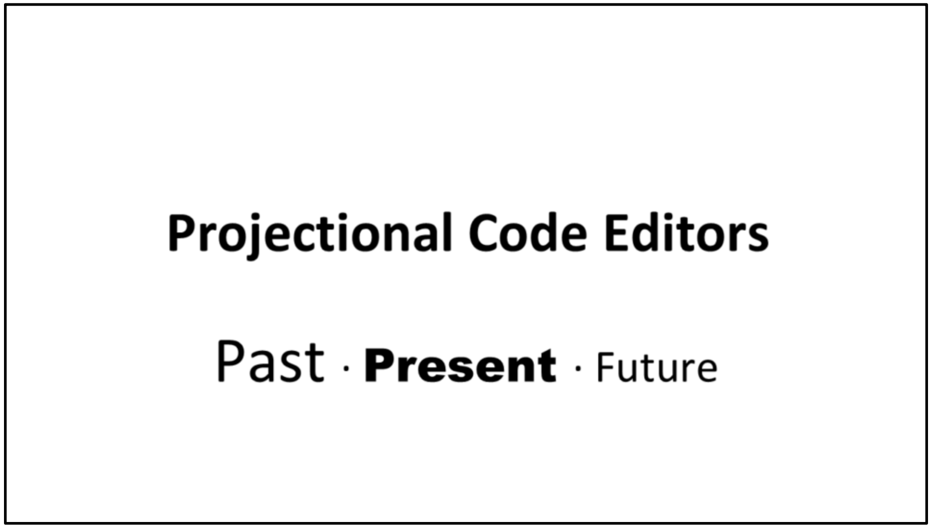


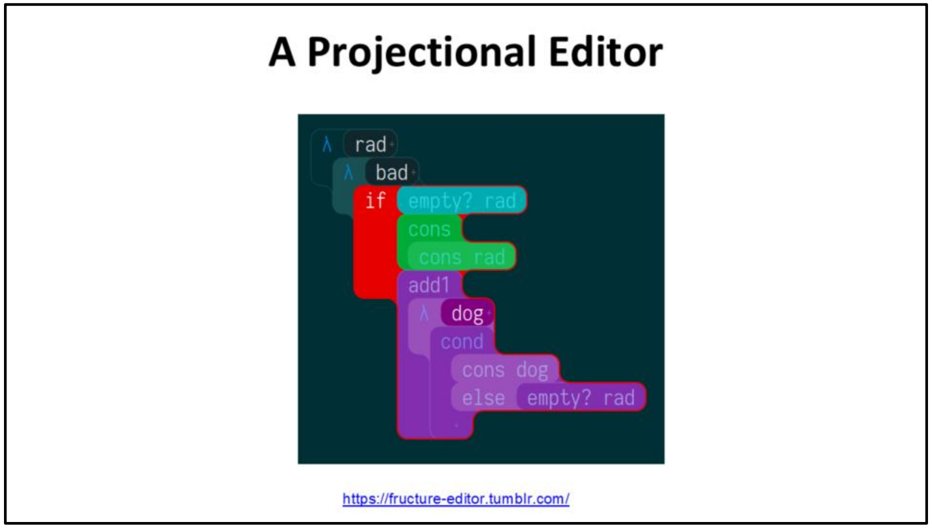
# Projectional Code Editors

Past · **Present** · Future

A friend once told me that the best way to learn about a subject is to sign up to deliver a presentation on the subject.  It's really tremendous advice, my whole understanding of what a projectional editor even is what substantially influenced by the research I did for this talk.

To give you a definition: A projectional code editor is an editor that understands that code is more than just text.  The tools that many of us work in day-to-day are "text editors", which could also theoretically be used to write forms of text, such as a poem or news article or something.  For instance, I wrote much of this talk in Emacs.  Projectional editors use their understanding of programming languages to prevent you from making certain kinds of errors; usually syntax errors but sometimes semantic errors.

Let me show you an example of an editor that is not a text editor.



## A Projectional Editor

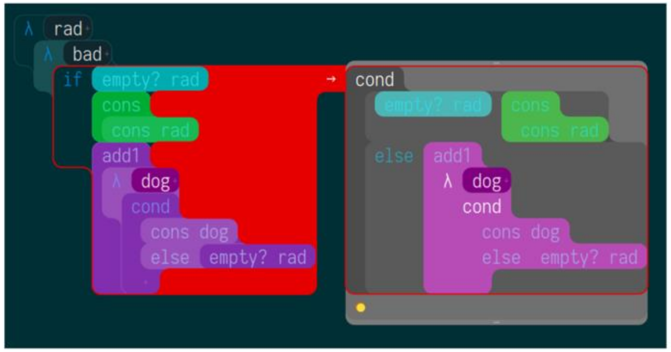
```
λ rad
  λ bad
    if empty? rad
       cons
        cons rad
       add1
        λ dog
           cond
             cons dog
             else empty? rad
```

https://fructure-editor.tumblr.com/

This is Fructure made by Andrew Blinn for the programming language Racket. I want to highlight a number of salient features visible in this screenshot:

1. You'll notice the colors. They're a bit garish, admittedly, but what they're doing is highlighting the different semantic portions of the code. You can see the entire `if` expression surrounded in red, the condition in teal, the "then" branch in green, and the "else" branch in purple.
2. The if statement is selected. If you press enter...

# A Projectional Editor

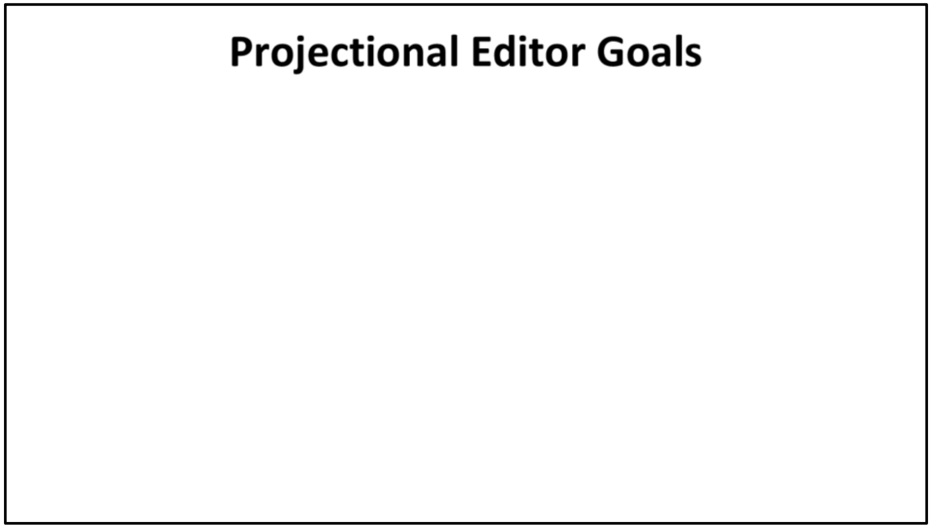


```
λ rad
  λ bad
    if empty? rad                        → cond
       cons                                  empty? rad   cons
         cons rad                                           cons rad
       add1                                  else  add1
         λ dog                                       λ dog
           cond                                      cond
             cons dog                                  cons dog
             else empty? rad                           else  empty? rad
```

https://fructure-editor.tumblr.com/

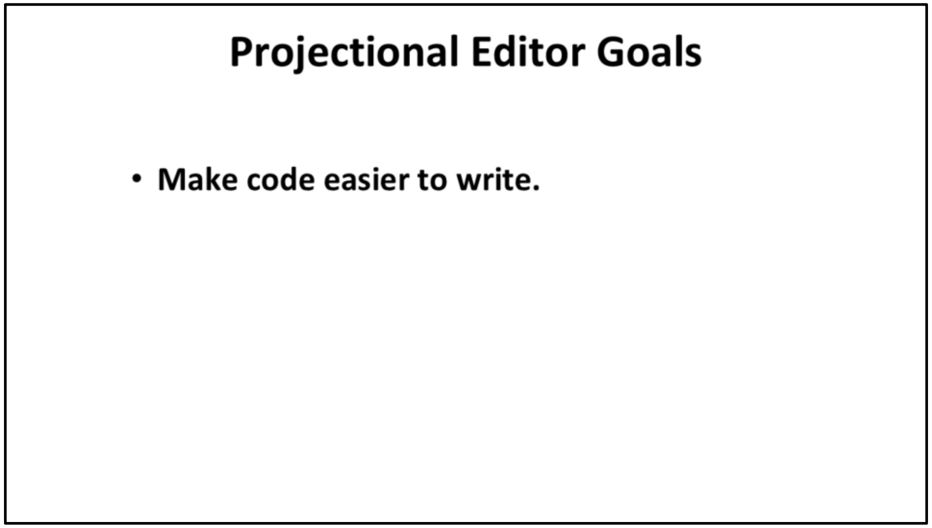
...Fructure shows valid replacements for the `if` expression, in this case an equivalent `cond` expression. Fructure only allows valid transformations of the code. For instance, just deleting the “f” of “if” is not a thing that Fructure allows you to do.

This approach to projectional editing is *far* from the only one. We’ll explore quite a few today!



## Projectional Editor Goals

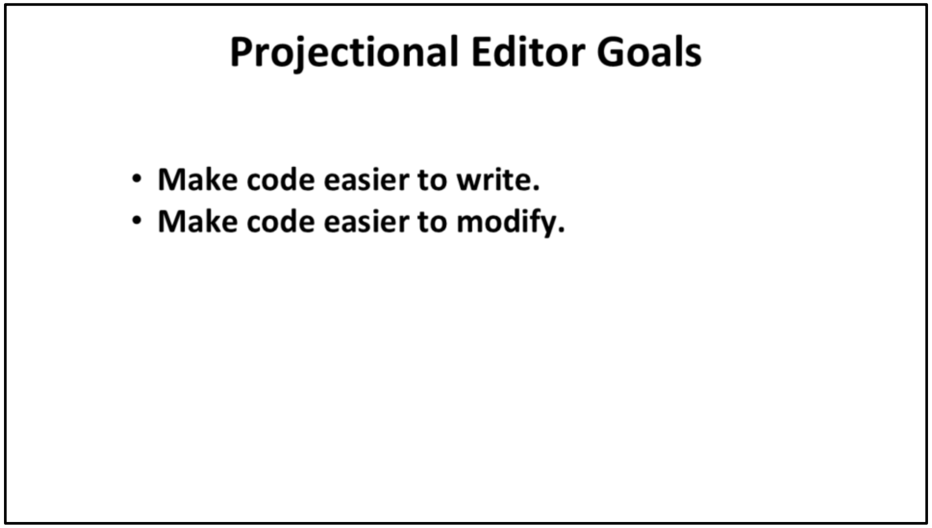
And so, the natural next question is “why is this useful?”  The truth of the matter is that programming is *hard*.  We laud ourselves on being able to grapple with the arcana of syntax and semantics, the patterns and idioms of our languages, but what if it wasn’t such a struggle?



## Projectional Editor Goals

- **Make code easier to write.**

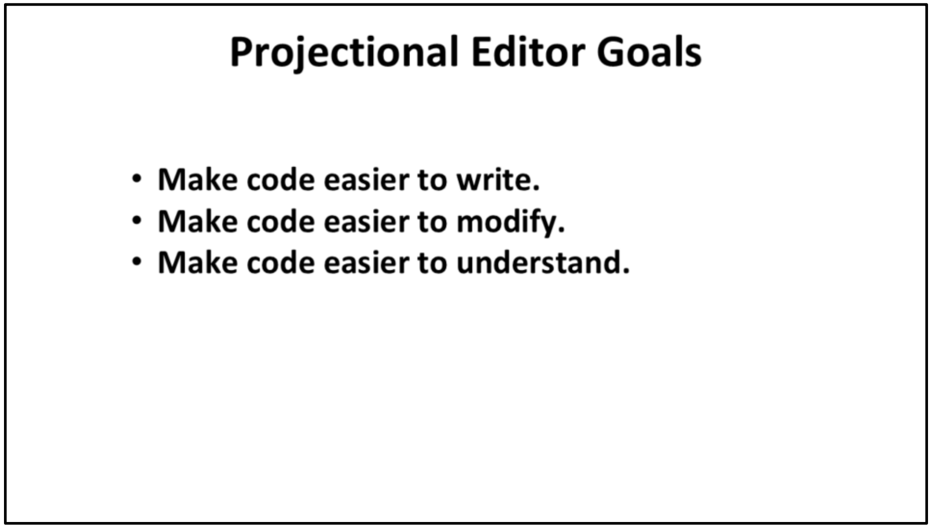
As I mentioned before, projectional editors act like guardrails preventing you from making certain classes of mistakes. This is especially useful for beginners, and we'll talk about that more later.



## Projectional Editor Goals

- Make code easier to write.
- Make code easier to modify.

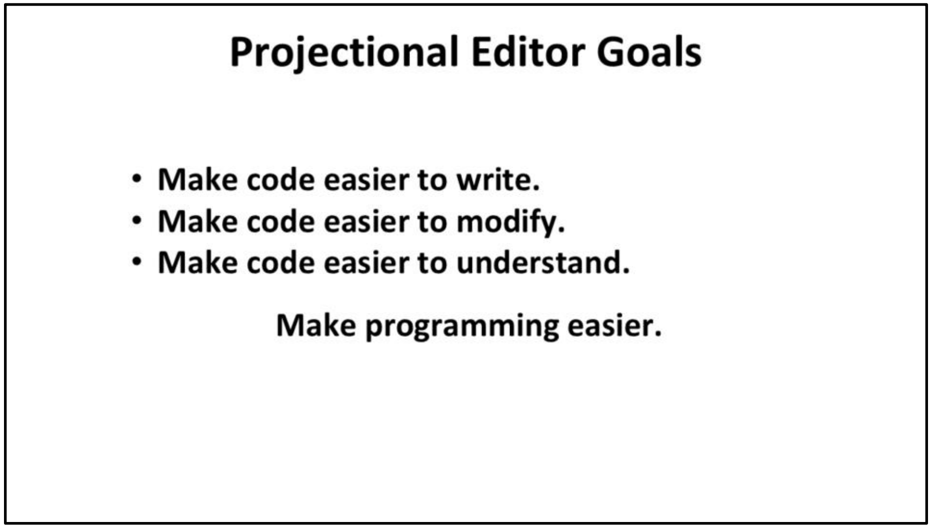
Let’s face it, we’re modifying code more often than writing it from scratch. An editor that only allowed you to write new code but didn’t allow you to edit code you’ve already written would be pretty rubbish. And the thing of it is, modifying existing code is usually much more difficult than writing new code! You have to ensure that your new code hasn’t violated a contract anywhere, such as creating a new type mismatch.



## Projectional Editor Goals

- Make code easier to write.
- Make code easier to modify.
- Make code easier to understand.

And, lastly, projectional editors strive to make code easier to understand. Remember how Fructure highlighted different parts of the expression different colors? It does that to help you understand the code at a glance. And this is important, because it’s often bandied about that developers only spend maybe 10% of their time writing code, but considerably more reading it, so making code easier to read is a laudable goal!



## Projectional Editor Goals

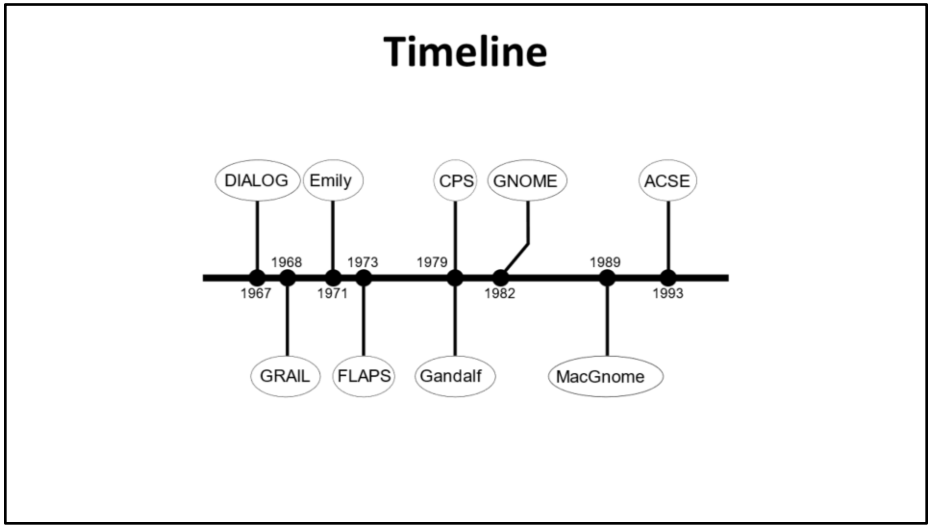
- Make code easier to write.
- Make code easier to modify.
- Make code easier to understand.

**Make programming easier.**

And these elements can be summarized into a single goal: Make programming easier.  Like I said before, programming is *hard*.   Many people are intimidated by the perceived difficulty of learning to program, and projectional editors strive to shallow that learning curve.  Projectional editors also strive to make us professional programmers more productive.

