Back in the day, the semicolon issue was a big deal. Compiling was a lot slower so putting a semicolon in the wrong place was a painful experience shared by everyone except Lisp programmers. Cliques and parties formed around concepts of proper semicolon use. Some enterprising rabblerouser did a study that showed that the Pascal syntax that treated semicolons as statement separators led to more syntax errors than the C syntax which treated semicolons as statement terminators. This inflamed the great controversy.

In those days, it was common for someone to introduce a new programming language with a brief description of the major features and those brief descriptions commonly included the language's position on the semicolon issue. Python became famous as the language that "solved" the semicolon issue by making indentation significant. It was an interesting language with lots of interesting features, but was mostly known for the indentation. The Icon programming language had some marvelous innovations. Descriptions of it often went something like this:

> *Icon is a procedural programming language where any expression can potentially generate a stream of results rather than just one. Boolean operations and control structures are defined to make use of multiple results in ways that make them more powerful than in single-result languages. There are powerful built in data structures and a string scanning construct that uses generators to create a new way of matching and parsing strings. Oh, and semicolons are statement separators except that they are optional at the end of a line when they are not needed for disambiguation.*

One of these things is not like the others. Would Icon have been a significantly different language if there had been a different semicolon rule?

It gets worse. It isn't enough that language designers get pedantic about semicolons, companies and programming groups get pedantic about how long your lines get, where the left brace goes, and how many spaces you indent.

Why don't mathematicians get hung up on syntax like this? Where are the competing mathematics languages with schools of thought about whether the factorial symbol ought to be a suffix operator or a prefix operator? Mathematicians make up notation as they go and there are usually not too many complaints (there are some notable exceptions like *Principia Mathematica*).

Notation doesn't matter so much to mathematicians because mathematics is only intended to be read by people. Programming languages must be readable by machines as well as by people and so there is a need to compromise between the strict rules that are needed by computers vs. the flexibility that is valued by people.

But that compromise is only needed because the computer needs to read the same text file that the people read. What if it didn't? What if the computer could read a file of abstract syntax trees

and people could still read programming language text?

An abstract syntax tree or AST is how the compiler sees your program after parsing and before generating code. It is basically the logical structure of your program. More specifically, the compiler sees your program as a symbol table that matches symbols like function names to abstract syntax trees that implement the function. And this is how the computer should always represent the logical structure of a program --not just in the compiler. The programming language text should only be an form of presentation to the user and never input to the computer.

Imagine an editor that guides you in writing a program. You don't have to know the exact computer-precise syntax for a function declaration. You just press a button that says "create function" and the computer prompts you for the function name. Then it prompts you for the return type, then for the parameters and their types. When you have entered the information the computer shows you the function declaration and puts your editing point in the body of the function. When you type "if" in the function body, the computer lays out an **if** statement with condition, **then** branch, and **else** branch empty for you to fill in.

There are programming editors that superficially work like this, but what they do is generate programming language text for the compiler to read. What I am proposing is that the computer builds the abstract syntax tree as the user types and generates code directly from that AST. The computer never stores programming language text on disk, only presents it to the user in an editing window as a view of the AST. And when the editor presents the text, it automatically formats it according to some specification so the user never has to worry about how many spaces to indent or how long the lines are.

Modern integrated development environments have a cataloging function or a separate program that reads through the source files parsing them just enough to get declarations so that they can construct a database of named entities (classes, functions, variables, etc.) and then the code browser uses that database to look up declarations in the source code. The database can get out of date, and it can slow down your computer when the cataloging function is running. This is not needed if the computer stores the program as an AST. The internal representation of the program already is a database of declarations that the code browser can access directly.

Other utilities like the debugger, the diff program, and the revision control system also have to work with ASTs rather than text so they can work with logical structure rather than lines and they don't need a parser to do so.

