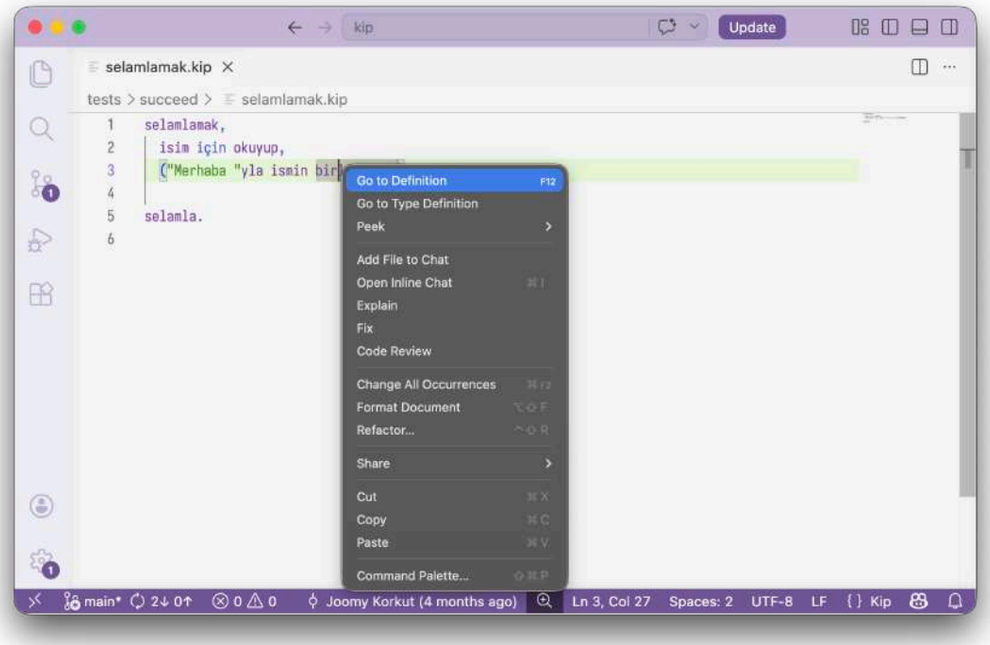
We have an LSP server implemented in Haskell. We have a VSCode plugin and a Vim plugin that interacts with this LSP server.



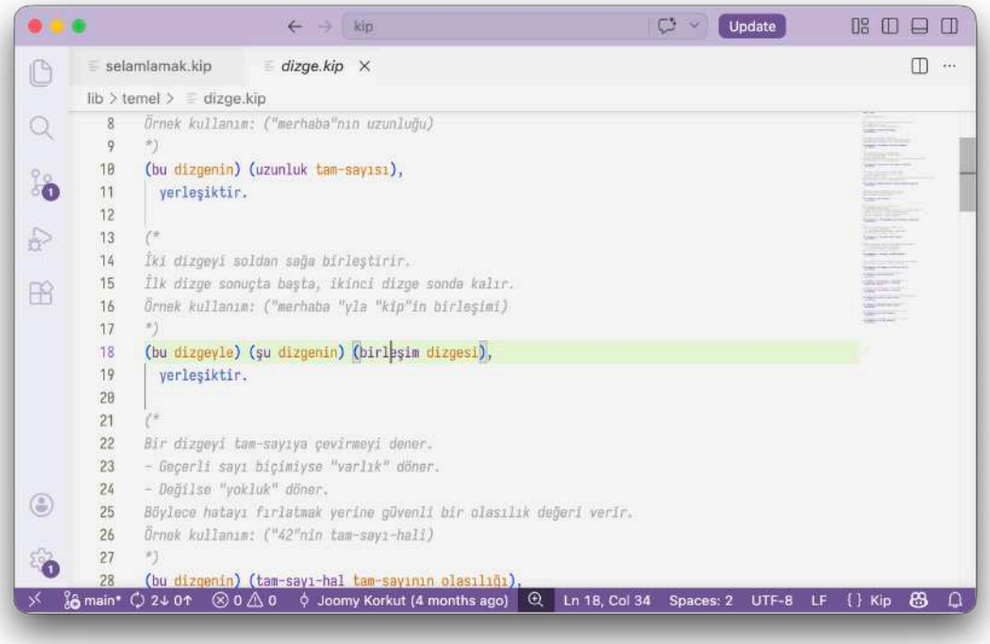
It has all the basic features like showing the type of a function when we hover with the cursor.



Here’s another one.



I can also right click and go to the definition of a function. Here we are trying to go to the definition of string append.


kip
Update
selamlamak.kip
dizge.kip
lib > temel > dizge.kip
8 Örnek kullanım: ("merhaba"nın uzunluğu)
9 *)
10 (bu dizgenin) (uzunluk tam-sayısı),
11 yerleşiktir.
12
13 (*
14 İki dizgeyi soldan sağa birleştirir.
15 İlk dizge sonuçta başta, ikinci dizge sonda kalır.
16 Örnek kullanım: ("merhaba "yla "kip"in birleşimi)
17 *)
18 (bu dizgeyle) (şu dizgenin) (birleşim dizgesi),
19 yerleşiktir.
20
21 (*
22 Bir dizgeyi tam-sayıya çevirmeyi dener.
23 - Geçerli sayı biçimiyse "varlık" döner.
24 - Değilse "yokluk" döner.
25 Böylece hatayı fırlatmak yerine güvenli bir olasılık değeri verir.
26 Örnek kullanım: ("42"nin tam-sayı-hali)
27 *)
28 (bu dizgenin) (tam-sayı-hal tam-sayının olasılığı),
main* 2↓ 0↑ 0 0 Joomy Korkut (4 months ago) Ln 18, Col 34 Spaces: 2 UTF-8 LF { } Kip

And we did.

https://github.com/kip-dili/kip

https://kip-dili.github.io

And that is all I have for you today. We made a Turkish programming language, called Kip, in which all function arguments carry grammatical cases, pure definitions are noun phrases, while effectful definitions are verb phrases in the infinitive, and effectful function calls appear in the imperative mood.

You can find the repo and the playground at the links here. Thank you for listening!