# The Past

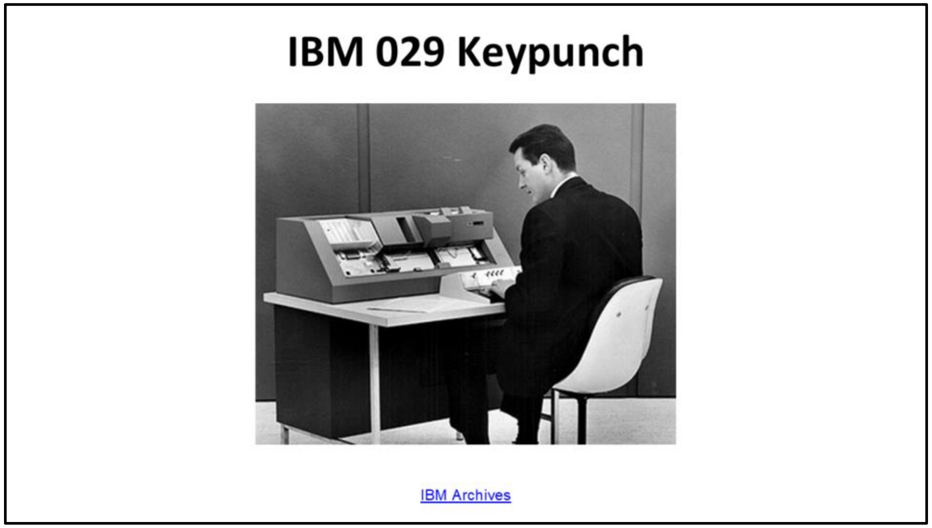
Before I show you the first projectional editor that I could find reference to, I want to set the stage a little bit.  You see, when I first had an idea for a projectional editor, in 2016, I thought it was a totally novel idea; that nobody had ever thought of this before.



## Timeline

Little did I know that projectional editors have had a long and illustrious history.

Anyways, the year in 1967.

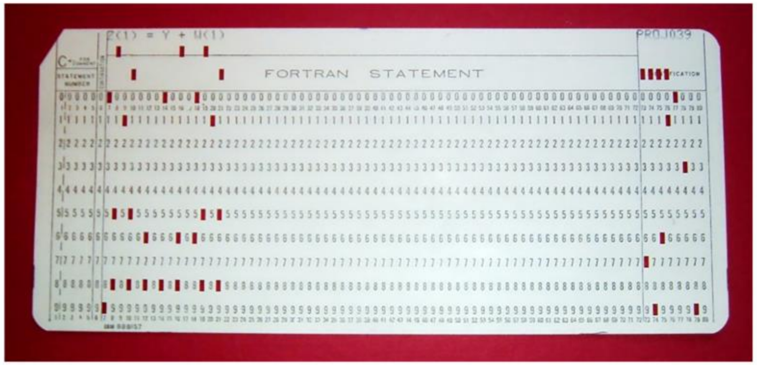


# IBM 029 Keypunch

IBM Archives

This is the IBM 029 keypunch.  You would stack up some punched cards and start typing on the typewriter-like mechanism there, and the machine would punch your cards for you.  COBOL’s original language design was totally based around the IBM-standard 80-column punched cards.  The card being punched is there in the middle.

# IBM 888157 Punched Card



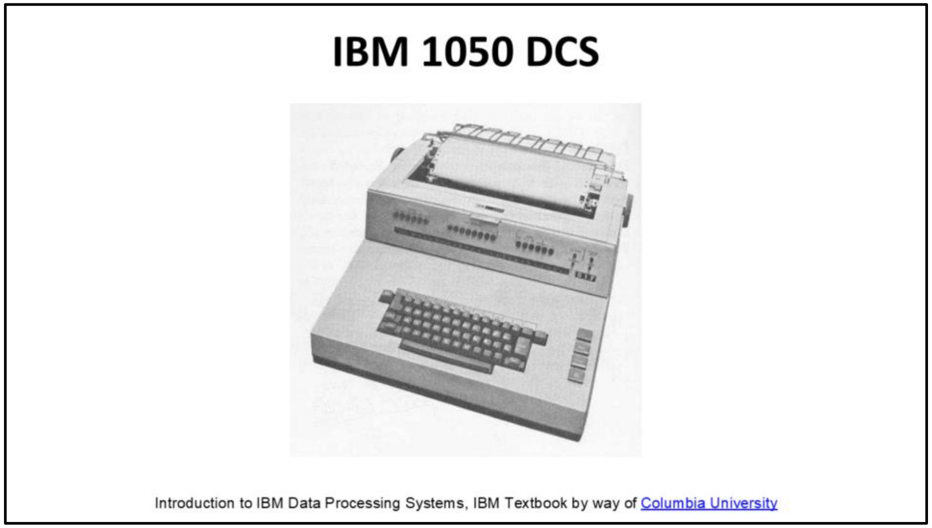
Arnold Reinhold via Wikimedia Commons

Here’s one of those punched cards, an IBM 888157 FORTRAN statement card. You can see that the keypunch also wrote the characters being typed in ink at the top of the card, which is nice.

# IBM System/360

Wikimedia Commons by Joe Mabel

And once you punched your cards, you'd carry them to the computer, like this IBM System/360 line of computers, IBM's most successful line of machines at this time.  On your right is a punched card reader, how the machine would accept input.  On the left is a line printer, which would print your output on these huge sheets of paper.

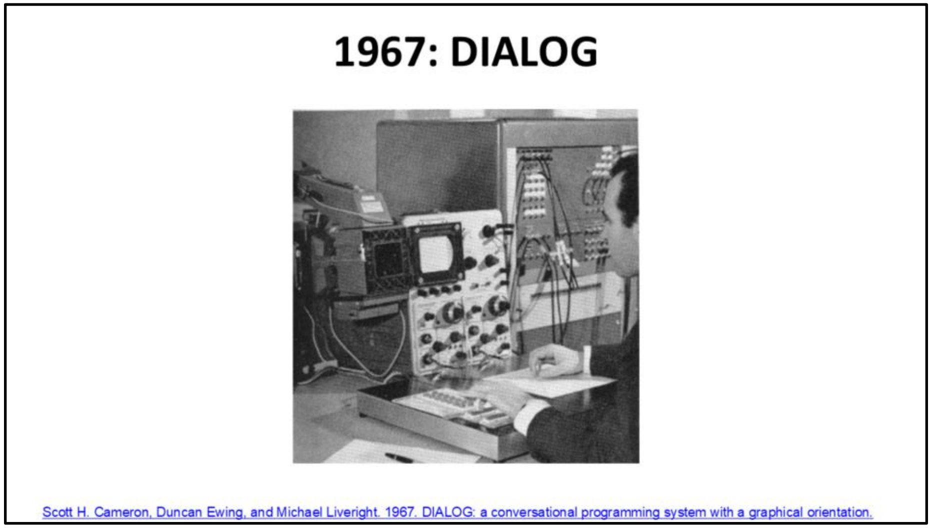


# IBM 1050 DCS

Introduction to IBM Data Processing Systems, IBM Textbook by way of Columbia University

System/360 was also compatible with the IBM 1050 Data Communications System, one of the first computer terminals. Using this device, you could type a line of input and send it to the machine, and the machine could send a response back to you and it would be printed there on the paper.

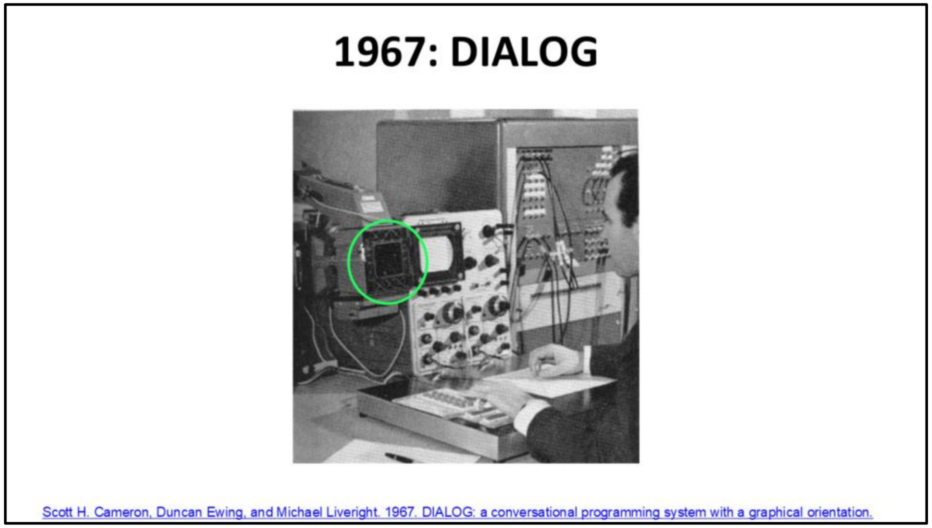
Alright, now that you hopefully have a concept of what computing was like in 1967, let’s look at our first projectional editor.



# 1967: DIALOG

Scott H. Cameron, Duncan Ewing, and Michael Liveright. 1967. DIALOG: a conversational programming system with a graphical orientation.

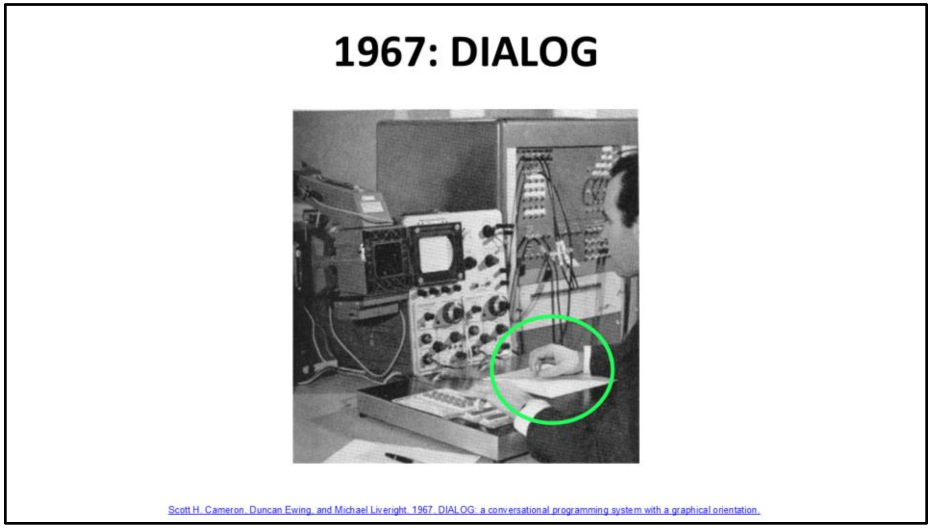
The first projectional editor I could find was DIALOG from IIT Research Institute, in 1967. DIALOG, the editor, was built for programming a language also named DIALOG. Before we get into it, I want to point out a few interesting tidbits. Can you find the screen for this system?



# 1967: DIALOG

Scott H. Cameron, Duncan Ewing, and Michael Liveright. 1967. DIALOG: a conversational programming system with a graphical orientation.

It’s there.  Another interesting bit, there’s no keyboard for this system.  The user interacted with it by moving a “stylus” around on a pad and pressing a button to “click.”  The paper about this system was published at the same time Doug Engelbart was granted his patent for the X-Y Position Indicator, what we now know as the computer mouse.  Can you spot the stylus?

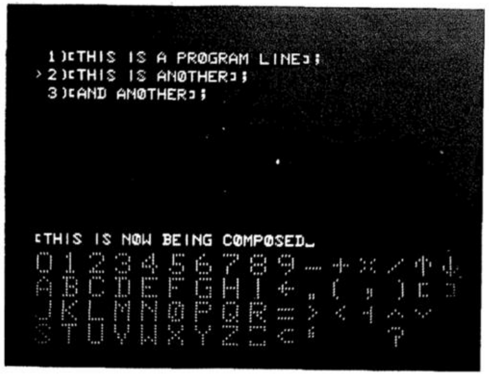


# 1967: DIALOG

Scott H. Cameron, Duncan Ewing, and Michael Liveright. 1967. DIALOG: a conversational programming system with a graphical orientation.

It’s this thing he’s got in his fingers.  It’s connected by strings to cams such that when you move it around the pad, the cams tell the computer where it is.  Alright, let’s look at the interface.

# 1967: DIALOG



```
 1)[THIS IS A PRØGRAM LINE];
>2)[THIS IS ANØTHER];
 3)[AND ANØTHER];


[THIS IS NØW BEING CØMPØSED_
0 1 2 3 4 5 6 7 8 9 — + × / ↑ ↓
A B C D E F G H I ← , ( ; ) [ ]
J K L M N Ø P Q R ≡ > < ⊣ ^ v
S T U V W X Y Z ⊔ ⊂ "     ?
```

Scott H. Cameron, Duncan Ewing, and Michael Liveright. 1967. DIALOG: a conversational programming system with a graphical orientation.

Here's a sample of what you would see on that tiny screen.  At the top are three lines that were previously written, and in the bottom-middle is another line being written.

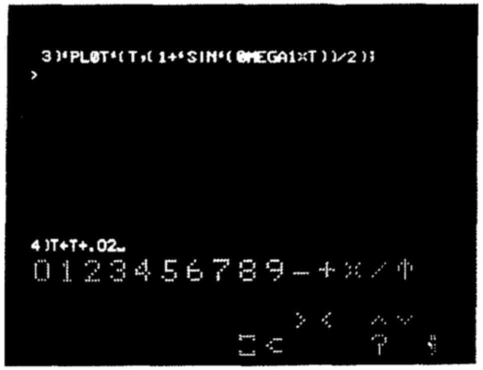
If you wanted to add the current line you were writing to your program, you'd end it with a semicolon.  If you didn't end the line with a semicolon, it was evaluated immediately and discarded.  A sort of marriage of a text editor and a REPL.  I think that's kind of lovely.  I believe MATLAB does something vaguely similar.

Also, this editor predates Vi by about a decade.  Just saying.

You'll notice the letters, numbers, and symbols at the bottom.  That's effectively an on-screen keyboard; you'd move the cursor over the letter you wanted, and press the interrupt button to type it.

What makes this system relevant for us is that this on-screen keyboard limited what characters you were allowed to type.  Here, everything is permitted because we're in a source code comment.

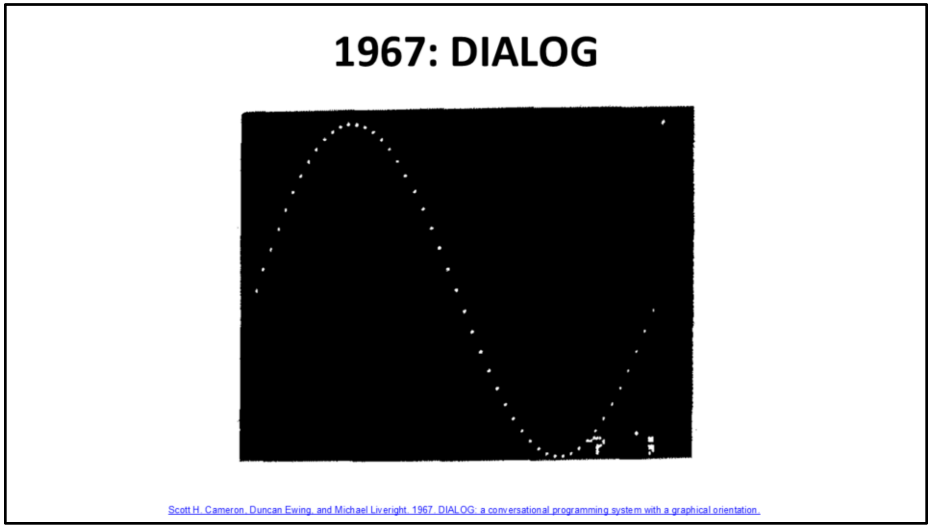
# 1967: DIALOG



```
 3)'PLØT'(T,(1+'SIN'(ØMEGA1×T))/2);
>

4)T←T+.02_
0123456789-+×/↑
                      > <   ^ v
                 □ <        ?   ‡
```

Scott H. Cameron, Duncan Ewing, and Michael Liveright. 1967. DIALOG: a conversational programming system with a graphical orientation.

However, here's a different line. We're assigning to the variable 'T' the value of 'T' plus .02, and, as you can see, we can continue this line by adding more digits or doing some arithmetic. There's a few other things here that are considered valid, like the less-than and greater-than that don't quite seem to make sense.

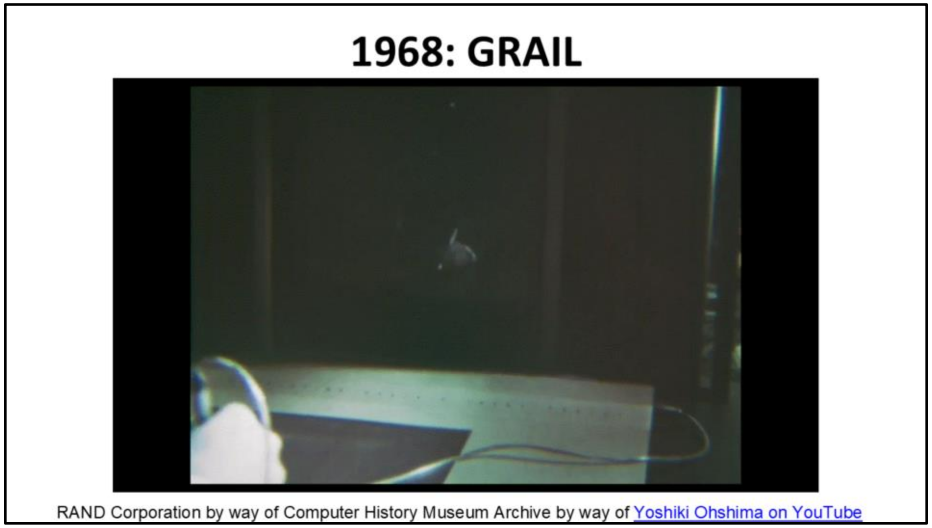


# 1967: DIALOG

Scott H. Cameron, Duncan Ewing, and Michael Liveright. 1967. DIALOG: a conversational programming system with a graphical orientation.

