Prometheus

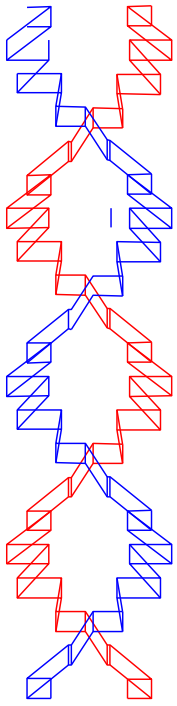
Notes on Converting the UCSD Pascal Compiler to C++

While Creating a New type of Compiler for AI – Neural

Applications by giving them Digital DNA

Gerold Lee Gorman

Sources available on GitHub: GNU/MIT.



DESCRIPTION:

What if we could add a kind of "digital DNA", to a classic language like UCSD Pascal, with an eye toward developing an efficient platform that will facilitate the creation of projects that integrate feature sets from several other programming languages and styles within a single unified framework that integrates the functionality of multiple devices within a cohesive environment. In effect, such a framework might allow an aspiring robot designer to write an application in a high-level language such as PASCAL, and then cross-compile that application to another intermediate language such as a specialized variant of LISP, which could then either be implemented in the form of a C++ library which provides some of the features of LISP or else another meta-compiler might be used to convert the intermediate representation to run on a microcontroller such as a Propeller 2 using the built-in FORTH interpreter or else into the native assembly, or even traditional UCSD p-code.

DETAILS:

Several approaches are frequently taken when developing projects that involve some type of AI. In the traditional approach, interaction with a simulated intelligence can be produced by combining simple pattern matching techniques with some type of scripting language which in turn provides a seemingly life-like experience, which within some contexts can be highly effective, even if only up to a certain point. "This is the approach taken by classic chatbots such as ELIZA, PARRY, MEGAHAL, SHURDLU, and so on." Whether this type of AI is truly intelligent might be some subject for debate, and arguments can be made both in favor of, as well as against claims that such systems are in some way intelligent, on the one hand - even though nobody can reasonably make any sort of claim that such systems might in any way be sentient - yet WHEN they work, they tend to work extremely well.

Most modern attempts at developing AI as of late seem to be focused on efforts to develop applications that more accurately model some of the types of behaviors associated with the types of neural networks found in actual biological systems. Such systems tend to be computationally intensive, often requiring massively parallel computing architectures which are capable of executing billions of concurrent, as well as pipelined non-linear matrix transformations so as to perform even the simplest simulated neuronal operations. Yet this approach gives rise to so-called learning models that might not only have the potential to recognize puppies, etc. but why not build networks that can try to solve more esoteric problems like certain issues in bio-molecular research, and mathematical theorem proving, etc.

Thus, the first approach seems to work best for problems that we know how to solve, and this method, therefore, leads to solutions, that - when they work - are both highly efficient, as well as provable, with the main issues being the amount of work that goes into content creation, as well as debugging and testing.

The second approach seems to offer the prospect of allowing for the creation of systems that are arguably crash-proof, at least in the sense that it should be possible to build simulations of large neural networks, that are just massively parallelized as well as pipelined matrix algebraic data flow engines, which from a certain point of view, is simplicity in and of itself. So that would of course seem to imply that from at least one point of view, the hardware can be made crash-proof, that is within reasonable limits,

even if an AI application running on such a system might hang from the point of view of the case where the proposed matrix formulation according to some problem of interest fails to settle on a valid eigenstate.

So, let's invent a third approach, according to the possible introduction of some type of neural network of the second type that can hopefully be conditioned to create script engines of the first type. Not that others haven't tried doing this with so-called hidden Markov models which concordantly will just as often introduce some kind of Bayesian inference to some hierarchical model. Thus, there have been many attempts at this sort of thing, often with interesting, even if somewhat, at times nebulous results, i.e., WATSON, OMELETTE. So, obviously - something critical is still missing!

Now as it turns out, the human genome consists of about 3 billion base pairs of DNA, each of which encodes up to two bits of information - which might therefore fit nicely in about 750 megabytes for a single set of up to 23 chromosomes, if it can be stored that is, in a reasonably efficient, but uncompressed form. Now if it should turn out that 99% of this does not code for any actual proteins, then it might very well be that all of the actual information needed to encode the proteins that go into every cell in the human body, well that information might only need a maximum of about 7.5 megabytes - and that is for the entire body, not just for the part that encodes how the brain is wired.

