Social Programming Considered as a Habitat for Groups

Joe Edelman
Human Systems Curriculum
Berlin, Germany
joe.edelman@gmail.com

ABSTRACT

A new way to code up social apps and information systems emerges from studying how people use ordinary speech to set up social roles and obligations. Sentences like "whoever hosts the potluck has to contact the guests" and "once two people swipe right on each other, they can exchange messages" are made machine interpretable with a little extra punctuation. This approach–combined with a shared database of these social rules–leads to social software that works more like social conventions: it can be defied, expressively reinterpreted, and remodeled by the user. It also leads to much more flexible style of coordination, with greater support for individual leadership, individual discretion, and more open-ended collaborations. This can address a number of modern social ills.

CCS CONCEPTS

• **Software and its engineering** → **Specialized application languages**; **Application specific development environments**; • **Information systems** → *Collaborative and social computing systems and tools*; • **Human-centered computing** → *Interactive systems and tools*; *Collaborative and social computing systems and tools*;

KEYWORDS

Programming experience; Social programming

ACM Reference Format:
Joe Edelman. 2018. Social Programming Considered as a Habitat for Groups. In *Proceedings of Programming Experience Workshop (PX/18).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.475/....

1 INTRODUCTION

Software is eating the world. [1]

Here's another way of saying that: Software is *coming between people* and *deciding how they talk and work*. It's *mediating interactions*, in organizations of all sizes, from farms to multinational companies to friend groups.

It's useful to break this into two related phenomena:

1. **Software eats older forms of bureaucracy**–replacing form-based paperwork, checklists and other ways of tracking work–usually in multilayered organizations.
2. **Software creates new bureaucracy** as software-based messaging spreads into informal, lightweight organizations like friend groups and book clubs.

Most technologists think this is a good thing, and there are certainly advantages. But let's consider the downsides of each:

PX/18, XXXXXXXXX, 2018, Nice, France
© 2018 Copyright held by the owner/author(s).
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of Programming Experience Workshop (PX/18)*, https://doi.org/10.475/.....

1.1 Software Alters Old Bureaucracy

In businesses, universities, government organizations, and so on, information flow and work coordination have been automated via software.

Compared to the previous technologies used–paper forms, stamps, signatures, private conversations, group meetings, checklists, written correspondence–the new systems suffer from **rigid expectations**. It's difficult for participants to do something different than what the software assumes, or to post data that doesn't fit its schemas.

Authority becomes **impersonal** and **mechanistic**, as people report to the script or to dashboards, rather than to one another. They then feel less responsible for broader goals and for each other.

The organizations become **over-mediated**, as everything needs to be done and tracked through screen-based interfaces. Simple social actions can't happen naturally through conversation, because these in-person dialogues won't advance the script correctly or leave the online data in the right state.

Finally processes become **esoteric** and users with better ideas often can't change the way the automation works, or suggest alternatives.

1.2 Software Creates New Bureaucracy

Information software and coordination also find their way into previously informal processes. Friend groups use group messaging, calendaring software, and systems like facebook groups and events to coordinate birthday parties, trips, and the like.

This usually means a shift towards asynchronous modes of collaboration, and thus a tendency towards **notification overload** or **endless checking** to keep things moving along.

To the extend that these systems advance via text utterances or posts, it's hard to **get an overview** of where things stand–both in terms of project status and in the relationship between people. There are **covert, unintentional power dynamics**.

Relatedly, interactions in these systems tend to be limited to **short term utterances** rather than **long-term visions or plans**. There's no **clarity about ongoing roles**.

Finally group processes run on generic friend-coordination services are always **starting from scratch**. There's nowhere to encode **wisdom about process**.

2 APPROACH

Taken together, these problems with software bureaucracy are significant. We are all familiar with them. They may even be responsible for an economic slowdown.[3]

Where did we go wrong with information systems and social software?

We can find clues in how humans use ordinary language.[1] We set up social coordination in human discourse all the time. We use language to make plans, to describe games or policies, to commit to things, to ask for reports on processes, etc.