This system seems to have been an early graphing calculator, a thing that notably didn't exist at the time, and probably was vaguely valuable.

Part of the trouble with DIALOG was that it required hardware that there was probably only a single instance of in the entire world, like that pseudo-mouse thing.  I also imagine typing with the "stylus" wasn't very much fun either.  The ideas of DIALOG sure didn't die—and are with us today—so let's keep moving.

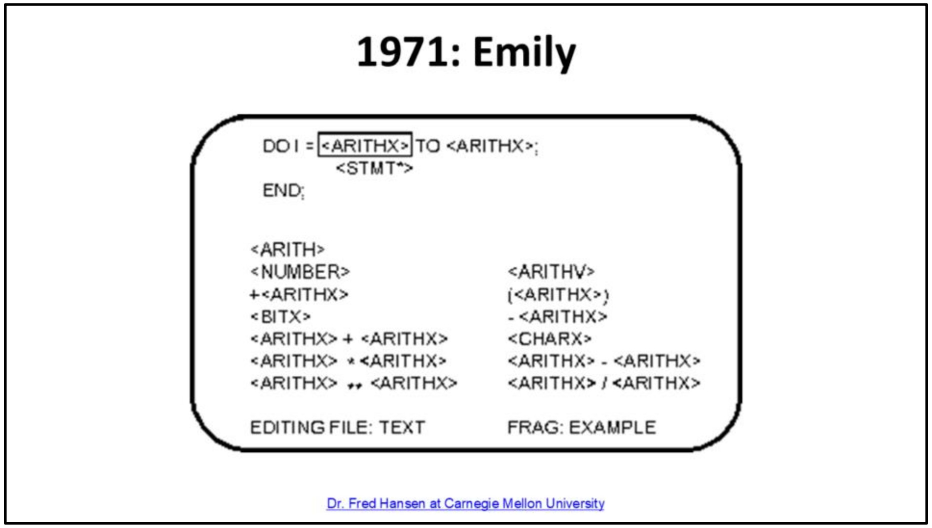


## 1968: GRAIL

RAND Corporation by way of Computer History Museum Archive by way of Yoshiki Ohshima on YouTube

Alright, it’s 1968, and RAND Corporation has created the GRAphical Input Language, GRAIL. This system featured a visual programming language. The paper says, and I quote, “The project deals with the problem of computer programming using flowcharts as a starting point from which to investigate man-machine communications...” The practicability of flowchart-driven development is still contentious today, and we’ll see some modern systems that have their own take on the paradigm.

This was all part of a grant application to the Department of Defense, which I think was ultimately not granted. There were also a few troubles with this system. In writing about the limitations of the system, RAND Corporation writes that the computer just wasn’t fast enough to keep up. That’s probably true, but handwriting tiny characters seems like a pain also. Let’s keep moving.



# 1971: Emily

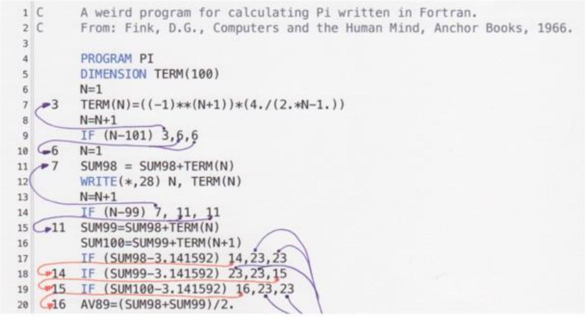
```
  DO I = <ARITHX> TO <ARITHX>;
         <STMT*>
  END;


<ARITH>
<NUMBER>                  <ARITHV>
+<ARITHX>                 (<ARITHX>)
<BITX>                    - <ARITHX>
<ARITHX> + <ARITHX>       <CHARX>
<ARITHX> * <ARITHX>       <ARITHX> - <ARITHX>
<ARITHX> ** <ARITHX>      <ARITHX> / <ARITHX>

EDITING FILE: TEXT        FRAG: EXAMPLE
```

Dr. Fred Hansen at Carnegie Mellon University

1971 brings us Fred Hansen's PhD dissertation, "Creation of Hierarchic Text with a Computer Display", detailing an editor named Emily.

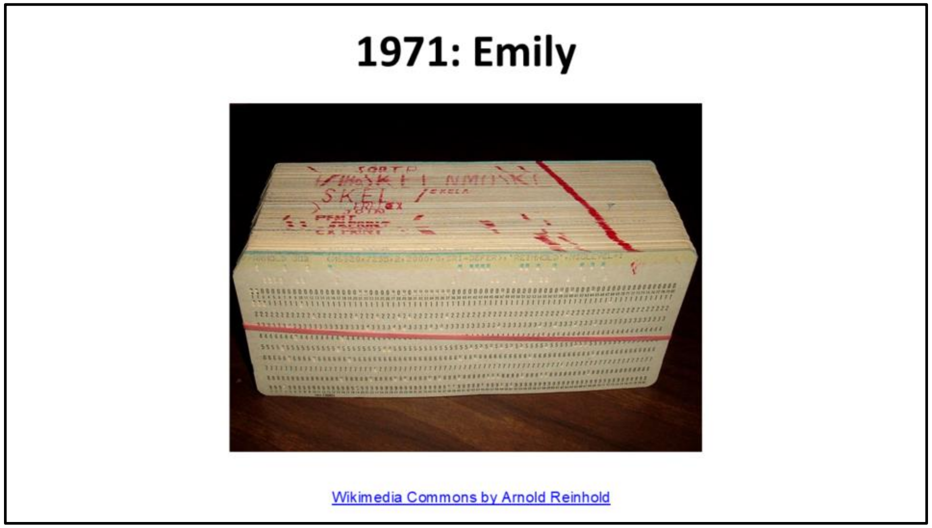
Reading Hansen's dissertation today is really wonderful, because those of us who care about projectional editors say all the same things, but the references are a bit dated.  For instance, Hansen writes "Paper and pencil, the traditional tools for creation of computer programs...

# 1971: Emily



```
C     A weird program for calculating Pi written in Fortran.
C     From: Fink, D.G., Computers and the Human Mind, Anchor Books, 1966.

      PROGRAM PI
      DIMENSION TERM(100)
      N=1
   3  TERM(N)=((-1)**(N+1))*(4./(2.*N-1.))
      N=N+1
      IF (N-101) 3,6,6
   6  N=1
   7  SUM98 = SUM98+TERM(N)
      WRITE(*,28) N, TERM(N)
      N=N+1
      IF (N-99) 7, 11, 11
  11  SUM99=SUM98+TERM(N)
      SUM100=SUM99+TERM(N+1)
      IF (SUM98-3.141592) 14,23,23
  14  IF (SUM99-3.141592) 23,23,15
  15  IF (SUM100-3.141592) 16,23,23
  16  AV89=(SUM98+SUM99)/2.
```

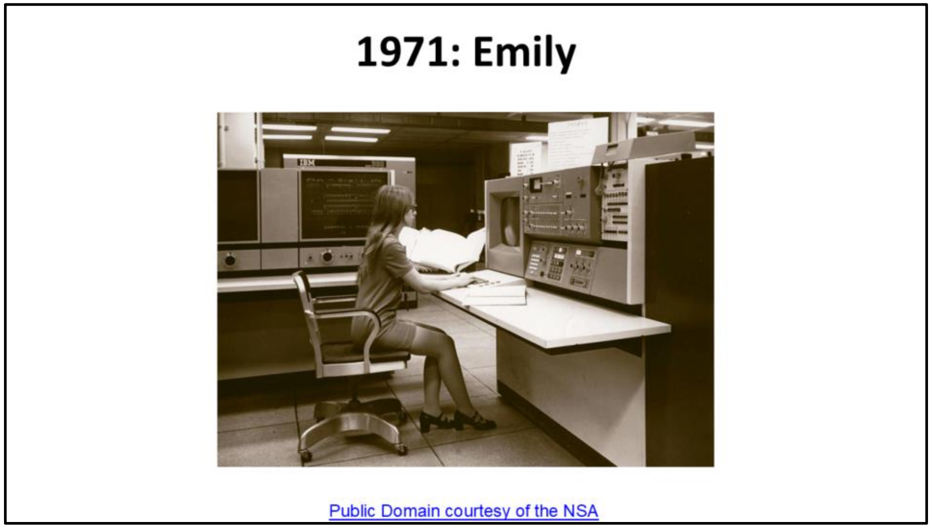
...assist the programmer very little. Proper punctuation demands precision, review of existing text requires clumsy paper shuffling...



## 1971: Emily

Wikimedia Commons by Arnold Reinhold

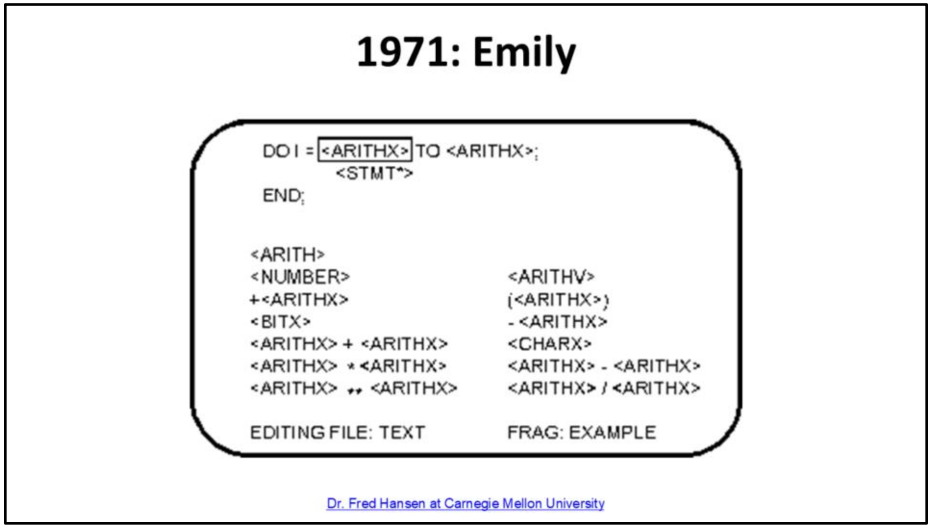
text modification is difficult and messy. In conjunction with a file storage device and a graphic display unit...



# 1971: Emily

Public Domain courtesy of the NSA

...a computer can provide a more flexible medium, but early systems still treated text as an unstructured string of characters.  Emily, the system described in this paper...



# 1971: Emily

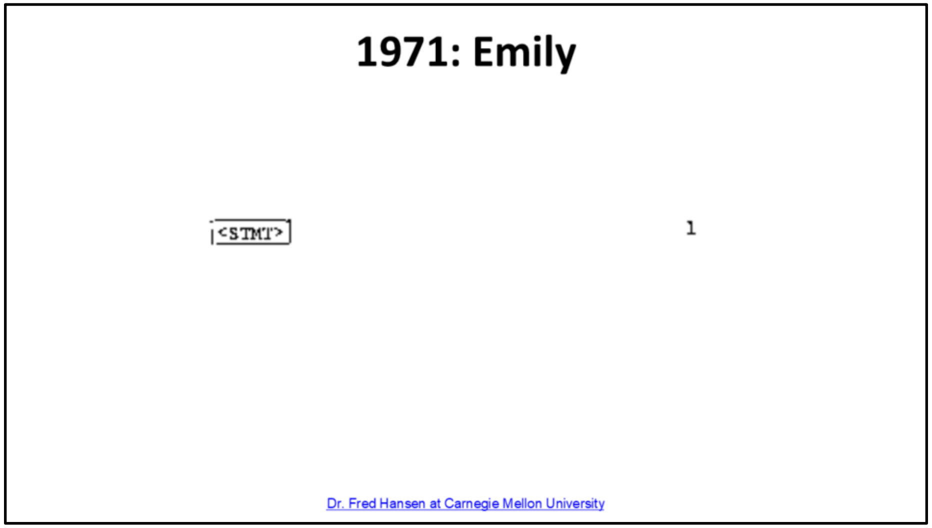
```
 DO I = <ARITHX> TO <ARITHX>;
        <STMT*>
 END;

<ARITH>
<NUMBER>                    <ARITHV>
+<ARITHX>                   (<ARITHX>)
<BITX>                      - <ARITHX>
<ARITHX> + <ARITHX>         <CHARX>
<ARITHX>  * <ARITHX>        <ARITHX> - <ARITHX>
<ARITHX>  ** <ARITHX>       <ARITHX> / <ARITHX>

EDITING FILE: TEXT          FRAG: EXAMPLE
```

Dr. Fred Hansen at Carnegie Mellon University

...avoids these problems because text is created, viewed, and modified in terms of the structure imposed by the syntax of the programming language."

Specifically, Emily was made use of what we today refer to as a "typed hole." The term "ARITHX" that is highlighted, indicated that some kind of arithmetic expression was expected. At the bottom of the interface were possible things that we could fill that hole with; a number, a different kind of expression, an addition, etc.

The paper provided an example:

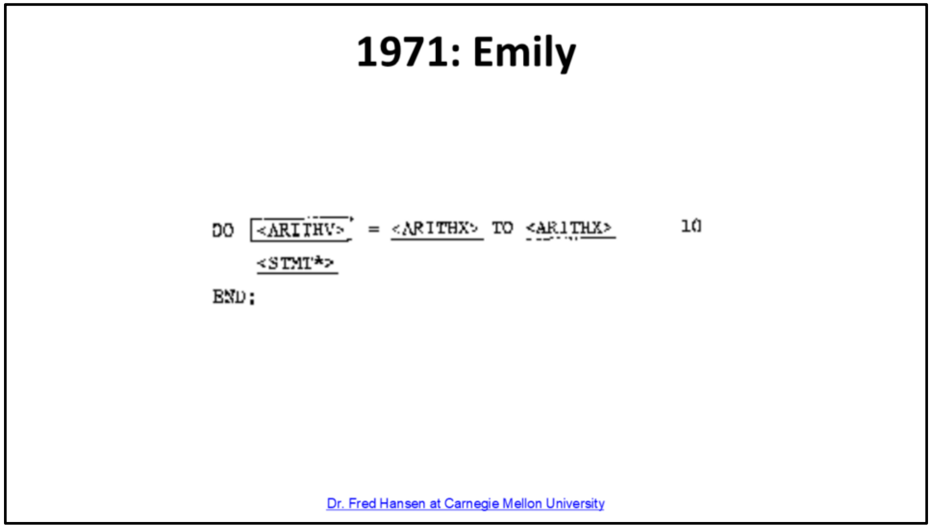


# 1971: Emily

```
<STMT>                                          1
```

Dr. Fred Hansen at Carnegie Mellon University

First, we start with a statement-type hole

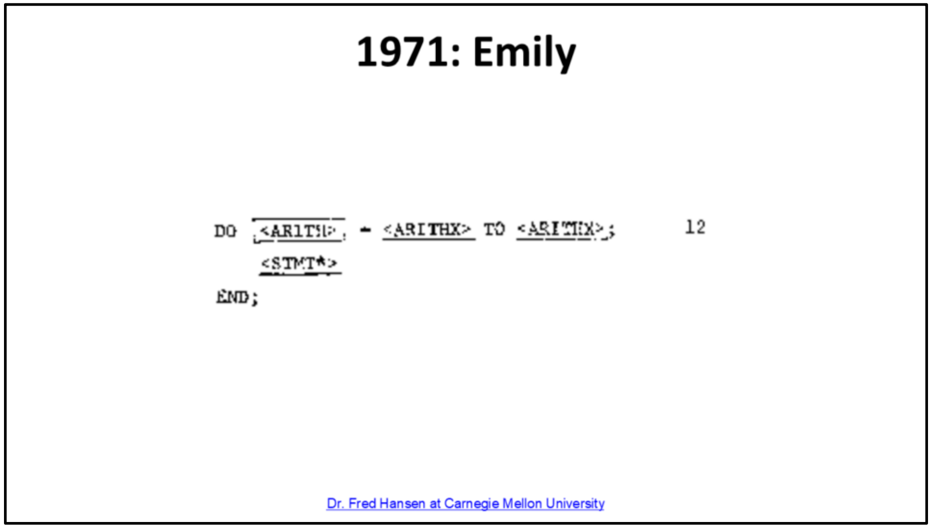


# 1971: Emily

```
DO <ARITHV>  = <ARITHX> TO <ARITHX>        10
    <STMT*>
END;
```

Dr. Fred Hansen at Carnegie Mellon University

Which we replace with a loop construct, with its various holes

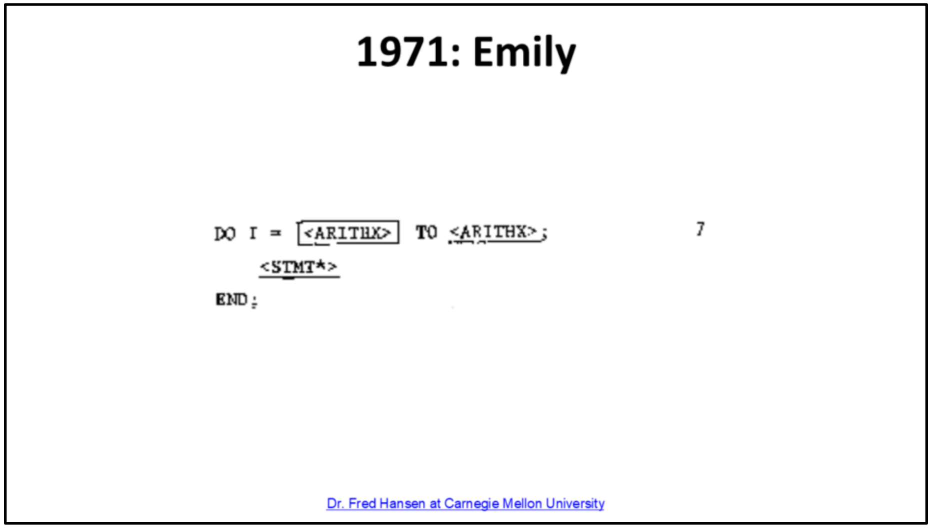


# 1971: Emily

```
DO <ARITHV> = <ARITHX> TO <ARITHX>;        12
    <STMT*>
END;
```

Dr. Fred Hansen at Carnegie Mellon University

Well, we’ve got to change the type of thing that fits in that first hole, can’t tell you why