O.K. so we haven't quite reduced the design problem of creating a seemingly sentient A.I. to a few lines of APL, but we are getting closer. So how about digital DNA? Whatever that might be?

Of course, if we proceed based on the concept that a real physical brain is typically thought of as being a highly connected network of neurons that exist according to some topology that in turn exists in what is usually thought of as three-dimensional space, then a successful A.I., therefore will need to incorporate some of the features of common CAD packages, for model generation, yet this will need to work according to some concept of geometrization of spacetime by hierarchal representation, and not merely according to some pre-suppositions concerning the principles of low-level symbol manipulation.

This issue has undoubtedly led many down the primrose path of failure, either because they fail to understand the issue at hand, at all - or else they want to prematurely invoke some kind of holographic principle, which might turn out to be necessary for the long run because of certain issues concerning symbolic processing vs. geometrization may very well require some kind of priority inversion. Yet from the point of view of computational memes, one cannot directly infer how such hierarchical refactorings might work, that is, by merely invoking some holistic principle.

Therefore, a new type of compiler is required. So now we are back to square one.

LOG ENTRY:
Art Official Intelligence

And in other news, I am continuing to work on porting the UCSD Pascal compiler to C++, so that I will eventually be able to compile Pascal programs to native Propeller assembly, or else I will implement P-code for the Propeller, or Arduino, or both, or perhaps for a NOR computer. Approximately 4000 lines out of the nearly 6000 lines of the original Pascal code have been converted to C++, that is to the point that it compiles, and is for the most part operational, but in need of further debugging. As was discussed in a previous project, I had to implement some functions that are essential to the operation of the compiler, such as an undocumented, and also missing TREESEARCH function, as well as another function which is referred to as being "magic" but which is also missing from the official distribution - and which is referred to as IDSEARCH. Likewise, I had to implement Pascal-style SETS, as well as some form of the WRITELN and WRITE functions, and so on - amounting to several thousand additional lines of code that will also need to be compiled to run in any eventual Arduino or Propeller runtime library. Then let's not forget the p-machine itself, which I have started on, at least to the point of having some functionality of floating point for the Propeller or Arduino, or NOR machine, etc.

Here we can see that the compiler, which is being converted to C++, is now - finally starting to be able to compile itself. The procedure INSYMBOL is mostly correct and the compiler is getting far enough into the procedures COMPINIT and COMPILERMAIN so as to be able to perform the first stages of lexical analysis.

Now, as far as AI goes, where I think that this is headed is that it is going to eventually turn out to be useful to be able to express complex types of grammar that might be associated with specialized command languages according to some kind of representational form that works sort of like BNF, or JSON, but ideally which is neither - and that is where the magic comes in - like suppose we have this simple struct definition:

```
struct key_info
{
    ALPHA        ID;
    SYMBOL        SY;
    OPERATOR    OP;
    key_info() { };
    key_info(char *STR, SYMBOL _SY,
    OPERATOR _OP)
    {
        strcpy_s(ID,16,STR);
        SY = _SY;
        OP = _OP;
    }
};
```

Then we can try to define some of the grammar of Pascal like this:

```
key_info key_map[] =
{
    key_info("DO",DOSY,NOOP),
    key_info("WITH",WITHSY,NOOP),
    key_info("IN",SETSY,INOP),
    key_info("TO",TOSY,NOOP),
    key_info("GOTO",GOTOSY,NOOP),
    key_info("SET",SETSY,NOOP),
    key_info("DOWNTO",DOWNTOSY,NOOP),
    key_info("LABEL",LABELSY,NOOP),
    key_info("PACKED",PACKEDSY,NOOP),
    key_info("END",ENDSY,NOOP),
    key_info("CONST",CONSTSY,NOOP),
    key_info("ARRAY",ARRAYSY,NOOP),
    key_info("UNTIL",UNTILSY,NOOP),
    key_info("TYPE",TYPESY,NOOP),
    key_info("RECORD",RECORDSY,NOOP),
    key_info("OF",OFSY,NOOP),
    key_info("VAR",VARSY,NOOP),
    key_info("FILE",FILESY,NOOP),
    key_info("THEN",THENSY,NOOP),
    key_info("PROCEDURE",PROCSY,NOOP),
    key_info("USES",USESSY,NOOP),
    key_info("ELSE",ELSESY,NOOP),
    key_info("FUNCTION",FUNCSY,NOOP),
    key_info("UNIT",UNITSY,NOOP),
    key_info("BEGIN",BEGINSY,NOOP),
    key_info("PROGRAM",PROGSY,NOOP),
    key_info("INTERFACE",INTERSY,NOOP),
    key_info("IF",IFSY,NOOP),
    key_info("SEGMENT",SEPARATSY,NOOP),
    key_info("IMPLEMENTATION",IMPLESY,NOOP),
    key_info("CASE",CASESY,NOOP),
    key_info("FORWARD",FORWARDSY,NOOP),
```

```
    key_info("EXTERNAL",EXTERNLSY,NOOP),
    key_info("REPEAT",REPEATSY,NOOP),
    key_info("NOT",NOTSY,NOOP),
    key_info("OTHERWISE",OTHERSY,NOOP),
    key_info("WHILE",WHILESY,NOOP),
    key_info("AND",RELOP,ANDOP),
    key_info("DIV",MULOP,IDIV),
    key_info("MOD",MULOP,IMOD),
    key_info("FOR",FORSY,NOOP),
    key_info("OR",RELOP,OROP),
};
```