Figure 1: Coordination in speech

It's surprising how powerful a "programming" language this is, and how flexible. We can build up a social coordination process like Tinder out of the same directives:

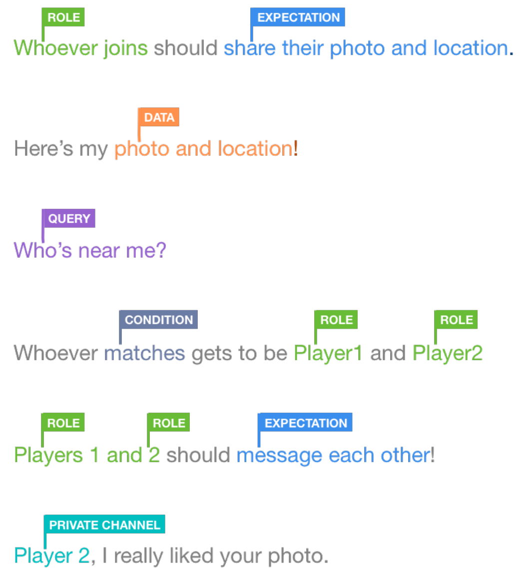


Figure 2: Setting up Tinder in speech

Much can be learned from how humans use speech this way. There is great wisdom in it, encoded into the structures of language and social convention.

In the next two sections, we will see what it's like to build a social programming environment for groups around this way of talking. In section 4, I'll identify challenges with this approach, and in section 5, I'll conclude with the advantages. I'll return to the social ills mentioned in the introduction and see whether this model for social programming can address them.

3 A HABITAT

Object-oriented and functional programming tend start with what the machine understands and abstract towards more human concepts.

In this paper, we'll go in reverse. We'll start with what humans use to set up coordination and find ways to bring the machines on board.

We will make it easy for a machine to recognize and support the discretionary coordination of a group. This leads to a very different way to build social apps–indeed instead of building them as programmers, the users can *take the lead* and use their own *discretion*, casually setting up and tearing down roles, expectations, and conditions as they saw fit.

We could do this using natural language processing, and perhaps that'd be best. But for simplicity's sake, let's start with emoji to help the machine along in understanding our intentions.

3.1 An Example

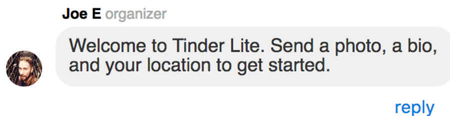
Imagine that we are both in a chatroom, and I send a message like this:

```
Welcome to Tinder Lite.
Send a photo, a bio, and your
```

[1] This all probably owes something to McCarthy's Elephant 2000 [8] and the speech acts literature. But my understanding of what language does is much closer to Charles Taylor's in The Language Animal (2016) [9]. In particular, Taylor describes how we set up and tear down "footings" with regard to one another–kinds of local roles, expectations, modes of discourse, and responsibilities. This is more or less what I'm trying to enable in software.

```
location to get started.
✎( 📎photo 📎bio 📎loc ☑like )
```

It could be interpreted to mean that I expect you to reply in a certain format. So it would appear to you like this:



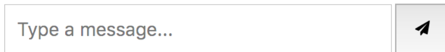


Figure 3: Message with Reply Button

And this notation could help you compose and send the reply, it might look like this:

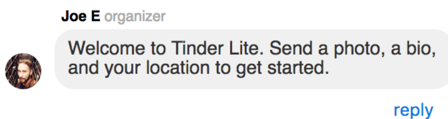


Figure 4: Assisted Composition

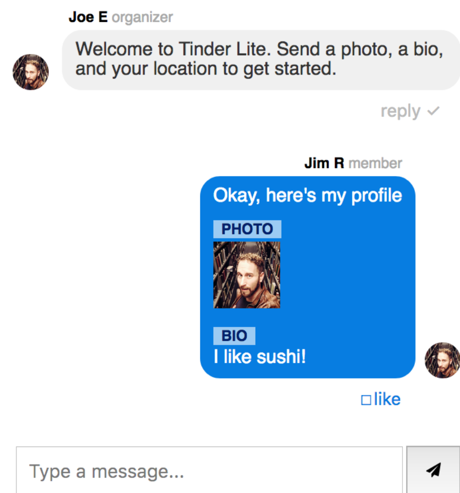


Figure 5: Structured Posting

(Note that the notation ☑like, used in the draft, has become a checkbox in the posted message.)