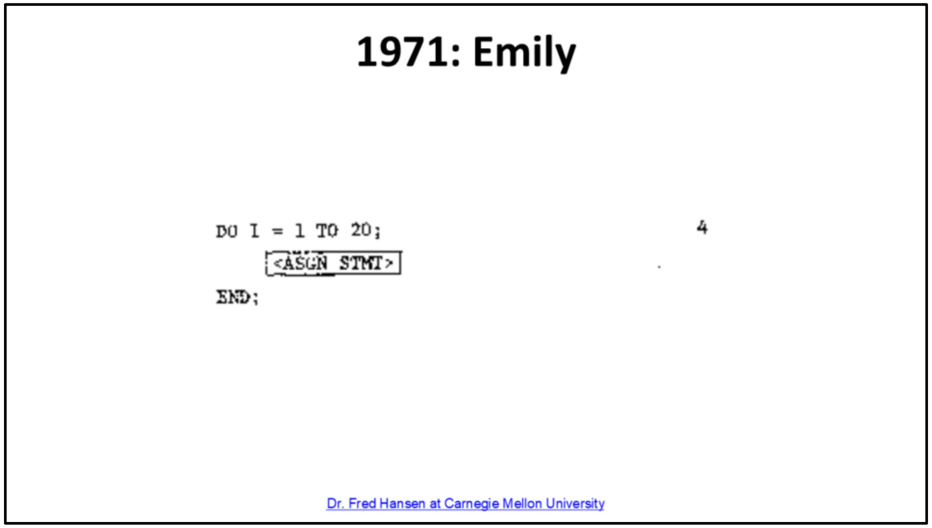


# 1971: Emily

```
DO I = <ARITHX> TO <ARITHX>;                7
    <STMT*>
END;
```

Dr. Fred Hansen at Carnegie Mellon University

Finally, we’ve been able to fill that hole with our loop variable, ‘I’, great, now onto the lower bound

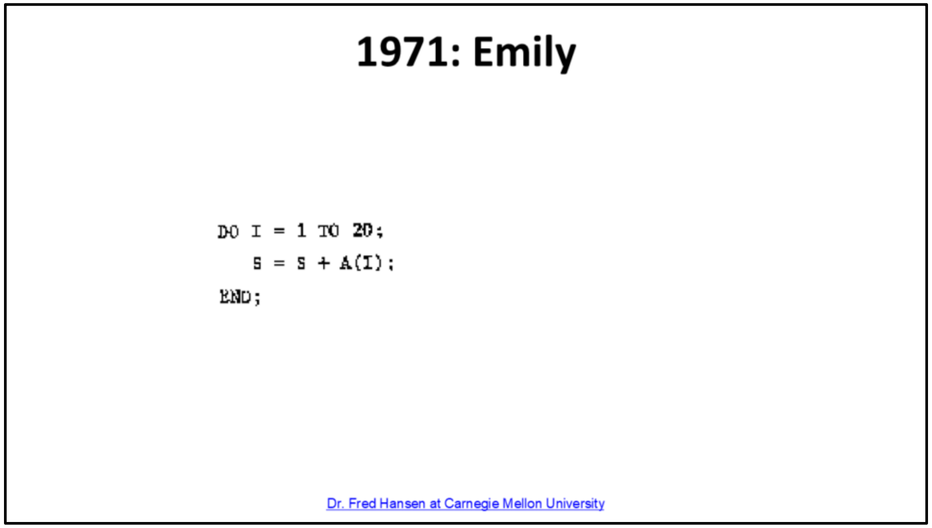


# 1971: Emily

```
DO I = 1 TO 20;                                4
    <ASGN STMT>
END;
```

Dr. Fred Hansen at Carnegie Mellon University

And we skip five steps and now we’ve got upper and lower bounds



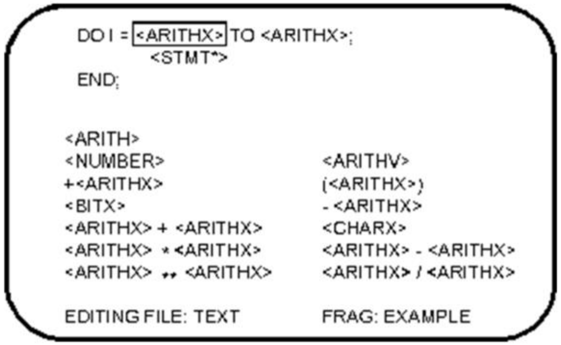
# 1971: Emily

```
DO I = 1 TO 20;
   S = S + A(I);
END;
```

Dr. Fred Hansen at Carnegie Mellon University

And 10 steps later we’ve drawn the rest of the owl.

# 1971: Emily



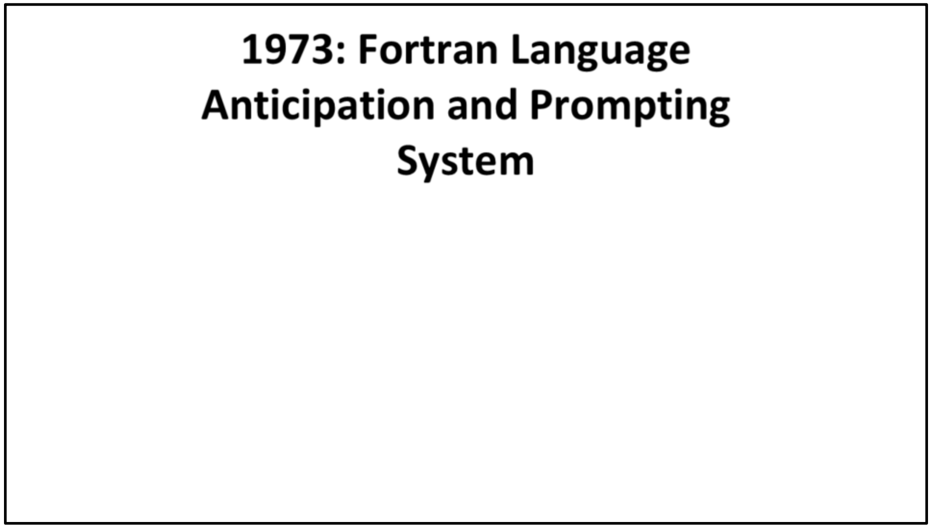
```
DO I = <ARITHX> TO <ARITHX>;
      <STMT*>
END;

<ARITH>
<NUMBER>                    <ARITHV>
+<ARITHX>                   (<ARITHX>)
<BITX>                      - <ARITHX>
<ARITHX> + <ARITHX>         <CHARX>
<ARITHX> * <ARITHX>         <ARITHX> - <ARITHX>
<ARITHX> ** <ARITHX>        <ARITHX> / <ARITHX>

EDITING FILE: TEXT          FRAG: EXAMPLE
```

Dr. Fred Hansen at Carnegie Mellon University

Dr. Hansen, on his website, very plainly discusses the limitations of Emily. Namely, that the system was slow to use, and hopefully we can all see why.

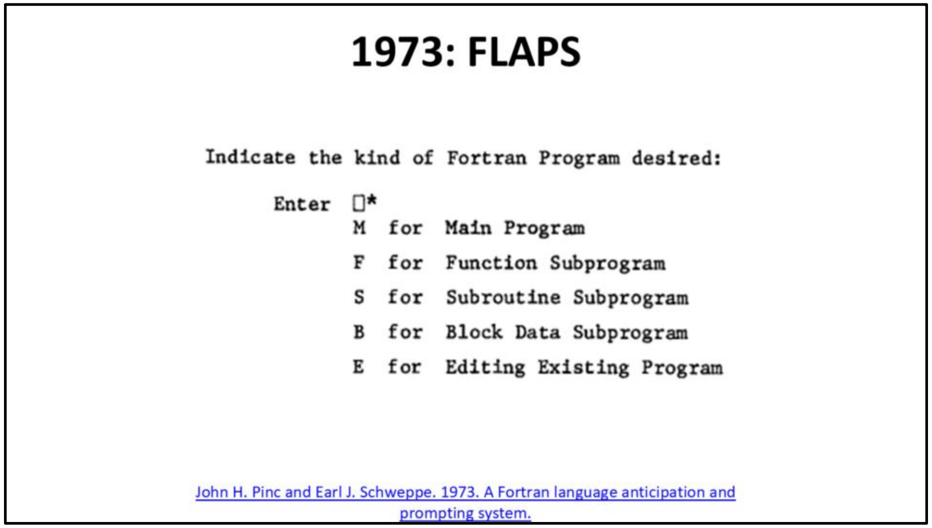
This is the first of several top-down systems that we'll see, where you're prompted to choose a construct through a sort of ad-hoc interface, then fill it out.

Dr. Hansen does say that Emily did successfully prevent syntax errors and was used for teaching PL/I at the University of St. Andrews. He also notes that noöne earnestly used it for programming.



## 1973: Fortran Language Anticipation and Prompting System

This system, the Fortran Language Anticipation and Prompting System, has such a long name that it takes up all the room on my slide.  I can't find any evidence of anyone using the acronym, but I'm going to call it FLAPS to save space.



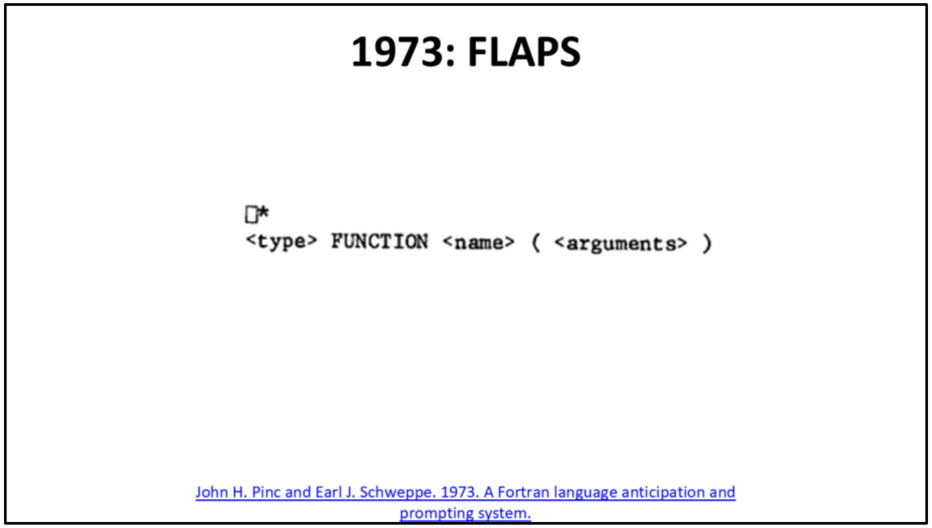
# 1973: FLAPS

```
Indicate the kind of Fortran Program desired:

      Enter  □*
             M  for  Main Program
             F  for  Function Subprogram
             S  for  Subroutine Subprogram
             B  for  Block Data Subprogram
             E  for  Editing Existing Program
```

John H. Pinc and Earl J. Schweppe. 1973. A Fortran language anticipation and prompting system.

The two gentleman at the University of Kansas responsible for the system seem to have a pretty good sense of humor, so I don't think they'd mind.  The paper on FLAPS is also only four pages, which was terribly polite of them.  Fortunately for us, this system is a sort of marriage of Emily and DIALOG, so we can go through it quickly as well.

You start on this screen, where you have to choose what part of the program you'd like to create (or modify).  Let's say that we select "Function."

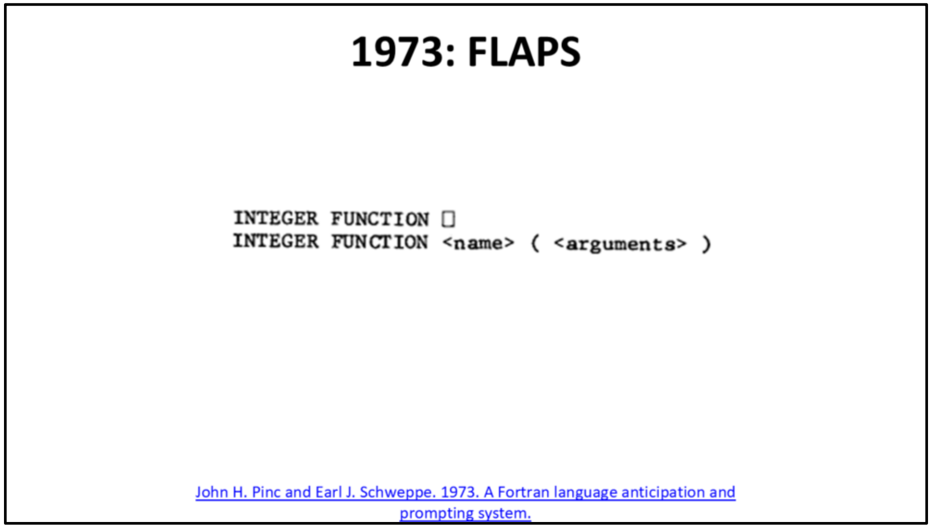


# 1973: FLAPS

```
□*
<type> FUNCTION <name> ( <arguments> )
```

John H. Pinc and Earl J. Schweppe. 1973. A Fortran language anticipation and prompting system.

We’re now presented with this interface. FLAPS knows that the first part of a function is its signature. That empty rectangle is our cursor, and below the line where our cursor is, FLAPS is telling us what we’re supposed to type. Let’s say we type “I”.

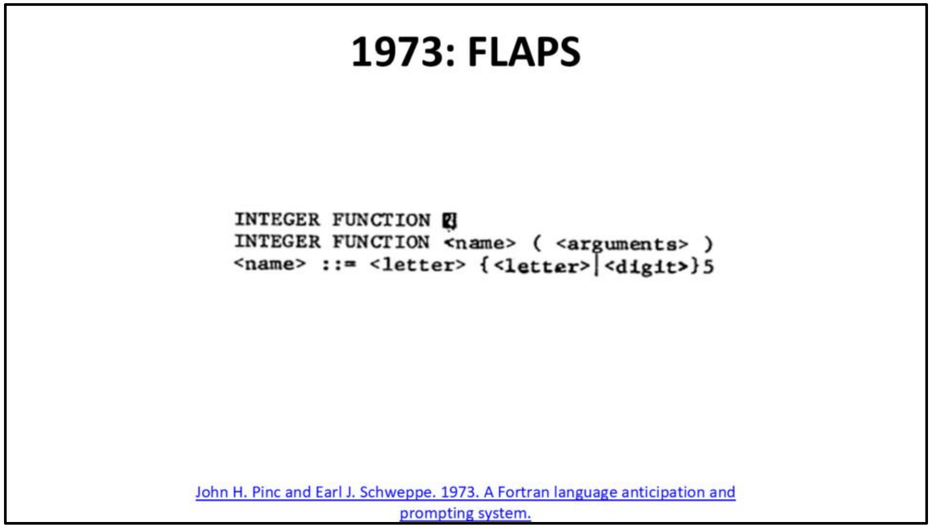


# 1973: FLAPS

```
INTEGER FUNCTION □
INTEGER FUNCTION <name> ( <arguments> )
```

John H. Pinc and Earl J. Schweppe. 1973. A Fortran language anticipation and prompting system.

FLAPS knows that there’s only one Fortran type that starts with “I”, “Integer”, and so just writes the rest of the word for us, as well as the word “FUNCTION” since that must come next.  Next, we’re to type the name.  Here, FLAPS can’t do any fancy autocompletion for us, so it’ll just accept what we type with the notable exception that if we type a character that isn’t permitted, FLAPS won’t append it and will beep angrily instead.



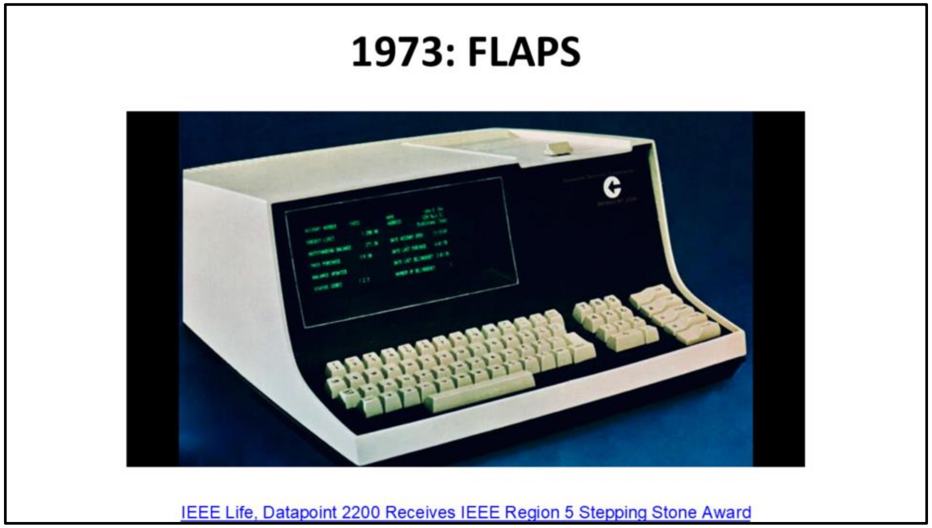
# 1973: FLAPS

```
INTEGER FUNCTION ?
INTEGER FUNCTION <name> ( <arguments> )
<name> ::= <letter> {<letter>|<digit>}5
```

John H. Pinc and Earl J. Schweppe. 1973. A Fortran language anticipation and prompting system.