How the computer presents the progam to the user is no longer bound to a particular language, only to the logical structure of the program. The user can ask for a C-like representation:

```
int f(int n) {
  if (n < 2) {
    return 1;
```

```
  } else {
    return n*f(n-1);
}
```

or Pascal:

```
function f(n: integer):integer is
begin
  if n < 2 then
    factorial := 1
  else
    factorial := n*f(n-1)
end
```

or Lisp:

```
(function (f n)
  (cond (< n 2) 1 (* n (f (- n 1)))))
```

or pick a more mathematical representation like

$f(n) \equiv 1, n < 2$

$f(n) \equiv n \times f(n-1)$ otherwise

The possibilities are endless. There is no need to stick with ASCII or even text. Any form of presentation is possible, perhaps a graphical representation based on flow charts or E/R diagrams.

Now let's go back to C notation and ask the editor to change the function name to "factorial":

```
int factorial(int n) {
  if (n < 2) {
    return 1;
  } else {
    return n*factorial(n-1);
}
```

There is no need for the programmer to go through the program finding every instance of "f", figuring out if it is a call to the function and changing it; the computer represents each call to f() as a pointer to symbol-table entry, not as a name. So when you change the name of f() in the symbol table the representation on the screen changes everywhere.

Up to now the computer has been hiding certain grungy parts of the code so that we can study the main logic undisturbed by the details of error handling. But when we want to see the grungy details, the computer shows us:

```
int f(int n) {
  assert(n>=0);
  if (n < 2) {
    return 1;
  } else {
    try {return n*f(n-1);}
    catch (NumericOverflowError e) {...}
```

```
        catch (StackOverflowError e) {...}
    }
```
Notice that we aren't changing the code, here, just revealing or hiding parts.

Even more complex view changes are possible. For example suppose you have a collection of object that all have a print method in common. Sometimes you want to see the print method as part of the object so that you can compare what it does with the members of the object:

```
class Foo::InterfaceX {
  int x;
  ...
  void print(ostream::out) {out << x;}
  ...
```

Other times you want to see all of the print methods together so you can compare what the different print methods in different classes are doing, perhaps with a syntax like this:

```
void InterfaceX::print(ostream::out) {
  typecase(this) {
  case Foo: {out << x;}
  case Bar: {out << y << "." << z;}
  case Zed:
  ...
```

Again, we aren't changing the code, just viewing it in a different way.
When the program is represented as text the editor has to be able to do a complete parse of the program in order to come up with these sorts of sophisticated global view changes. But when the program is in the form of a name table of ASTs, then it is just a matter of some tree manipulation.

With this model of programming, a whole class of errors become impossible. You can't put a semicolon or brace in the wrong place because you don't type them at all (except possibly as a keyboard shortcut to indicate the end of a statement or something). Parentheses, braces, semicolons, function headers, class headers, type declarations --these are all just computer generated representations of your program structure. With programs-as-text, you have to tell this structure to the computer. With programs-as-ASTs, the computer tells you the structure. Of course you have to define the structure in some way. Perhaps with a dialog or perhaps with actual text --but in this case, the computer will not let you enter wrong text --or at least, will not accept dumb typing errors.

For example, misspelling variable or function names is a thing of the past because the computer can't parse the name until you spell it right. And since the computer knows all names that are in scope at any point, it can give you an abbreviated list of choices to complete the name. When you want to use a function from another module, you type the name; the computer gives you a list of modules that define the name, and you pick one. From then on, that name means what you told the computer it means and you don't have to worry about changing what it means by careless use of declarations, imports or includes. The computer writes the declarations, imports

and includes for you and always does it properly.

Of course there are complexities to deal with. If you look carefully at the different language examples then you will see cases where there are slightly different abstract syntax trees represented in the different languages. To handle these language differences the presenter of the AST has to be sophisticated enough to do certain local transformation based on the requested source language.

It would be impossible (or at least, very difficult) to implement full language-to-language translations. It would be better to invent extensions to various languages to support certain features. For example, basic Pascal does not have classes so if the language being presented has classes and you want to present it in Pascal form then you have to use a form of Pascal that has a class extension.

This is the way things ought to be in the Twenty-First Century. Computers should not use text as the internal representation for complex structures such as programs. Data should be represented as object and there should be generic tools to deal with such objects. Having a logical internal representation to deal with makes so many things so much easier to do, that it could lead to an explosion of new development methods if we had this sort of representation for programs.