And as if, isn't this all of a sudden - who needs BNF, or regex, or JSON? Thus, that is where this train is headed - hopefully! The idea is, of course, to extend this concept so that the entire specification of any programming language (or command language) can be expressed as a set of magic data structures that might contain lists of keywords, function pointers, and special parameters associated therewith, such as additional parsing information, type id, etc.

Elsewhere, of course - I did this - just to get a Lisp-like feel to some things:

```
void SEARCH::RESET_SYMBOLS()
{
    frame &f = SEARCH::m_pFrame;
    symbol_table *t=NULL;
    t = f.cons(keywords)->sort();
     m_keywords = t;
}
```

Essentially constructing a symbol table for the keywords that are to be recognized by the lexer, as well as sorting them with what is in effect, just as if we only needed to write only one line of code! Naturally, I expect that parsing out the AST will eventually turn out to be quite similar and that this is going to work out quite nicely for any language - whether it is Pascal, C++, LISP, assembly, COBOL, or whatever.

LOG ENTRY:
Oh Lazarus, where 'art thou?

The Art Officials never commented about my last post, and it is going to be a while before I actually get any version of Pascal, whether it is Lazarus, or some other version of FreePascal or UCSD actually up and running on the Parallax Propeller P2. So I figure that this might be just as good of a time as any for a quick conversation with Eliza.

Now as it turns out, in an earlier project I was discussing how I have been working on a library called Frame-Lisp, which is sort of a frames-based library of Lisp-like functions that I would like to eventually get running as a back end for ports of ELIZA, and SHURDLU and PARRY and MEGAHAL and pretty much any compiler that I would like to be able to create, invent, or just simply port to other interesting and fun platforms, like Propeller, or Arduino, or FPGA, or pure retro TTL based systems Well, you get the idea. Yet, well then - guess what? It also turns out that I did ELIZA something like 25 years ago, and I recently somehow managed to find the archive of that build and get it running again, sort of. Which of course gives me an idea - since what the original Eliza lacked, like many attempts at creating chat 'bots, is some kind of internal object compiler that could in principle give a language like C/C++ some capacity for new object type creation at run time, which according to some, is considered a form of reflection - which is, of course, going to be necessary, that is if we are going to try to simulate any kind of sentience.

Getting back to the idea therefore of how a compiler should be able to recompile itself is, I think, important. Even while there is also this idea that if the human genome actually consists of only around 20,000 coding genes, of which only about 30% of which are directly involved in affecting the major function of the brain and how it is wired; then I am thinking that the complexity of a successful A.I. that is capable of actual learning might not be as complicated as others are trying to make it. It is simply going to be a matter of trying to build upon the concepts of how compilers work, on the one hand, with an idea toward developing data flow concepts based on the contemporary neural network approach.

Interestingly enough, this particular ELIZA only needs about 150 lines of code to implement, along with about 225 lines for the hard-coded script, i.e., canned dialog and keywords. That is in addition to a few thousand or so lines that are needed to run the back-end lisp-like stuff. So, is it possible that that is where others are failing, that is because they are failing to include essential concepts of compiler design in their approach to A.I.?

Along another line of reasoning, I have never been a particular fan of Maslow's hierarchy of needs, which I won't get into quite yet, other than that I think that Ericson's stages of conflicts throughout life work out much better in sense of how the effects of the critical period notion affect psycho-social development.