If we type a question mark, it’ll tell us what a valid name looks like, using this BNF-like syntax.  Notably here, all function names were supposed to be six characters long, which is a limitation of FLAPS, not Fortran.

Anyhow, similar mechanisms are used for the other parts of the programs, and the paper details those.  In the paper’s discussion, they admit that the system was never really fully functional.  It seemed clear to me when I was reading the paper that there were going to be ambiguities, and those simple weren’t addressed.  FLAPS was also slow due to running on a “smart” terminal...

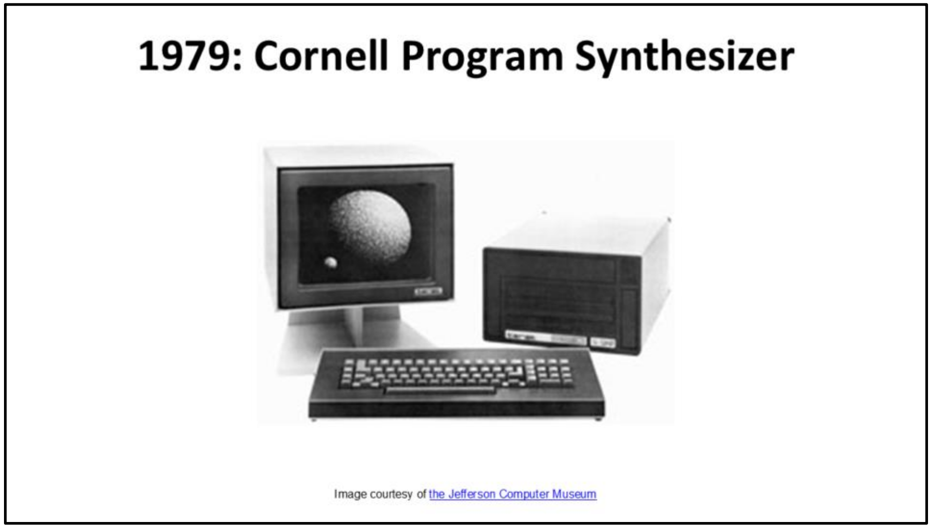


# 1973: FLAPS

IEEE Life, Datapoint 2200 Receives IEEE Region 5 Stepping Stone Award

...the Datapoint 2200.  It is now considered by IEEE to be the first desktop computer, but it wasn’t especially fast; the Fortran code didn’t even run on the terminal, it had to be sent to a mainframe for execution.

But I’ll be honest, I kind of love FLAPS.  I think its overall approach is pretty decent, and it does bear resemblance to a modern-day system that we’ll discuss soon.

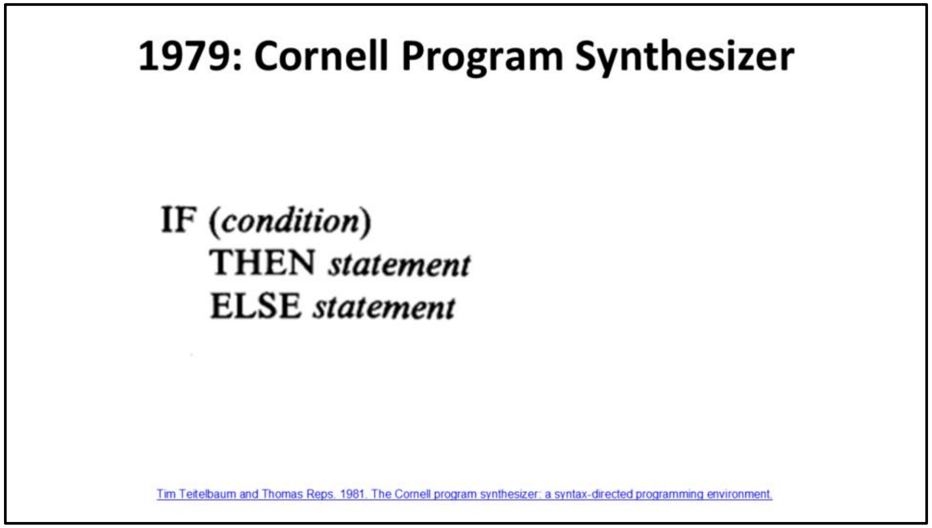


# 1979: Cornell Program Synthesizer

Image courtesy of the Jefferson Computer Museum

Ah, the mighty Cornell Program Synthesizer. Several sources refer to CPS—wrongly, in my opinion—as the first projectional editor. I couldn't find any images of the Cornell Program Synthesizer or any imitations of its use, so that's the Terak 5310/a, the machine on which it ran.

The Cornell Program Synthesizer, originally built by Tim Teitelbaum and Thomas Reps, was unique in a few ways. Firstly, it's input mechanism was based around "templates and phrases." A "template" is a kind of syntactic structure, like an if statement. Teitelbaum wrote "All but the simplest statement types are predefined in the editor as templates. The template is a formatted syntactic skeleton that contains the keywords, matched parentheses and other punctuation marks of the given statement form."

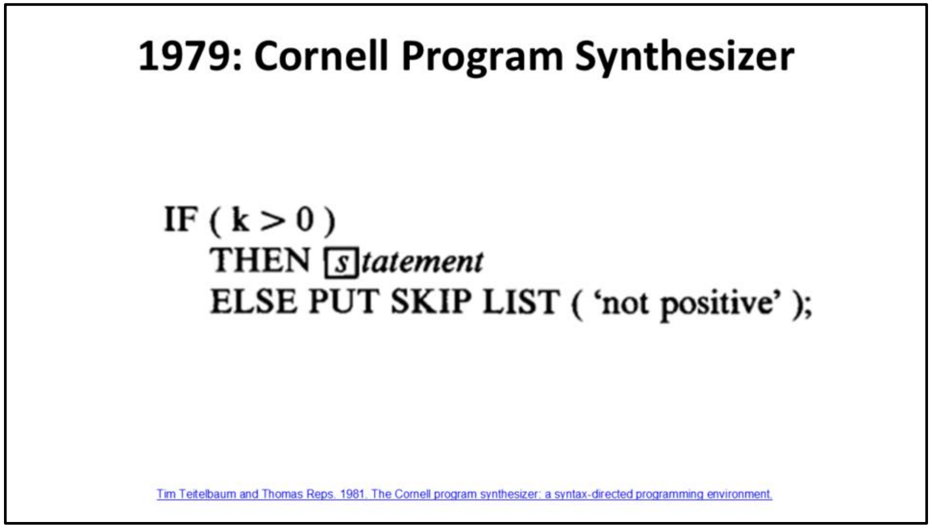


## 1979: Cornell Program Synthesizer

```
IF (condition)
    THEN statement
    ELSE statement
```

Tim Teitelbaum and Thomas Reps. 1981. The Cornell program synthesizer: a syntax-directed programming environment.

This is a template for an “if” statement, and there are three placeholders in it, “condition”, “statement”, and “statement”. You could move the cursor to any of the three placeholders, but you can’t move the cursor within the template, for instance inside the words “THEN” or “ELSE”, because templates were immutable. Once your cursor is in a placeholder, you could choose to insert another template or insert a “phrase.” A phrase was just a sequence of characters; just text. In the 1981 paper, the example given is replacing the *condition* with “k > 0”.



## 1979: Cornell Program Synthesizer

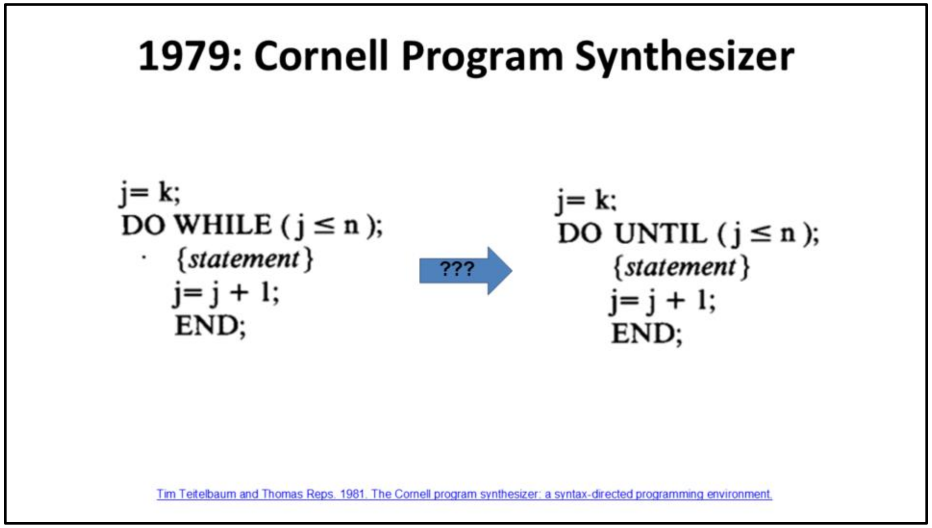
```
IF ( k > 0 )
    THEN statement
    ELSE PUT SKIP LIST ( ‘not positive’ );
```

Tim Teitelbaum and Thomas Reps. 1981. The Cornell program synthesizer: a syntax-directed programming environment.

That was entered as a phrase, just those letters typed precisely in that order. CPS didn’t do any checking of phrases until you tried to leave one. When you attempted to leave, it would check that the program still parsed correctly, and if it didn’t, you would be forced to stay within the phrase and fix it.

This was important, implementation-wise because CPS represented programs internally by their ASTs, not text. The text that was displayed to the user was a “pretty-printing” or “unparsing” of the AST.

As I alluded to earlier, templates themselves were immutable, which led to a perennial problem of tools of this sort.



## 1979: Cornell Program Synthesizer

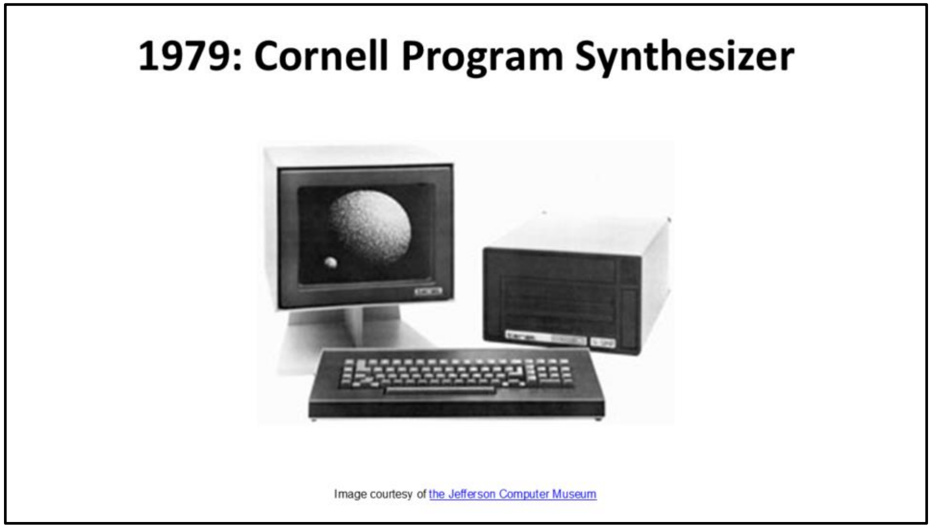
```
j= k;
DO WHILE ( j ≤ n );
      {statement}
      j= j + 1;
      END;
```

```
j= k:
DO UNTIL ( j ≤ n );
      {statement}
      j= j + 1;
      END;
```

Tim Teitelbaum and Thomas Reps. 1981. The Cornell program synthesizer: a syntax-directed programming environment.

If you have a “DO WHILE” template, how to you change it to a “DO UNTIL” template? In a text editor, you can simply delete the word “WHILE” and replace it with the word “UNTIL”. Since “WHILE” is part of a template in CPS, and templates are immutable, that’s illegal. The conventional answer to how to perform this operation is that you’d have to navigate through your WHILE loop, clipping the salient parts of it, then delete the WHILE template, insert an UNTIL template, and insert all the formerly clipped pieces back in. Not a very pleasant experience.

To work around this, CPS added several specific shortcuts for doing this kind of transformation.

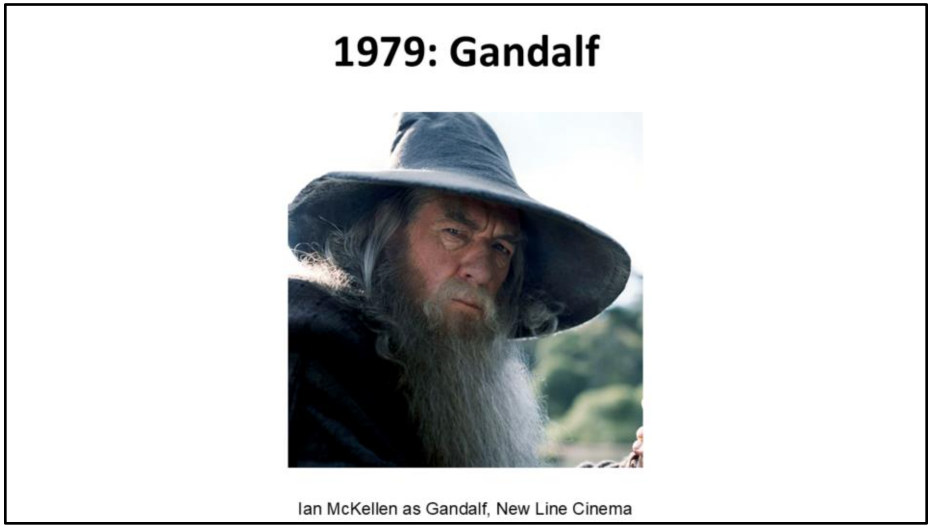


# 1979: Cornell Program Synthesizer

Image courtesy of the Jefferson Computer Museum

One of the big features of CPS was that it wasn’t just for program *construction*, it was also involved in running and debugging programs.  As your program ran, you could watch the cursor jump around to whichever part of the program was currently executing.  The pace at which your program was running could be slowed down such that you could observe problematic sections, and execution could even be stopped and run backwards up to ten steps.  For this reason, CPS is often—and this time, rightly—considered the first IDE.

CPS was used didactically in many universities—something like 1500 students used it per year—and even a few companies got licenses to use it commercially.  It garnered a lot of attention, and there were several other editors that were inspired by it, including ALBE at Yale and Gandalf at Carnegie Mellon.

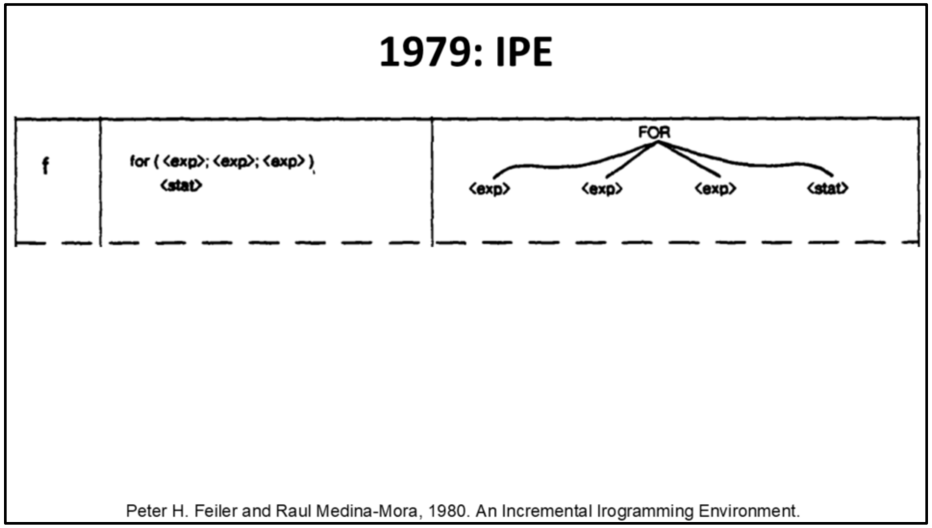


# 1979: Gandalf

Ian McKellen as Gandalf, New Line Cinema

Some staff at Carnegie Mellon had seen a demonstration of CPS, and were impressed, but didn't like it's various limitations at that time, such as that programs could only be 24 lines long. Tim Teitelbaum's doctoral adviser, Nico Habermann, kicked off another program to generate a next-generation development environment named "Gandalf".

There were many aspects to Gandalf, including project management and version control features, but the most salient feature to us is its projectional editor known as an "Incremental Programming Environment," or IPE.