It'd be helpful if–besides viewing all this as a chatroom–we could also see what has happened in terms of a hierarchical database:

```
/userID1/posted/introMessageID/...
/userID2/posted/bioMessageID/...
/userID2/data/bio/...
/userID2/data/photo/...
/userID2/data/loc/...
/userID1/clicked/like/bioMessageID
```

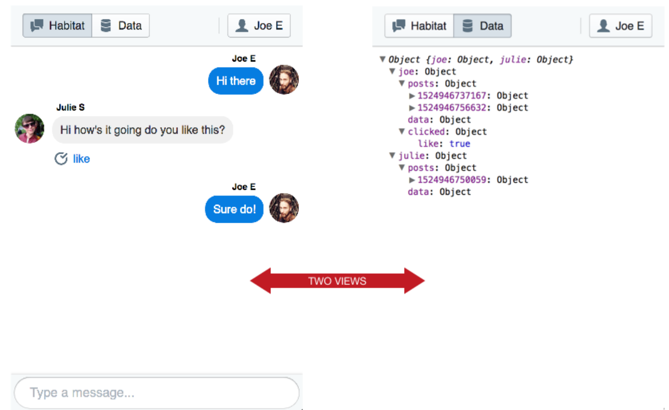


Figure 6: Habitat can be viewed as *a thread of chat messages/actions* or as *a browsable real-time database.*

If I like I can post a live list of who's matched with who so far, using an embedded query on this database:

```
Here are all the matches so far
{
```

```
  $a/posted/$x
  $b/clicked/like/$x
  $b/posted/$y
  $a/clicked/like/$y
}
```

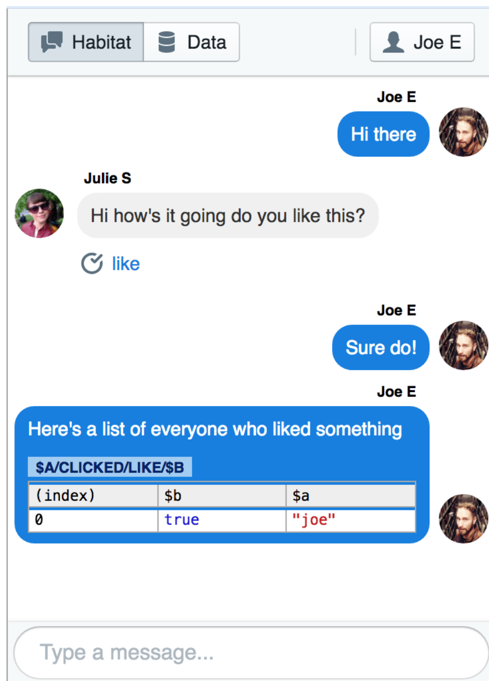


Figure 7: A Live Query

Or send a private message to the people who've matched, with their images attached.

```
{
  $a/posted/$x
  $b/clicked/like/$x
  $b/posted/$y
  $a/clicked/like/$y
}
🔒Hi $a, meet $b. You've matched!
{$a/data/photo} {$b/data/photo}
```

Or even–as in the actual Tinder app–invite matches into a private chatroom just for them.

```
{
  $a/posted/$x
  $b/clicked/like/$x
  $b/posted/$y
  $a/clicked/like/$y
}

🔒Hey 🚪(match $a $b).
You've matched!
```

Putting all this together, we have a very short script for our tinder clone, which can be pasted into any chatroom to add tinder-like functionality:

```
Welcome to Tinder Lite.
Send a photo, a bio, and
your location to get started.
✏️( 📎photo 📎bio 📎loc ☑️like )

---

{
        $a/posted/$x
        $b/clicked/like/$x
        $b/posted/$y
        $a/clicked/like/$y
}

🔒Hey 🚪(match $a $b).
   You've matched!
```

4 THE CORE LANGUAGE AND ENVIRONMENT

I've built a prototype that parses such messages as emergent social software. Once one person posts such a script, the chatroom can continue to evolve as other rules are presented and adopted by members. I call this prototype, and the rich text format it implements, "Habitat".

This may look simple, but I believe these abstractions allow for the collaborative construction of a wide variety of social software, in such a way that non-programmers will be able to understand, alter, accept, or reject the social rules provided.

I'll step you through the main features:

- Roles, mentions, and hashtags (@, #)
- Discreet Messages (🔒)
- Structured Messages (📎, ☑️)
- Embedded queries ({})
- Reply templates (✏️)
- Doors (🚪)
- Dynamic Messages
- Dynamic Doors

And then discuss extensions to support custom UI.

4.1 Roles, mentions, and hashtags

In Habitat, people have user names but also roles. The person who creates a chatroom becomes the chatroom's `@organizer`. Everyone who joins it is a `#member`.

The at sign (@) is used for roles held by one person at a time, where the hash (#) is used for multi-person roles. You can also use the at sign with a username, as on twitter or facebook.