Even if Eliza doesn't actually learn, there is still some appeal to writing an AI that can re-compile itself. Hidden Markov models do pretty well up to a point with learning, and then there was M5 of course, in the classic Star Trek, which was programmed by Daystrom with engrams, or so we were told, including the one "this unit must survive."

LOG ENTRY:
Life on Square One - Part II

In an earlier project, I was looking at how It might be possible to get the C/C++ - processor to chow down on Pascal programs, that is if the preprocessor would allow us to do things like temporarily redefining things like the semicolon or equals symbols, and so on - with nested #ifdef's, #undef's and the like. Sort of like this - which doesn't actually work with all of the macros, but it does work with some so that you can at least partially convert a Pascal program to C/C++ by creating some kind of "pascal.h" file and then add #include "pascal.h" in your Pascal code, and then grab the preprocessor output, right? Well, no - but almost, very very almost like this:

```
#define {            /*
#define }            */
#define PROCUEDURE    void
#define BEGIN        {
#define END          }
#define :=          ASSIGN_EQ
#define =        COMPARE_EQ
#define IF            if (
#define ASSIGN_EQ     =
#define COMPARE_EQ    ==
#define THEN        )
#define REPEAT        do {
#define UNTIL        } UNTIL_CAPTURE
#define UNTIL_CAPTURE    (!\
#define ;            );\
#undef            UNTIL_CAPTURE
#define ;           );
#define = [        = SET(\
#define ]        )\
#define )        \
#undef    =        [\
#undef    ]
// so far so good ....
#define WITH        ????????
```

I mean, if someone else once figured out how to get the GNU C/C++ preprocessor to play TETRIS ... then it should be possible to do whatever else we want it to do, even if some other powers claim that strictly speaking the preprocessor isn't fully Turing complete in and of itself, but that it is actually only just some kind of push-down-automation, because of some issues like having a 4096 byte limit on the length of string literals, and so on. Yeah, right - I think I can live with that one if what they are saying, is in effect is that it is probably as Turing complete as anyone might actually need to be.

Still, this gives me an idea that seems worth pursuing, like what does ELIZA have in common with the preprocessor or a full-blown compiler for that matter? Well, the Eliza code from the previous log entry used the following static string tables, arrays, or whatever you want to call them, based on a C++ port of an old C version that was converted from an example that was written in BASIC and which most likely appeared in some computer magazine, most likely, Creative Computing, back in the '70s.

```
char *wordin[] =
{
     "ARE", "WERE", "YOUR", "I'VE", "I'M",
     "ME", "AM", "WAS", "I",
     "MY","YOU'VE","YOU'RE","YOU",NULL
};

char *wordout[] =
{
     "AM", "WAS", "MY", "YOU'VE", "YOU'RE",
     "YOU","ARE", "WERE", "YOU", "YOUR",
     "I'VE", "I'M", "ME",NULL
};
```

This could probably be fixed up a bit - to be more consistent with the methods that I am using in my port of the UCSD Pascal compiler to solve the problem of keyword and identifier recognition, as was also discussed earlier, and for which in turn I had to write my own TREESEARCH and IDSEARCH functions.

```
struct subst
{
    char *wordin;
    char *wordout;
    subst();
    subst (char *str1, char *str2)
    {
        wordin = str1;
        wordout = str2;
    }
};
```

Which should allow us to do something like this - even if this is, as of right now - untested.

```
subst conjugates [] =
{
    subst("ARE","AM"),
    subst("WERE","WAS"),
    subst("YOUR","MY"),
    subst("I'VE","YOU'VE"),
    subst("I'M","YOU'RE"),
    subst("ME","YOU"),
    subst("AM","ARE"),
    subst("WAS","WERE"),
    subst("I","YOU"),
    subst("MY","YOUR"),
    subst("YOU'VE","I'VE"),
    subst("YOU'RE","I'M"),
    subst("YOU","I"),
    subst(NULL,NULL),
};
```

So, I searched Google for Eliza source code, and among other things, I found variations of Weizenbaum's original paper on the subject are now available, as well as variations of things like some kind of language called GNU SLIP, which is a C++ implementation of the symmetric list processing language that the original Eliza was originally written in since it seems that Eliza wasn't actually written in pure Lisp at all, contrary to popular belief! Yet, documentation for the SLIP language looks impossibly bloated, and it just as well warns about having a steep learning curve. So, I won't venture down that rabbit hole, at least not yet, and will prefer instead to continue on the path that I am currently following:

Of course, it should become obvious that Pascal to C conversion might start to look like this:

```
subst pascal2c [] =
{
    subst("{","/*"),
    subst("}","*/"),
    subst("PROCEDURE","void"),
    subst("BEGIN","{"),
    subst("END","}"),
    subst(":=","ASSIGN_EQ"),
    subst("=","COMPARE_EQ"),
    subst("IF","if ("),
    subst("ASSIGN_EQ","="),
    subst("COMPARE_EQ","=="),
    subst("THEN",")"),
    subst("REPEAT","do {"),
    subst("UNTIL","}
       UNTIL_CAPTURE"),
    subst("UNTIL_CAPTURE","(!"),
    subst("= [","= SET("),
    subst("]",")"),
    subst(NULL,NULL),
};
```

With some work to be done with rules so as to implement #ifdef and #undef, or other means for providing context sensitivity, since for now a weird hack is still required that might temporarily require redefining the semicolon so as to property close out UNTIL statements, as well as rules for capturing the parameters to FOR statements, with the WITH statement still being an interesting nightmare in and of itself.

```
WITH (provided ingredients)
BEGIN
  make(deluxe pizza);
  serve(deluxe pizza);
END;
```

Of course, that isn't proper Pascal. Neither is it proper C, but maybe it could be if the variable ingredients was a member of some class which in turn had member functions called make and serve. Welcome to free-form, natural language software development! Well, not quite yet. Still, is too much to ask if the preprocessor can somehow transmogrify the former into something like this:

```
void make_pizza (provided ingredients)
{
  Ingredients->make (deluxe,pizza);
  Ingredients->serve (deluxe,pizza);
}
```

Maybe provided is an object type, and ingredients is the variable name or specialization so that we can in the style of the Pascal language tell the C/C++ preprocessor to find a way to call the make function, which is a member of the provided ingredients class hierarchy, and which in turn can find the appropriate specializations for making not just a pizza, but a deluxe pizza, just as we might call the Pascal WRITELN function with a mixture of

integers, floats, and strings, and then it is the job of a preprocessor, or compiler to resolve the object types, so the WRITELN function will know which sub-specialization to invoke on a per object basis, which I figured out how to do, elsewhere in C++, by using an intermediate s_node constructor to capture the object type in C++ via polymorphism, thus allowing the C style var_args to capture type information, which it can't do, as far as I know in pure C, but as I have shown elsewhere, it can be done in C++, via a hack!

And thus, we inch ever so slowly toward figuring out how to accomplish free-form natural language programming. Obviously, if the meanings of words can be deduced, and therefrom intentions can be ascribed, then it should follow that from the ascribed intentions there can be associated the appropriate objects and methods, if proceeding algorithmically, or else there should also be a way of programmatically generating a corresponding data-flow based approach which can be embodied in the form of some kind of neural network.

I think that the Pascal compiler source might be making an appointment to have a conversation with Eliza sometime in the near future.

LOG ENTRY:
Eliza meets C, whether this is to be, or not to be - we shall see.

Further integration of my 25-year-old C++ port of the '70s vintage Eliza program into the Pascal compiler is moving along nicely. The original code had an important step, referred to as conjugation - wherein words like myself and yourself, or you and I would be swapped. Yet, after realizing there are similarities to this process, and what goes on in the C/C++ preprocessor, I decided to rename the function pre_process, for obvious reasons - since that is one of the directions that I want this project to be headed. So even though I have no desire to learn APL, there is still some appeal to the notion that perhaps an even better ELIZA can be done in just one line of something that conveys the same concepts as APL, as if there is any concept to APL at all.

```
void    ELIZA::pre_process    (const    subst
*defines)
{
    int word;
    bool endofline = false;
    char *wordIn, *wordOut, *str;
    node<char*> *marker;
    process.rewind();
    while (endofline==false)
    {
    word = 0;
    marker = process.m_nPos;
    process.get (str);
    endofline = process.m_bEnd;
    for (word=0;;word++) {
        wordIn = (defines[word]).wordin;
        wordOut = (defines[word]).wordout;
        if (wordIn==NULL)
        break;
        if (compare (wordIn,str)==0) {
        marker->m_pData = wordOut;
        break; }
    }
    }
}
```