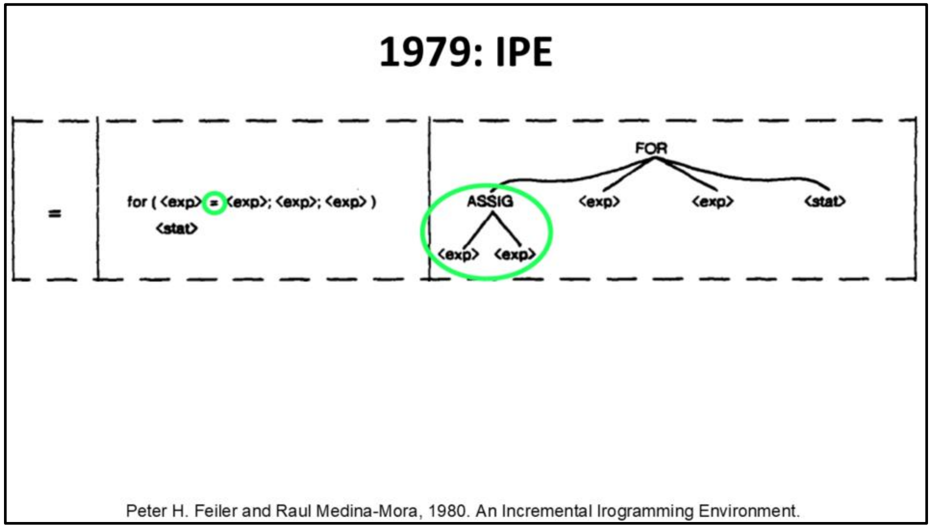


# 1979: IPE

```
for ( <exp>; <exp>; <exp> )
    <stat>
```

Peter H. Feiler and Raul Medina-Mora, 1980. An Incremental Irogramming Environment.

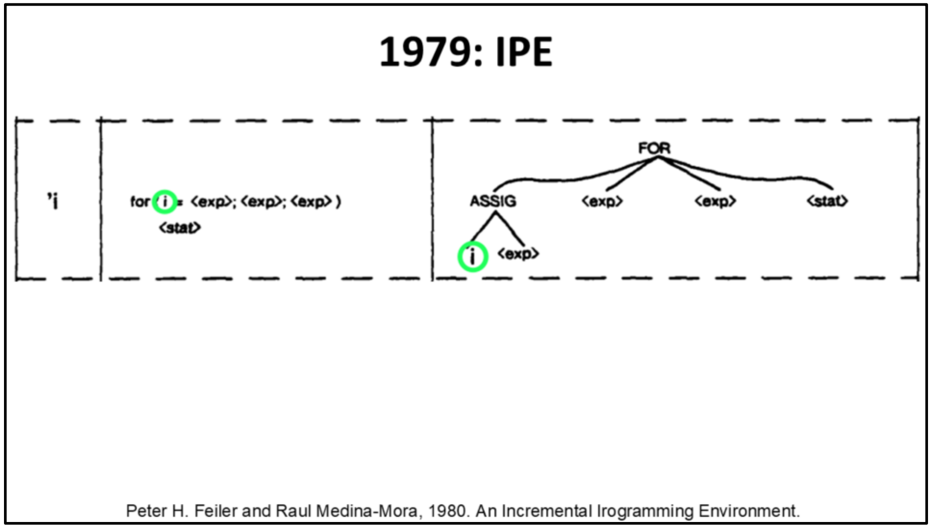
Gandalf used, give-or-take, the same kind of templating system as CPS.  Here is an example of filling in a for-loop template with Gandalf; you can see it contains several placeholders.  Notably, Gandalf targeted a subset of C instead of PL/I.



# 1979: IPE

Peter H. Feiler and Raul Medina-Mora, 1980. An Incremental Irogramming Environment.

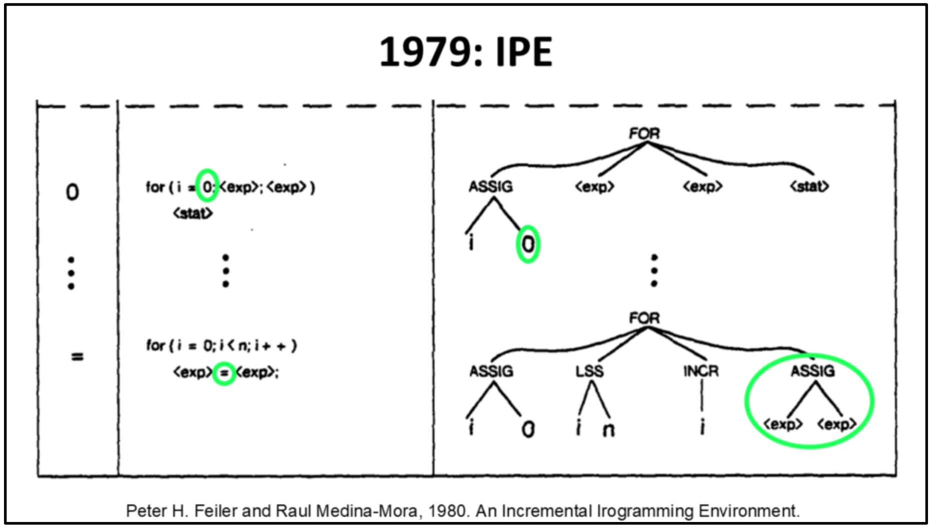
Unlike CPS, Gandalf didn’t have the concept of phrases. Instead, if you wanted to write “i = 0”, you would start by inserting the “=” template, which itself has a placeholder on each side.



# 1979: IPE

Peter H. Feiler and Raul Medina-Mora, 1980. An Incremental Irogramming Environment.

Then filling in the first placeholder there with “i”

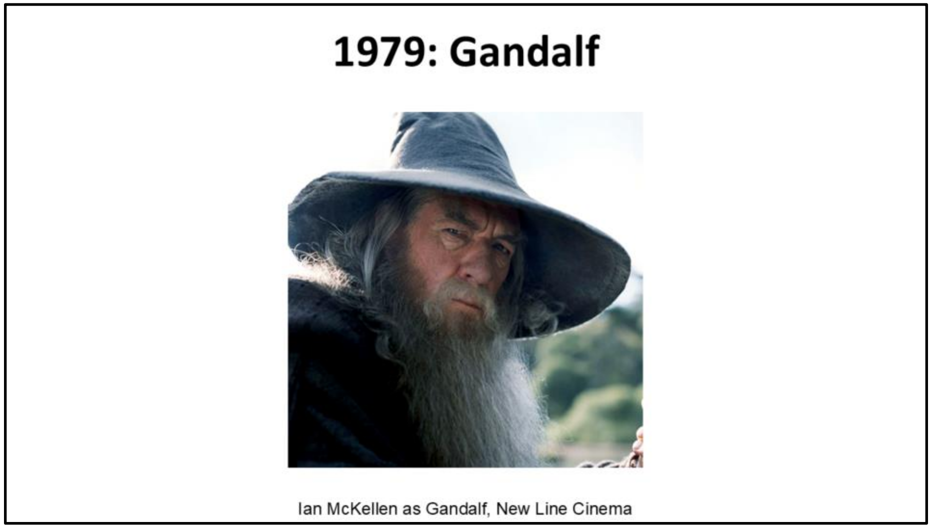


# 1979: IPE

Peter H. Feiler and Raul Medina-Mora, 1980. An Incremental Irogramming Environment.

And the second with “0” and so on.

The literature about the system is divided as to whether users found this to be annoying.  There is general agreement, though, that this style was disliked by experienced programmers.



# 1979: Gandalf

Ian McKellen as Gandalf, New Line Cinema

One of the interesting ideas that came out of Gandalf was the notion of different “unparsings” called “views”. I mentioned before that CPS’ internal representation of the program was an AST, and that was the same for Gandalf also. CPS took a few liberties with its unparsing; namely that it didn’t always show semicolons and some other syntactic noise. The authors of Gandalf realized that this idea could be stretched even further, as in showing the program in a form that didn’t resemble its native syntax at all. Unfortunately, I couldn’t find any good examples of this, but it’ll come back later.

# 1982: GNOME

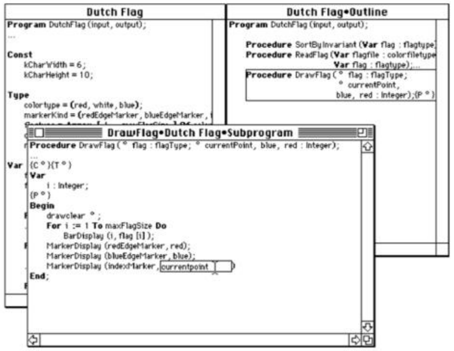
Introduction to the VAX-11 : concepts : 11/780 overview from Computer History Museum on YouTube

There was a successor program to Gandalf at CMU, named the Gandalf NOvice programMing Environment or GNOME. Again, I couldn’t find any images of the system, so this is a stock photo of the VAX-11/780.

The team decided, as many have, that the editor’s best use-case may be in teaching students to program.  There were GNOME environments for a few different languages, the first one being “Karel the Robot”, a PL/I subset used to move a robot about the screen.

Most of the documentation for GNOME just discusses its shortcomings, such as the difficulty of navigating the AST by way of a textual representation, so I’m going to move on to the next system, MacGnome.

# 1989: MacGnome



```
Program DutchFlag (input, output);
...

Const
    kCharWidth = 6;
    kCharHeight = 10;

Type
    colortype = (red, white, blue);
    markerKind = (redEdgeMarker, blueEdgeMarker,
```

```
Program DutchFlag (input, output);

    Procedure SortByInvariant (Var flag : flagtype)
    Procedure ReadFlag (Var flagfile : colorfiletype
                          Var flag : flagtype);...
    Procedure DrawFlag (° flag : flagType;
                          ° currentPoint,
                          blue, red : Integer);{P °}
```

```
Procedure DrawFlag (° flag : flagType; ° currentPoint, blue, red : Integer);
...
{C °}{T °}
Var
    i : Integer;
{P °}
Begin
    drawclear ° ;
    For i := 1 To maxFlagSize Do
        BarDisplay (i, flag [i] );
    MarkerDisplay (redEdgeMarker, red);
    MarkerDisplay (blueEdgeMarker, blue);
    MarkerDisplay (indexMarker, currentpoint )
End;
```

Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann, 1994. Evolution of Novice Programming Environments: the Structure Edtiors of Carnegie Mellon University.

MacGnome began as a port of the GNOME system from the Vax 11-780 to the Macintosh, but it ultimately became so much more.

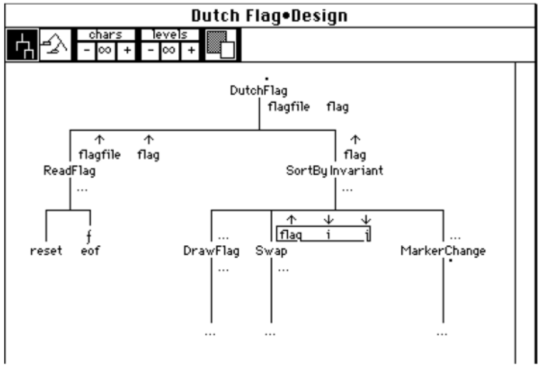
As a bit of an aside, the actual editors were called “genies”, created by MacGnome. Something like that, the verbiage is a little confusing. In any event, one of the big changes that MacGnome introduced was text-editing, in a style somewhat similar to CPS. However, in MacGnome you could write arbitrary text into a tree node and it would be parsed on exit, but any AST node could be selected and edited as text.

MacGnome also solved GNOME’s navigation problems with the mouse. Instead of figuring out how to get from A to B via tree operations, you could simply click on where you wanted to go.

Remember the different unparsings mentioned before? It’s not immediately apparent, but this screenshot is actually showing three views of the same program. In the back-left, there’s the program header with variables, constants, and types. In the back-right, there’s the overview of all the procedures in the program. Clicking on one of the procedure headers would result in the third—center—view, which was the code for only a single procedure.

Now, if you use a nice, modern editor, it probably has capabilities to show or hide things in this way.  All well and good, but not very exciting.  Let’s look at a view that’s a bit more interesting.

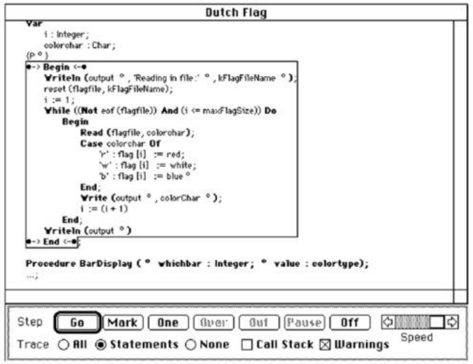
# 1989: MacGnome



Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann, 1994. Evolution of Novice Programming Environments: the Structure Edtiors of Carnegie Mellon University.

This is another unparsing, the design view, a sort of call-graph styled view. Something interesting to note here is the little arrows. The arrows indicate whether a parameter is passed by reference (up arrow) or by value (down arrow). This is a feature I'd love to see more of in modern editors; I went and made something like this for Elixir in my own work.

# 1989: MacGnome



Dutch Flag

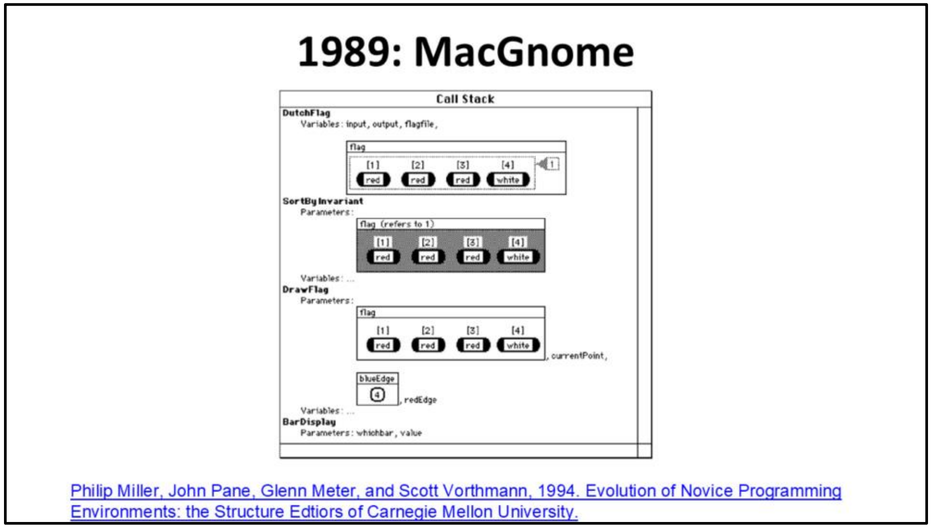
```
Var
    i : Integer;
    colorchar : Char;
{P ° }
•-> Begin <-•
    Writeln (output ° , 'Reading in file:' ° , kFlagFileName ° );
    reset (flagfile, kFlagFileName);
    i := 1;
    While ((Not eof (flagfile)) And (i <= maxFlagSize)) Do
        Begin
            Read (flagfile, colorchar);
            Case colorchar Of
                'r' : flag [i]  := red;
                'w' : flag [i]  := white;
                'b' : flag [i]  := blue °
            End;
            Write (output ° , colorChar ° );
            i := (i + 1)
        End;
    Writeln (output ° )
•-> End <-•;

Procedure BarDisplay ( °  whichbar : Integer; °  value : colortype);
...;
```

Step  Go  Mark  One  Over  Out  Pause  Off

Trace ○ All ● Statements ○ None □ Call Stack ☒ Warnings

Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann, 1994. Evolution of Novice Programming Environments: the Structure Edtiors of Carnegie Mellon University.

As with CPS, the MacGnome environments, or Genies, featured a run-time tracer or debugger with a speed slider.  If you've had the good fortune of using a GUI debugger before, this should feel pretty familiar.



# 1989: MacGnome

```
Call Stack
DutchFlag
    Variables: input, output, flagfile,
        flag
          [1]    [2]    [3]    [4]
          red    red    red    white    1
SortByInvariant
    Parameters:
        flag (refers to 1)
          [1]    [2]    [3]    [4]
          red    red    red    white
    Variables: ...
DrawFlag
    Parameters:
        flag
          [1]    [2]    [3]    [4]
          red    red    red    white   , currentPoint,
        blueEdge
          4   , redEdge
    Variables: ...
BarDisplay
    Parameters: whichbar, value
```

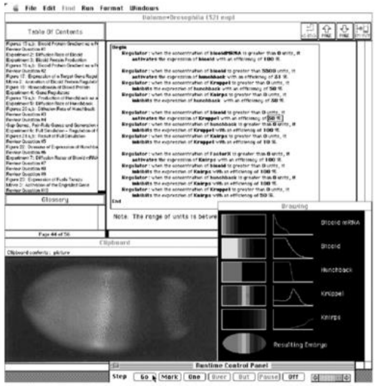
Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann, 1994. Evolution of Novice Programming Environments: the Structure Edtiors of Carnegie Mellon University.

The last thing that I wanted to call out about Genies is their visual debugger view. Here, the debugger is showing us that the DutchFlag procedure called the SortByInvariant procedure and passed a reference to its “flag” variable. The flag variable is an array and is drawn in a special style to indicate this, also showing what all the values are.

Illustrating data structures visually is something that I’m interested in exploring more, and there’s one modern tool that we’ll look at momentarily that is very good at this.

In any event, MacGnome was considered something of a success at CMU. One study showed that students learning to program with MacGnome performed one letter grade higher than their peers who were not. However, MacGnome was painfully slow. Another issue was that the dream had been to be able to generate Genies for languages simply by feeding in an annotated description of the language, but this proved untenable. Ultimately, the Genies that were created were extensively hand-tuned to create friendly environments.