This punctuation is used to scope notifications to the right group.

```
Hey can anyone who's #attending
pick me up on the way to the party?

@organizer, I'm going to be late
```

Roles in the top-level chatroom may be created and joined by posting a message with a checkbox for them:

```
Who's coming to the potluck?
☑#potluck

Sara, do you want take over from me?
☑@teamLeader
```

4.2 Discreet Messages

Messages posted discreetly are only visible to the people who are explicitly mentioned in the message. This is signaled by prepending a lock emoji (🔒) to a message:

```
🔒 Sara, do you want take over from me?
☑@teamLeader
```

4.3 Structured Messages

Paperclips allow us to attach structured data along with a message being composed.

```
@organizer, here's my 📎photo and 📎bio
```

Data may also be collected using checkboxes attached to messages.

```
Who's ☑#attending?
And who can bring ☑pasta ☑salad or ☑wine?

Oh and we need 4 ☑#horsemen
```

When a discreet message has attachments, the data that's collected and posted is scoped so as to be visible only to the message sender and recipients.

```
🔒 @organizer, I have a crush on Anne,
can you seat me with her and pass
her this creepy 📎note?
```

4.4 Embedded Queries

Curly braces allow us to post messages which contain live queries into such structured data:

```
Check out the list of people {#attending}
and everyone's photos
{#attending/data/photo}
```

Of course, such queries display only the data that's visible to each user. To further narrow their scope, a `discreet query` shows only rows that involve the user somehow. So this query would only show the users' own matches:

```
Here are your matches
🔒{
      $a/posted/$x
      $b/clicked/like/$x
      $b/posted/$y
      $a/clicked/like/$y
}
```

Live queries may also be used as part of an attachment, to have the user pick from structured data[2] that's already in the database:

```
Here's my 📎favoritePhoto{*/data/photo}.
```

(Or equivalently)

```
Here's my 📎favoritePhoto{photo}.
```

4.5 Reply Templates

In traditional social software, it's the "app" that asks you, impersonally, to submit structured data. "Tinder" asks you for your profile, "facebook" asks you for data about an event you want to host, etc. In Habitat, these requests come from real people–often the chatroom organizer–who have set up rules that respond to your submitted data.

So the organizer might ask:

```
Which book should we pick for the
book club this month?
✏️( 📎bookChoice{bookOption} )
```

Or anyone might say:

```
Hey @members, I'm going to have a movie night,
what should we watch?
✏️( How about 📎suggestion{movie}? )
```

And thus collect structured responses.

4.6 Doors

Besides discreet messaging, there is another way to have chats between small groups. If you'd like to chat in a small group with a few other members of a chatroom, you can send them a door (🚪)

```
Hey @julie @eric, let's plan here 🚪theproject
```

Doors have names. In this case you've been given a door to "theproject". Door names can also be in parentheses, and can contain structured data.

```
Let's plan the next book club event!
🚪(bookclub 📎date)
```

Finally, when you make a door, you can assign roles that are only be active inside the door.

```
Let's plan the next book club event!
🚪(bookclub 📎date; @host: @julie)
```

[2] In this there are similarities with Chorus [2][6]. From Chorus, I've taken the sensible emphasis on typed data, but set it within a human, time-sensitive message-passing paradigm rather than a timeless data-definition paradigm.

4.7 Dynamic Messages

If a message starts with a query in curly braces, it works a little differently. Rather than making a live query post, the query is used to generate dynamic messages.

```
{
  $a/posted/$x
  $b/clicked/like/$x
  $b/posted/$y
  $a/clicked/like/$y
}
🔒Hi $a, meet $b. You've matched!
{$a/data/photo} {$b/data/photo}
```

Every time there is a new match for the query, a new message will be posted.

This can also be used to generate reminders. I could remind myself to choose a book for our club a few weeks before we meet:

```
{
  $user/data/bookclubDate/$x
  /time/now/($x - 21 days)
}
🔒@self ✏️(
  Great, lets all read 📎book{bookIdea}
  for our meeting on $x.
  Who's ☑️#attending?
)
```

4.8 Dynamic Doors

Dynamic messages can be combined with doors to make subgroups automatically, or subgroups that exist only for a certain time or place, or under certain conditions.

```
{
    $a/posted/$x
    $b/clicked/like/$x
    $b/posted/$y
    $a/clicked/like/$y
}

🔒 Hey, you've matched! 🚪(match $a $b).
```