Thus, with further debugging, I can see how a function like this should most likely be moved into the FrameLisp::text_object class library, since in addition to being generally useful for other purposes, it also helps to try to eliminate as many references to objects of char* type in the main body of the program as possible, with an eye toward having an eventual UNICODE version that can do other languages, emojis etc. Which certainly should be doable, but it can turn into a debugging nightmare if it turns out to be necessary to hunt down thousands of char and char* objects. Thus, I have created my own node<char*>, node_list<char*> and text_object classes using templates, for future extensions and modifications. Thus, even though ELIZA is kind of broken right now, and is being debugged, this pretty much embodies the simplicity of the algorithm:

```
text_object ELIZA::response ()
{
    int sentenceNum;
    text_object result, tail;
    char *str = NULL;
    node<char*> *keyword,
    *tail_word, *last_word;

    process = textIn;
    pre_process (conjugates);
    return process;

    keyword = find_keyword ();
    sentenceNum = currentReply [key];
    currentReply [key]++;
        if (currentReply[key]>lastReply[key])
        currentReply[key] = firstReply [key];
        result = replies[sentenceNum];
        node<char*> *marker = process.m_nPos;
        if (keyword!=NULL)
                tail.m_nList.m_nBegin      =
        marker;
    else
    tail = "?";

    tail_word = result.findPenultimate (str);
    result.get (str);
    result.peek (str);
    if (strcmp(str,"*")==0) {
    last_word = tail_word->m_pNext;
    delete last_word;
    tail_word->m_pNext = NULL;
    result.m_nList.m_nEnd = tail_word;
    result.append (tail);
    result.append ("?");
    }
    result.m_nPos = result.begin();
    return result;
}
```

Yep, maybe the ELIZA algorithm, with the right text processing libraries just might only take about 40 lines or so of code, with no APL needed or desired. Now testing just the

pre-processing part yields some interesting results. Making me wonder if at least for that part of English grammar analysis, that part of natural language processing is completely solvable.

Interesting stuff. Plenty of stuff to do as of yet. Yet converting Pascal to C, or C to LISP, or LISP to FORTH might turn out to be much easier than it sounds at first blush - even if I meant to say - converting Pascal to C, and C to LISP, and LISP to FORTH, and so on.

LOG ENTRY:
The Road Much Less Travelled.

I cooked up an Eliza-based Pascal source tokenizer and tried using it to see how good it was (is) at doing some of the initial steps in converting the Pascal compiler to C++. Although the initial results seem a bit cringe-worthy, they are not a complete disaster either. So, I got really aggressive in creating a debugging environment for the Eliza-based tokenizer, as well as the original and these results together are looking quite promising. First, a glimpse of the Eliza-based method.

```
void PASCALCOMPILER::SOURCE_DUMP ()
{
    ELIZA eliza;
    text_object source;
    char *buff1, *buf2;
    int line;
    line = 0;
    if (SYSCOMM::m_source==NULL)
    {
        WRITELN(OUTPUT,"NULL source file");
        return;
    }
    else
        if ((*SYSCOMM::m_source).size()==0)
    {
        WRITELN(OUTPUT,"Empty source file");
        return;
    }
    else do
    {
        buff1 = (*SYSCOMM::m_source)[line];
        source = buff1;
        buf2;
        eliza.process = source;
        eliza.pre_process (pascal2c);
        eliza.process >> buf2;
        WRITE(OUTPUT,buf2);
        delete buf2;
        line++;
    }
    while (buff1!=NULL);
}
```

The mostly complete source for this mess can be found of course in the GitHub repositories for this project and will be updated regularly. Be very afraid. Use at your own risk. Guaranteed to contain LOTS of bugs. On the other hand - creating a bunch of debugging code that inspects each symbol as it is parsed, and which selects for things like whatever is found starting with every occurrence of the keyword PROCEDURE and continuing until the first SEMICOLON encountered thereafter - yields a very promising result - which looks (in part) like this.

```
12762: PROCEDURE
12763:  "ASSIGN"
12764: (
12765:  "EXTPROC"
12766: :
12767:  "NONRESIDENT"
12768: )
12769: ;

12859: PROCEDURE
12860:  "GENJMP"
12861: (
12862:  "FOP"
12863: :
12864:  "OPRANGE"
12865: ;

13012: PROCEDURE
13013:  "LOAD"
13014: ;

13017: PROCEDURE
13018:  "GENFJP"
13019: (
13020:  "FLBP"
13021: :
13022:  "LBP"
13023: )
13024: ;

13048: PROCEDURE
13049:  "GENLABEL"
13050: (
13051: VAR
13052:  "FLBP"
13053: :
13054:  "LBP"
13055: )
13056: ;

13078: PROCEDURE
13079:  "PUTLABEL"
13080: (
13081:  "FLBP"
13082: :
13083:  "LBP"
13084: )
13085: ;

13175: PROCEDURE
13176:  "LOAD"
13177: ;

13469: PROCEDURE
13470:  "STORE"
13471: (
13472: VAR
13473:  "FATTR"
13474: :
13475:  "ATTR"
13476: )
13477: ;
```