# 1993: ACSE



```
Begin
    Regulator : when the concentration of bicoidRNA is greater than 0 units, it
        activates the expression of bicoid with an efficiency of 100 %.

    Regulator : when the concentration of bicoid is greater than 5500 units, it
        activates the expression of hunchback with an efficiency of [illegible] %.
    Regulator : when the concentration of Kruppel is greater than 0 units, it
        inhibits the expression of hunchback with an efficiency of 50 %.
    Regulator : when the concentration of Kairps is greater than 0 units, it
        inhibits the expression of hunchback with an efficiency of 50 %.

    Regulator : when the concentration of bicoid is greater than 0 units, it
        activates the expression of Kruppel with an efficiency of 50 %.
    Regulator : when the concentration of hunchback is greater than 0 units, it
        inhibits the expression of Kruppel with an efficiency of 100 %.
    Regulator : when the concentration of Kairps is greater than 0 units, it
        inhibits the expression of Kruppel with an efficiency of 10 %.

    Regulator : when the concentration of FactorX is greater than 0 units, it
        activates the expression of Kairps with an efficiency of 100 %.
    Regulator : when the concentration of bicoid is greater than 0 units, it
        inhibits the expression of Kairps with an efficiency of 100 %.
    Regulator : when the concentration of hunchback is greater than 0 units, it
        inhibits the expression of Kairps with an efficiency of 100 %.
    Regulator : when the concentration of Kruppel is greater than 0 units, it
        inhibits the expression of Kairps with an efficiency of 50 %.
End
```

Note: The range of units is betwe

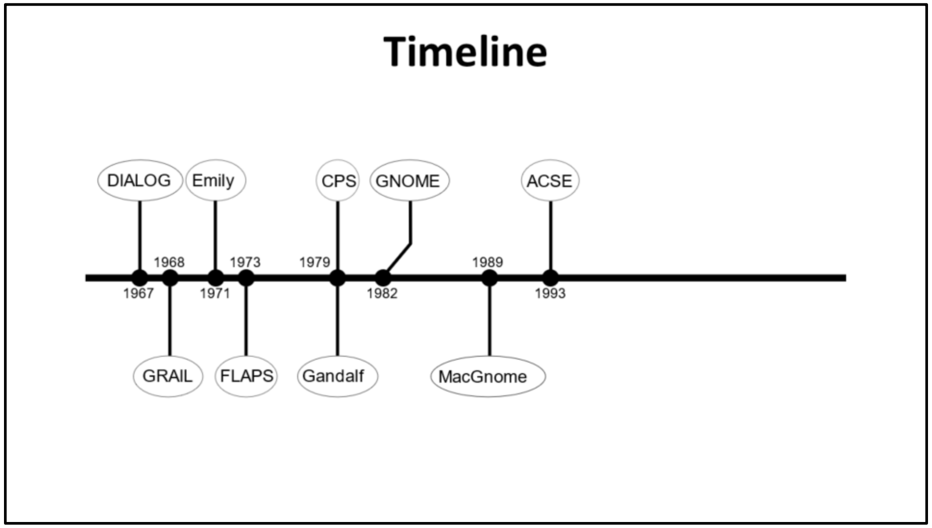
Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann, 1994. Evolution of Novice Programming Environments: the Structure Edtiors of Carnegie Mellon University.

Advanced Computing for Science Education, or ACSE, is the last in the line of Carnegie Mellon structure editors, and the last structure editor we'll discuss.

ACSE was an extension of MacGnome with a few specific goals. Chief among them is that it was designed for a different audience; as the name implies, ACSE was designed for use by students of the sciences, namely biology. ACSE added both a video view, which you can see in the bottom-left and a "simulation view" (bottom-right) where graphics generated by the program could be shown.

The team behind MacGnome believed that they had done such a good job making programming approachable that they believed the environment would be easily usable by students who were not studying programming. I'm not sure how much that was actually true. ACSE was also used by students studying computer science, and object orientation support was another addition feature of the environment. I think the premier environment went from Pascal to Object Pascal.

After ACSE, I couldn't find much reference to work on projectional editors from this point—the mid 90s—until the mid 2010s. I'm not sure why that is exactly; I'd be interested to learn if anyone knows.



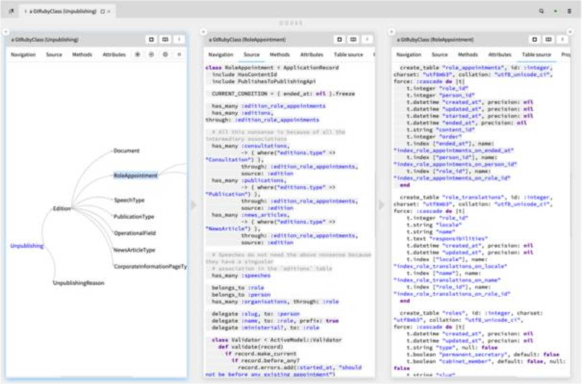
## Timeline

Extending the timeline we saw before to the present, we see that there's been something of a dearth of projectional editors over the last 30 years. However, there are a few more recent projects that are trying new approaches.

# The Present

Now that we know where we came from, let’s take a look at what we have today.

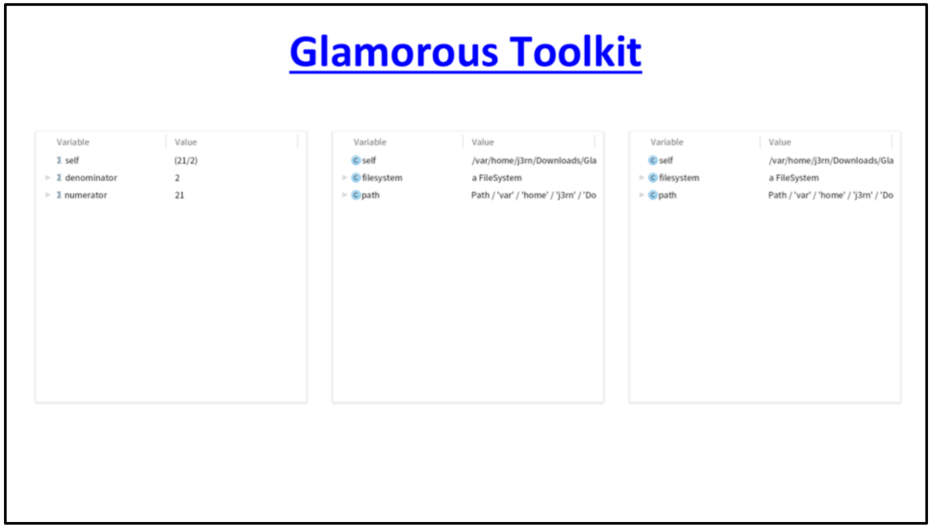
# Glamorous Toolkit



```
class RoleAppointment < ApplicationRecord
  include HasContentId
  include PublishesToPublishingApi

  CURRENT_CONDITION = { ended_at: nil }.freeze

  has_many :edition_role_appointments
  has_many :editions,
through: :edition_role_appointments

  # All this nonsense is because of all the
intermediary associations
  has_many :consultations,
           -> { where("editions.type" =>
"Consultation") },
           through: :edition_role_appointments,
           source: :edition
  has_many :publications,
           -> { where("editions.type" =>
"Publication") },
           through: :edition_role_appointments,
           source: :edition
  has_many :news_articles,
           -> { where("editions.type" =>
"NewsArticle") },
           through: :edition_role_appointments,
           source: :edition

  # Speeches do not need the above nonsense because
they have a singular
  # association in the `editions` table
  has_many :speeches

  belongs_to :role
  belongs_to :person
  has_many :organisations, through: :role

  delegate :slug, to: :person
  delegate :name, to: :role, prefix: true
  delegate :ministerial?, to: :role

  class Validator < ActiveModel::Validator
    def validate(record)
      if record.make_current
        if record.before_any?
          record.errors.add(:started_at, "should
not be before any existing appointment")
```

```
  create_table "role_appointments", id: :integer,
charset: "utf8mb3", collation: "utf8_unicode_ci",
force: :cascade do |t|
    t.integer "role_id"
    t.integer "person_id"
    t.datetime "created_at", precision: nil
    t.datetime "updated_at", precision: nil
    t.datetime "started_at", precision: nil
    t.datetime "ended_at", precision: nil
    t.string "content_id"
    t.integer "order"
    t.index ["ended_at"], name:
"index_role_appointments_on_ended_at"
    t.index ["person_id"], name:
"index_role_appointments_on_person_id"
    t.index ["role_id"], name:
"index_role_appointments_on_role_id"
  end

  create_table "role_translations", id: :integer,
charset: "utf8mb3", collation: "utf8_unicode_ci",
force: :cascade do |t|
    t.integer "role_id"
    t.string "locale"
    t.string "name"
    t.text "responsibilities"
    t.datetime "created_at", precision: nil
    t.datetime "updated_at", precision: nil
    t.index ["locale"], name:
"index_role_translations_on_locale"
    t.index ["name"], name:
"index_role_translations_on_name"
    t.index ["role_id"], name:
"index_role_translations_on_role_id"
  end

  create_table "roles", id: :integer, charset:
"utf8mb3", collation: "utf8_unicode_ci",
force: :cascade do |t|
    t.datetime "created_at", precision: nil
    t.datetime "updated_at", precision: nil
    t.string "type", null: false
    t.boolean "permanent_secretary", default: false
    t.boolean "cabinet_member", default: false, null:
false
    t.string "slug"
```

The first system I'd like to show you is the Glamorous Toolkit. Glamorous Toolkit feels to me like a native GUI environment for writing code, in the same what that something like Autocad or Solidworks is a native GUI environment for doing CAD. It has a very interesting idea at its core that I'd like to show you.



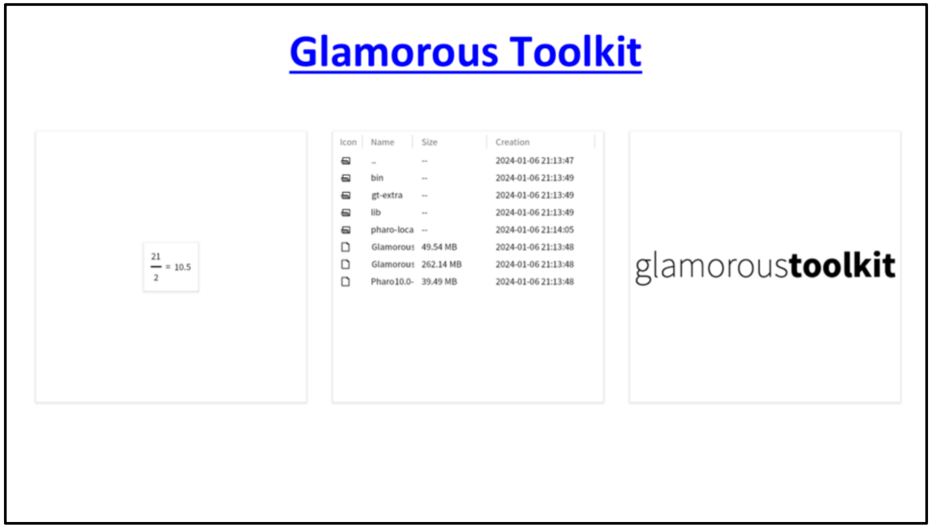
## Glamorous Toolkit

| Variable | Value |
|---|---|
| Σ self | (21/2) |
| ▸ Σ denominator | 2 |
| ▸ Σ numerator | 21 |

| Variable | Value |
|---|---|
| Ⓒ self | /var/home/j3rn/Downloads/Gla |
| ▸ Ⓒ filesystem | a FileSystem |
| ▸ Ⓒ path | Path / 'var' / 'home' / 'j3rn' / 'Do |

| Variable | Value |
|---|---|
| Ⓒ self | /var/home/j3rn/Downloads/Gla |
| ▸ Ⓒ filesystem | a FileSystem |
| ▸ Ⓒ path | Path / 'var' / 'home' / 'j3rn' / 'Do |

Some nice editors have a handy object inspector, like the one shown here. These objects are specifically Pharo Smalltalk objects, but they should be somewhat familiar; they have an identity and a set of attributes, methods, etc. If you `console.log` a variable in JavaScript, Firefox and Chrome will give you a pretty similar inspector. However, this view is homogenous; it looks the same regardless of what object is being inspected. Glamorous Toolkit asks the question, “what if different objects could have unique inspect views?”



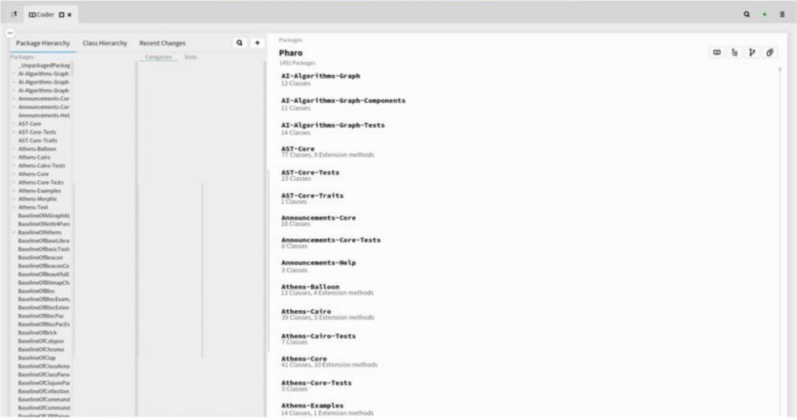
# Glamorous Toolkit

$\frac{21}{2} = 10.5$

| Icon | Name | Size | Creation |
|---|---|---|---|
| 🗀 | .. | -- | 2024-01-06 21:13:47 |
| 🗀 | bin | -- | 2024-01-06 21:13:49 |
| 🗀 | gt-extra | -- | 2024-01-06 21:13:49 |
| 🗀 | lib | -- | 2024-01-06 21:13:49 |
| 🗀 | pharo-loca | -- | 2024-01-06 21:14:05 |
| 🗋 | Glamorous | 49.54 MB | 2024-01-06 21:13:48 |
| 🗋 | Glamorous | 262.14 MB | 2024-01-06 21:13:48 |
| 🗋 | Pharo10.0- | 39.49 MB | 2024-01-06 21:13:48 |

These are different, custom views for the same objects that were being inspected on the previous slide. The rational number looks like a fraction and shows a decimal approximation, the path object looks like a file browser, etc. The hope is that by showing objects in formats that are specific to them, it will be easier to ascertain the knowledge you were seeking by inspecting them in the first place.

## Glamorous Toolkit



Packages

Pharo

1451 Packages

AI-Algorithms-Graph
12 Classes

AI-Algorithms-Graph-Components
11 Classes

AI-Algorithms-Graph-Tests
14 Classes

AST-Core
77 Classes, 8 Extension methods

AST-Core-Tests
23 Classes

AST-Core-Traits
1 Classes

Announcements-Core
10 Classes

Announcements-Core-Tests
8 Classes

Announcements-Help
3 Classes

Athens-Balloon
13 Classes, 4 Extension methods

Athens-Cairo
39 Classes, 5 Extension methods

Athens-Cairo-Tests
7 Classes

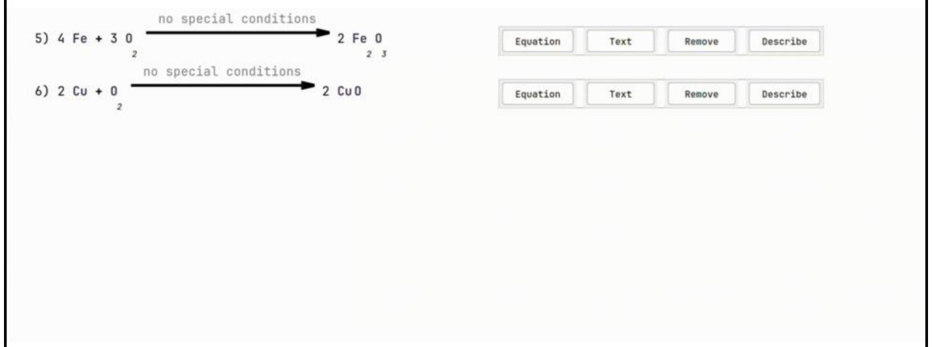
Athens-Core
41 Classes, 10 Extension methods

Athens-Core-Tests
3 Classes

Athens-Examples
14 Classes, 1 Extension methods

There are many more aspects to Glamorous Toolkit to explore, but what is perhaps most relevant to this talk is an interface called “Coder”; the interface you use to actually write code in the system. It opens with an interface that looks a bit like a file browser, showing you all the packages available in your project. From here you can click a package to see its classes, click a class to see its methods, and so on. In the method view, you can click an individual method to see its source and modify it. While the code editor doesn’t prevent you from making syntax errors, like some of the other systems we viewed, it *does* prevent you from, say, writing a function without a body or putting an expression where it doesn’t belong.