This mechanism is very flexible. In restaurant management software, waiters might be assigned to tables. In project management software, employees might be associated with projects.

```
We need someone to ☑️project:cleanUpTheLoft

---

{
  $user/clicked/project:$x
}

🔒 $user, thanks for helping with 🚪(project $x).
```

Or to create subgroups where everyone can discuss their favorite band.

```
What was your favorite band as a teenager?
✏️( 🔒📎favoriteBand )

---

{ $user/data/favoriteBand/$x }
🔒 $user 🚪$x
```

5 EXTENSIONS

The above language is simple. There's no keywords, and not much new punctuation (#, @, {}, ✏️, 📎, ☑️, 🚪, 🔒). I believe these components can be used to model a wide variety of social flows, including those from mainstream and business apps.

It seems reasonable to imagine that, by starting with chat and moving through structured data, to queries, to dynamic messaging and doors, many people will find an easy path towards social programming. Indeed, this would recapitulate the way many learned to code in the 1980s and 90s, using MUDs, MUSEs, MOOs, or IRC bots.[10]

The environment can be extended to bring its capabilities closer to those of current social apps. I'll cover two such extensions: custom UI and alternate media.

5.1 Custom UI

In the prototypes above, there are two views of the chatroom available: one of them looks like a chatroom, the other like a hierarchical database.

For tinder prototype, we might like to view the chatroom as a stack of profile cards, either in a random order, or ordered by some algorithm that reacts to the users' previous likes.

We could do this with a kind of plug-in–a `presenter`–which maps from the hierarchical database onto a custom view. Users could then choose to view what's happening as a chatroom, a database, or using a plugin.

Such plugins could be made inspectable and safe; instead of having them operate directly on the database, they could implement declarative UIs atop queries that the user is allowed to inspect. Instead of making changes to the database, they could be limited to composing drafts that the user could inspect before posting.

5.2 Alternate Media

Since everything that happens in Habitat is made of messaging, the above approach to custom UI might also be extended to conversational and audio-based interfaces, like Alexa or Siri, without too much work.

There can be spoken-word or audio presenters as well as screen-based presenters, and the same newly-arriving message data can be presented in different ways depending on the users' state and preferences.

6 CHALLENGES

Habitat exists as a proof-of-concept.[5] There are some challenges to giving it a real life.

1. **Data migration.** As the social script in a chatroom evolves, the database will tend to grow obsolete. Is there an approach to data migration which can work in this environment and which is social rather than authoritative?

2. **Divergence** Could users really cooperate on a set of directives? Would they tend to conflict? If users are allowed to endorse different sets of directives, how would changes end up converging?
3. **Scalability.** Even the simple pattern-matching-based query language used here is difficult to make efficient. And how can it be extended to support aggregations and the like?
4. **Ease of Use.** How can use of the directives and sub-languages be made easier with UX, such as the popups that appear when you type @ on facebook, or the autocomplete components of some IDEs?
5. **Extension.** Can presenter plugins be made secure, sharable, and the like?

7 ADVANTAGES

In the introduction I pointed out some problems with existing social software: there are rigid expectations, impersonal sources of authority, and it's usually impossible to see an an overview of state or of the relationships between people, to evolve wisdom about process, or to step outside the software mediation.

Why do these problems arise with conventional software? Perhaps it's because traditionally, programmers imagine interacting with users as if they were similar to other software components. People aren't objects with APIs, but if we were, then we wouldn't be bothered by rigid expectations, impersonal authority, lack of an overview, and so on.

Perhaps software designers have defaulted to treating people like objects with APIs, and treating group processes (or "*business logic*") like code.

But people need to defy expectations, they need discretion over their own jobs, timing, communication, and visibility, they need an overview (because sometimes they need to take the lead), and people-based processes are always subject to revision and elision.

It's worth going through these differences in detail, and seeing how Habitat addresses them.