Now without taking another digression into a discussion of the meaning of the word SELECT, and what might mean in the context of relational databases, it should be easy to see how if all we were to do is to tokenize the input and the select sub-sections according to certain properties, then obviously - this leads to something that looks like it might be handled quite easily by some kind of #define TYPEGLOB_REORDER (A, B, C, ...) macro. Even if I am not proceeding at this point with trying to do a pure preprocessor macro-based language scheme. Somewhere, over the rainbow, maybe someday?

Well, sort of - this is going to be a LONG journey - but things are starting to move very quickly as of late. Writing code is like that - months go by and NOTHING gets done - then in a couple of weekends I write a few thousand lines of code. This should be fun after all.

As if figuring out how to write a completely independent lexer, that works as good as, or better than the original wasn't enough work to do - then there is the notion of how to create ASTs (abstract syntax trees) that not only work with PASCAL, with C/C++, and yet also with standard English grammar, which might contain dialog, or it might contain commands like "KILL ALL TROLLS!", or "Build me a time machine". Oh, what fun.

```
Int PASCALSOURCE::SYMBOL_DUMP   (LPVOID)
{
  size_t i;
  CREATE_SYMLIST(NULL);
  size_t sz = m_symbols.size();
  for (i=0;i<sz;i++)
  {
  DEBUG_SY( m_symbols[i],
  FORSY,DOSY);
  }
  WRITELN(OUTPUT);
  WRITELN(OUTPUT,(int)sz,
  " decoded");
  return 0;
}
```

Yet isn't it nice to contemplate being able to search a project for every FOR statement or every IF-THEN, or to make a list of all of the procedures in the source, to be better able to make sure the conversion is going correctly? Yet why not search "The Adventures of Tom Sawyer" for every reference to whitewash preceded by or followed by fence, or paragraphs that contain the name Injun Joe, and cave or caves in either same, the preceding or the following sentence, paragraph, or context? Seems like a daunting task, but is it? Maybe, or maybe not.

So, let's throw another log on the fire, and do it not with string manipulating functions like strcmp, strcpy, etc., but with abstract functions that can operate on, and transform text objects, whether they are in the form of pure ASCII strings, or tables, or linked lists, or vectors connection maps that link tree structures where the individual nodes of the subtrees point to linked lists or vectors of tokenized, and possibly compressed input which might in turn reference tables of dictionary pointers.

Writing, or re-writing a compiler is quite a chore. Having some interesting code analysis tools makes things a LOT more interesting.

Now, back to killing trolls, and inventing time travel?

Not, quite yet. Let's suppose that we are analyzing real DNA, then one way of doing THAT involves lab techniques that involve things like restriction enzymes, centrifuges, HPLC, CRISPR, DNA chip technology, etc. All so that we can later look at a genome, among other things, and have some way of doing something like "Find sequences that have CATTAGGTCTGA followed by

ATCTACATCTAC or something like that, with whatever else might be in the middle. Like if we had a partial analysis of some fragments of a real protein that we want to learn more about, and we need to find out where in some three billion base pairs that might be encoded, even if that is also in fragments, which might be subjected to later post-translation editing.

Something like this looks VERY doable.

```
DEBUG_GENE ( genome, "CATTAGGTCTGA" , "ATCTACATCTAC" );
```

Just in case that sort of thing might be useful to someone.

Suffice to mention, also, that if you have been programming long enough, then you know what it is like to sprinkle your code with 1000's of TRACE statements, or trying to pipe debugging information to a logfile with fprintf statements, and all of the hassle that goes into creating the format strings, setting up and cleaning up buffers for all of that, and so on. When PASCAL does it so nicely - like this --

```
WRITE (OUTPUT,' ',SYMBOL_NAMES2[p.SY]);
WRITE (OUTPUT,'(',p.VAL.IVAL,')');
```

Letting us use the PASCAL-style WRITE and WRITELN functions, which are perfectly happy to accept strings, characters, integers, floats, etc., and without needing all of the Sanskrit.