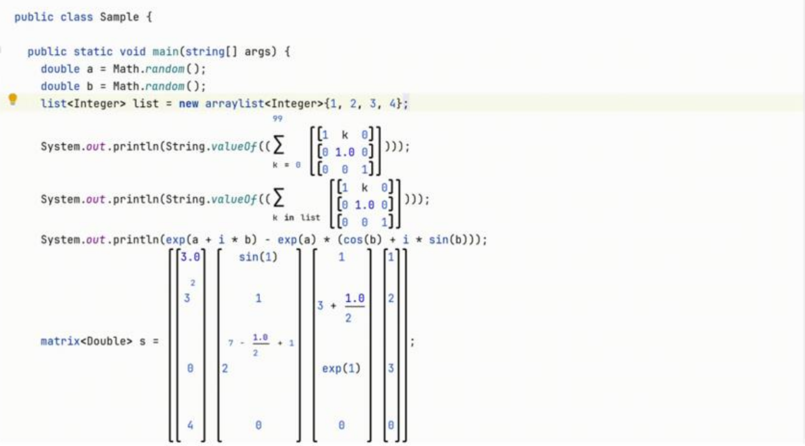
# JetBrains MPS



5) $4\ Fe + 3\ O_2 \xrightarrow{\text{no special conditions}} 2\ Fe_2O_3$

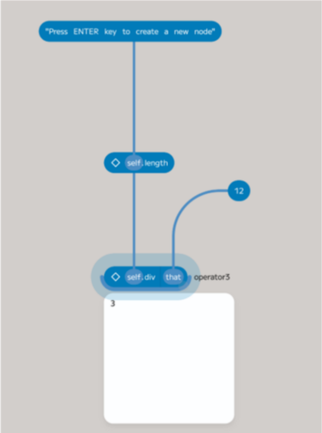
6) $2\ Cu + O_2 \xrightarrow{\text{no special conditions}} 2\ CuO$

# JetBrains MPS

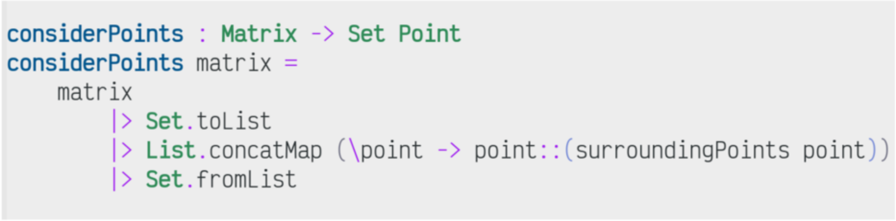


```
public class Sample {

  public static void main(string[] args) {
    double a = Math.random();
    double b = Math.random();
    list<Integer> list = new arraylist<Integer>{1, 2, 3, 4};
                                            99
                                                [[1   k  0]]
    System.out.println(String.valueOf((∑        [[0 1.0  0]]));
                                         k = 0  [[0   0  1]]
                                                      [[1   k  0]]
    System.out.println(String.valueOf((∑              [[0 1.0  0]]));
                                         k in list    [[0   0  1]]
    System.out.println(exp(a + i * b) - exp(a) * (cos(b) + i * sin(b)));
                        [[3.0] [     sin(1)        ] [     1     ] [1]]
                        [[   ] [                   ] [           ] [ ]]
                        [[ 2 ] [                   ] [           ] [ ]]
                        [[3  ] [       1           ] [3 + 1.0    ] [2]]
                        [[   ] [                   ] [      ---  ] [ ]]
                        [[   ] [                   ] [       2   ] [ ]]
    matrix<Double> s =  [[   ] [  7 - 1.0 + 1      ] [           ] [ ]];
                        [[   ] [      ---          ] [           ] [ ]]
                        [[   ] [       2           ] [           ] [ ]]
                        [[ 0 ] [2                  ] [  exp(1)   ] [3]]
                        [[   ] [                   ] [           ] [ ]]
                        [[ 4 ] [       0           ] [     0     ] [0]]
```

# Enso

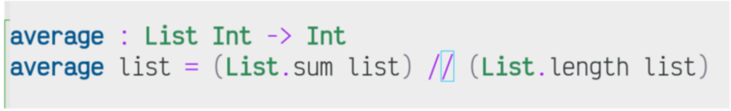

"Press ENTER key to create a new node"
self.length
12
self.div that
operator3
3

# Pipes



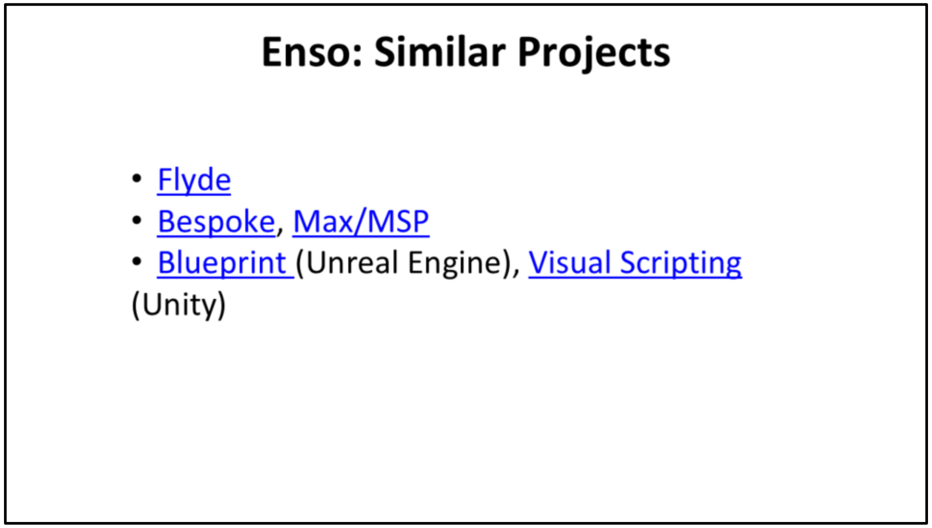
```
considerPoints : Matrix -> Set Point
considerPoints matrix =
    matrix
        |> Set.toList
        |> List.concatMap (\point -> point::(surroundingPoints point))
        |> Set.fromList
```

# Pipes Break



```
average : List Int -> Int
average list = (List.sum list) // (List.length list)
```

# Enso



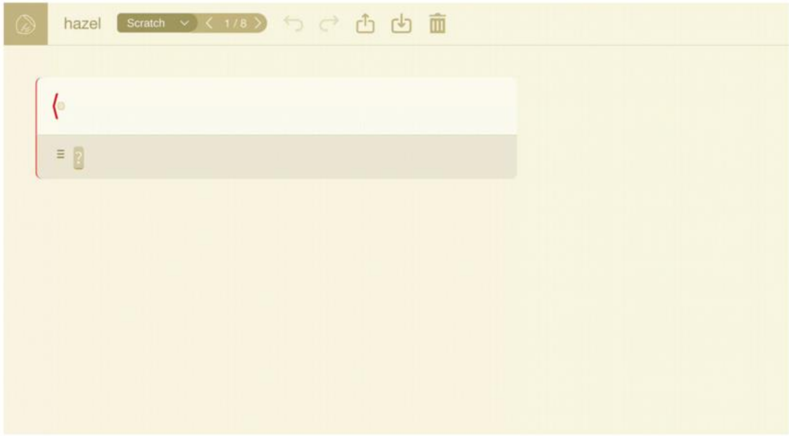
## Enso: Similar Projects

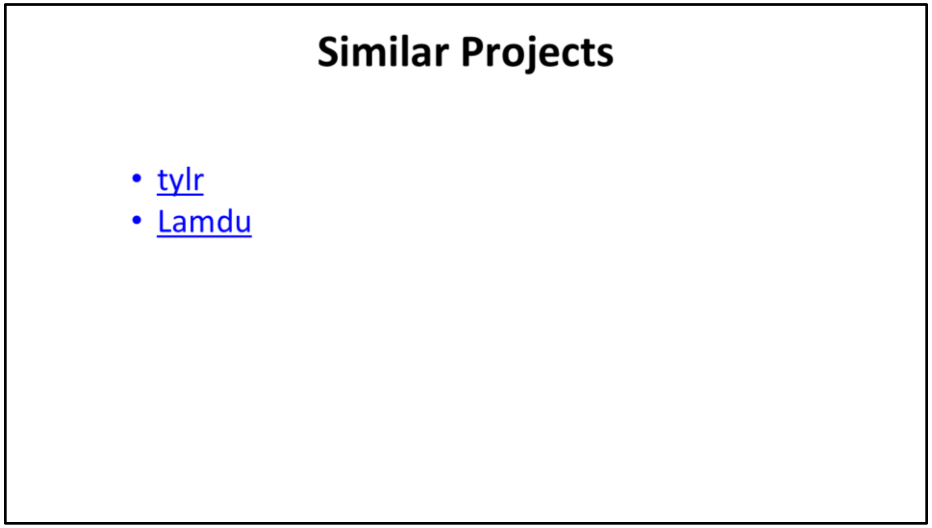
- Flyde
- Bespoke, Max/MSP
- Blueprint (Unreal Engine), Visual Scripting (Unity)

Almost all projectional editors act like guardrails, preventing you from making certain classes of mistakes; syntax errors, namely, but sometimes others as well, like type errors. This is especially useful for beginners, and we'll talk about that more later.

And this, frankly, was all I thought projectional editors were striving to achieve when I first learned about them, and is the only goal of many earlier projectional editors. However, there are two further goals of projectional editors.

# Hazel



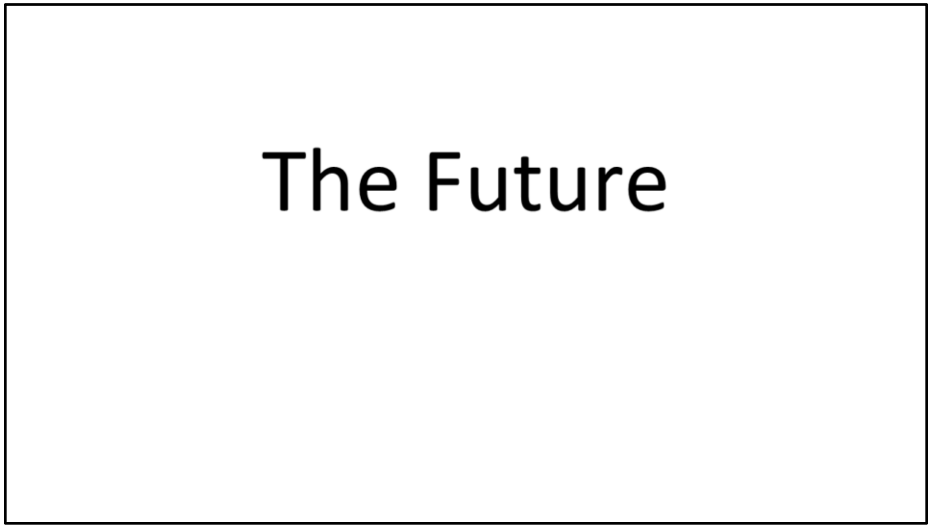


## Similar Projects

- tylr
- Lamdu

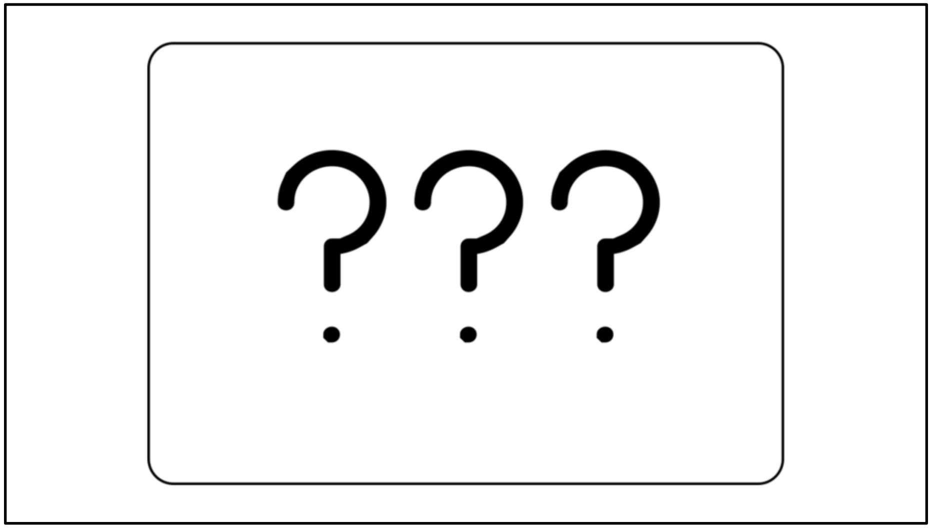
Almost all projectional editors act like guardrails, preventing you from making certain classes of mistakes; syntax errors, namely, but sometimes others as well, like type errors.  This is especially useful for beginners, and we'll talk about that more later.

And this, frankly, was all I thought projectional editors were striving to achieve when I first learned about them, and is the only goal of many earlier projectional editors.  However, there are two further goals of projectional editors.

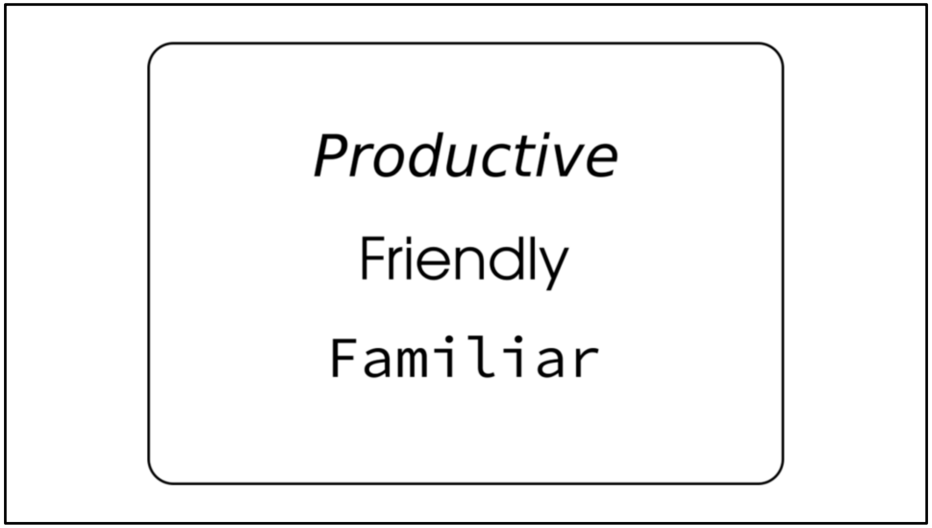


# The Future

And this brings us to: The Future.  An aspirational future, that is.



What does our perfect projectional editor of tomorrow look like?



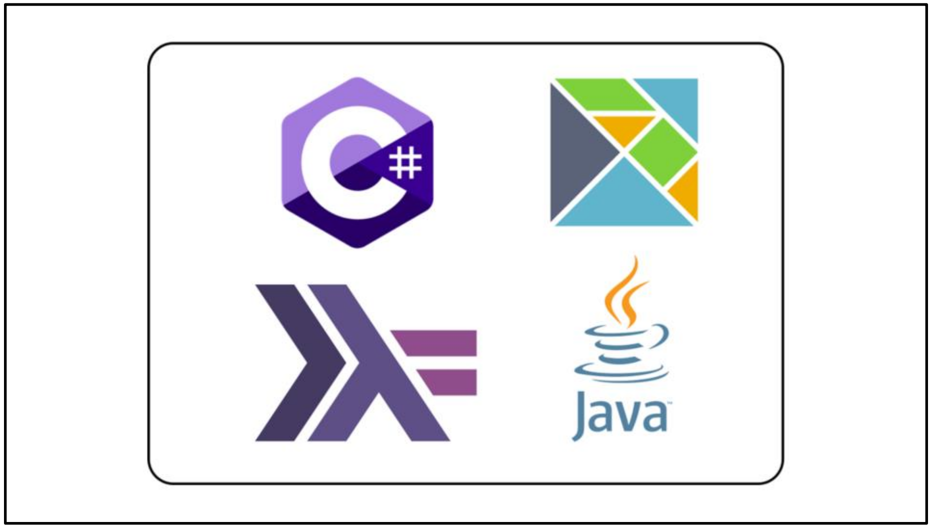
*Productive*

Friendly

`Familiar`

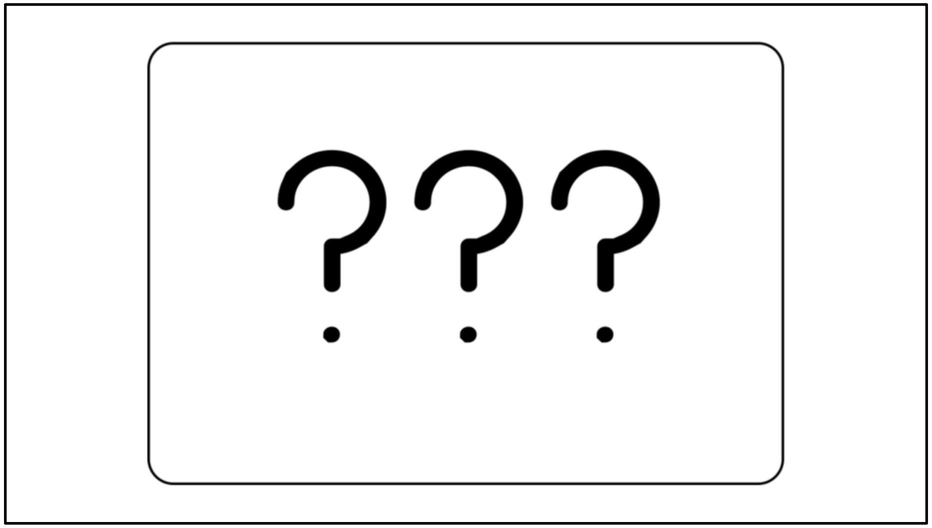
I believe that this editor of the future should have the following three traits: productive, friendly, and familiar.

It must be productive because if it is slow to use (like Emily), it will be eschewed by programmers with deadlines (essentially everyone).
It must be friendly because it is asking developers to change how they work. It should welcome them in and help them on their journey.
It must be familiar because people resist change. Edwin Brady, the creator of Idris, has spoken about the “weirdness budget”, arguing that a tool can only be so weird, or people will be immediately repulsed by it.



I also think it should fully support at least one general-purpose programming language.  Obviously, something industrial like Java or C# would be ideal, but I'd be happy even with something weird like Elm or Haskell.

Well, I think it should have the features of the editors we've talked about; namely Glamorous Toolkit's visualization of datastructures, MPS' ability to write code through GUIs, and Hazel's ability to prevent syntax errors.  I also tossed the IntelliJ logo in there to represent typical IDE features like a debugger.



Beyond these generalities, I *don’t* know what the perfect editor of the future looks like.  But if we care about the ergonomics of programming—and we should!—it behooves us to try.

Jonathan Arnett

j3rn.com
@j3rn@fosstodon.org
github.com/J3RN