1. Humans **need to understand what they're part of**, and software components don't. In Habitat, there's a **global, inspectable, directly-editable state**. Users can always understand why they matched with someone, what they signed up for, and so on, using the database view. The entire social flow and structure of the app is made legible, easy to understand, and editable by users. This includes the specification for which information is collected, from which users, who gets shown what, who gets notified, what all the roles and expectations are, and the timing for all of the above.
2. Humans **need to explore** to understand. In Habitat, users can explore the social consequences of different roles and actions by trying them out. They can accept roles and shed them later, and the environment will figure out how to recover. They can unsend messages. If they like, they can uncheck boxes they previously checked, delete media, etc. Everyone's live queries and pattern matches will update to reflect these changes.
3. Humans **need the discretion to alter plans**. Habitat is **suggestion based**. Instead of specifying exactly what users can do at any time, dynamic messages and doors implement a kind of *soft automation*. Everything scripted *defers* to the humans that are actually working together on a related problem. The user is always free to submit some *other* data, send it to some *other* person, and even to write their own queries, doors, and checkboxes to change roles and rules completely.
4. Humans don't emit technical procedures, rather they **express aims using composable media**. By embedding representations for data collection, querying, and pattern matching, and setting expectations into ordinary human discourse, users have the sense that they are involved in a discussion using a shared medium of expression, rather than instructing a computer what to do. We can refer to this medium as **data-bound, media-independent rich text**.
5. Humans are **motivated by nuanced, adaptable allegiances** to *one another* (not to work itself). So, in Habitat, people are accountable to one another, not to the script. The script is never the source of messages and instructions itself. The script and its author are not an authority, but lend authority to the players insofar as the players accept the roles in the script.
6. Humans **contribute to the social processes they find themselves in**. In Habitat, a user can investigate the dynamic queries behind whatever she sees, see who wrote them, and write alternatives herself which run in parallel in the same group of users and suggest other ways of coordinating.[3] If I send you a door, you can enter and immediately start writing your own rules, changing which information is collected, from which users, who gets shown what, who gets notified, and what all the roles and expectations are.
7. Finally, humans have bodies and **operate across many channels and media**. Habitat doesn't assume everything will be done through it's screen-based interfaces. *The same script lines can automate interactions across several mediums*: text chat, video chat, VR, and most importantly just crossing the room to talk with your colleagues. Even if apps have custom UX designed only for screens, other interfaces can update the same data.

For all of these reasons, I believe traditional approaches to coding social software and information systems are inhumane. Computer programs must no longer hardcode people's social interactions or roles. Instead, they should provide a medium-independent way for users to do whatever they want with other people, and limit themselves to making suggestions for how the user could proceed; suggestions that the user can modify. Systems like Habitat show one way to do so.[4]

ACKNOWLEDGMENTS

CEML was made possible by the investors in my failed startup, Groundcrew, and my work here was partly funded by a grant from Stripe.

[3]This environment descends from a previous project of mine, CEML[4]–which in turn descends from workflow programming languages like FlowMark, Lotus Notes, and so on. CEML was used by non-programmers, and this was encouraging. But with CEML I also saw how easy it is for people to design their processes as funnels rather than playgrounds, and how hard it is to work around that in a programming environment. Many of the design choices here are to prevent this tendency.

[4]Another approach is to leave the negotiation of social roles out-of-band, in the real world. This can work for in-person teams. See Dynamicland[7]

I am grateful to Joshua Schacter, Rob Ochshorn, and Bret Victor for conversations about social coordination and non-screen-based programming environments. And to the program committee and organizers of the PX! workshop for their comments and feedback.

REFERENCES

[1] Marc Andreessen. 2011. Software is Eating the World. (2011). http://www.wsj.com/articles/SB10001424053111903480904576512250915629460
[2] Jodie Lian Chen. 2017. *Chorus: End User Programming of Social Applications*. Master's thesis. MIT. http://www.chorus-home.org/jodie_thesis.pdf
[3] Patrick Collison. 2017. Tweet. (2017). https://twitter.com/patrickc/status/705039375852130304
[4] Joe Edelman. 2011. CEML Github Repo. (2011). https://github.com/jxe/ceml
[5] Joe Edelman. 2018. Habitat Github Repo. (2018). https://github.com/jxe/habitat
[6] Jonathan Edwards, Jodi Chen, and Alessandro Warth. 2016. Live end-user programming: a demo/manifesto. In *Proceedings of LIVE 2016*. http://www.chorus-home.org/LIVE16.pdf
[7] Bret Victor et al. 2018. Dynamicland. (2018). https://dynamicland.org/
[8] John McCarthy. 1989. *Elephant 2000: A Programming Language Based on Speech Acts*. Technical Report.
[9] Charles Taylor. 2016. *The Language Animal: The Full Shape of the Human Linguistic Capacity*. Belknap Press: An Imprint of Harvard University Press. https://www.amazon.com/Language-Animal-Shape-Linguistic-Capacity/dp/067466020X?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=067466020X
[10] Wikipedia. 2018. MOOs. (2018). https://en.wikipedia.org/wiki/MOO