This brings me back to LISP since it is so easy to imagine something like this:

```
for_statemtns                                   =
CONS(MAPCAR(DEBUG_SY(m_symbols,FORSY,DOSY)))
;
```

That is to say, instead of simply writing out the debugging information, we could re-capture the snippets from the token stream and then construct the new meta-objects in such a fashion, to make them more suitable for additional processing, as if wanted to re-parameterize the FOR statements from a Pascal program, to be able to convert them to C/C++ style for statements, just not with a statement converter that is written explicitly to convert for statements. but which works more like regex in PERL, where if we could find snippets that need conversion, then we could more easily offer regex snippets that are appropriately sanitized (where was I reading out that?), on the one hand, yet while remaining perfectly capable of working with other data types, like actual DNA sequences, that is to say - without loss of generality.

Thus, when contemplating how digital DNA might work, in the context of so-called deep learning neural network models, it should now become more clear how the interaction between traditional programming and deep learning might result in several orders of magnitude improvement in the computational effectiveness, that is so as to address not just the inefficiency of contemporary efforts at A.I.

Now for whatever it's worth, let's take a look at how well ELIZA does, at least at first blush at converting the definition of the pascal RECORD type contained in the original compiler source code to C or C++. Obviously, this is neither C or C++, nor is it Pascal, but it does look interesting. Obviously, some kind of TYPEGLOB_REORDER(A,B,C,...) might come in use here if we wanted to figure out how to convert this nightmare, entirely within the framework of the C/C++ preprocessor, i.e., with nothing more than #defines and the other stuff that the pre-processor allows. Yet, I don't think that that is necessary, and I am not trying to turn the preprocessor into a standalone compiler, translator, bot engine, or whatever. Suffice to say that it is sufficient for me to know if

someone else could get Conway's game of life to run in one line of APL and if yet someone else could get a Turing complete deterministic finite automata to run within Conway, and if Tetris will run in the C++ preprocessor using ANSI graphic character codes, etc., then it pretty much follows that someone could probably shoe-horn Conway into the pre-processor, and then write a program that runs under Conway, that accepts a collection of statements according to some grammar, which we might call "A", and transmogrifies it according to some set of rules, such as might be referenced according to some set ["Q","R","S"...], and so on, so as to transform "A"->"B". Yeah, Yeah, Yeah, Yeah. Or so the theory goes, insofar as context-free grammars go.

```
struct STRUCTURE
{
    ADDRRANGE SIZE;
    FORM STRUCTFORM;
    union
    {
        SCALAR : ( union ( SCALKIND : DECLKIND
)
        DECLARED : ( FCONST : CTP ));
        SUBRANGE : ( RANGETYPE : STP ;
        MIN , MAX : VALU );
        POINTER : ( ELTYPE : STP );
        POWER : ( ELSET : STP );
        ARRAYS : ( AELTYPE , INXTYPE : STP ;

        union ( AISPACKD : bool )
        TRUE : ( ELSPERWD , ELWIDTH : BITRANGE
;

        union ( AISSTRNG : bool )
            TRUE :( MAXLENG :
            1 .. STRGLGTH )));
        RECORDS : ( FSTFLD : CTP ;
        RECVAR : STP );
        FILES : ( FILTYPE : STP );
        TAGFLD : ( TAGFIELDP : CTP ;
        FSTVAR : STP );
        VARIANT : ( NXTVAR , SUBVAR : STP ;
        VARVAL : VALU;
    )
};
```

Actually, now that I think about it, I don't think that I particularly like context-free grammars. Contextuality is actually good! We need contextuality. Yet, as was discussed earlier, one way to try to trick the pre-processor into capturing parameters to function and procedure declarations might be to sometimes redefine the semicolon at the end of a line or statement, temporarily as in "#define ;));" so as to in effect add a closing parenthesis to other substitutions, like "#define UNTIL } while (|(" which does some of the work for converting REPEAT... UNTIL blocks, but it needs help with captures and closures. Yet with code and not a lot of code, we can do this:

Now we are running our DEBUB_SY function with variables, instead of const parameters. - and we have specified that we are interested in capturing globs or whatever might be found in between sets of parentheses. With LOTS of work yet to be done. Yet here we have a potentially very interesting, and yet very simple solution to the problem of providing principles of capture and contextuality within the same method, even while providing a method that should work, or be easily extensible to work in even more general cases.