A tour of my wannabe C compiler [WIP]

by blockeduser (N.B. I'm a hobbyist not an expert !), December 2013

As in many programs, everything starts in `main.c`. The bulk of that file is boring business like parsing command-line options, and setting up dynamic-sized buffers for the input code (which is read from `stdin`). The real relevant part starts when `main.c` occurs when the lexical analysis module ("tokenizer") is "set up" and run on the input code buffer:

```
setup_tokenizer();
tokens = tokenize(buf);
```

Okay, now, the lexical analyzer ("tokenizer"), which is defined entirely in the file `tokenizer.c`. Well, the crucial data structure which makes it works is an "NFA" (I think that's the term), which are a kind of graph where the edges are of two kinds:

1. "character edges", which are followed as characters are consumed from the input;
2. "epsilon edges", which may be followed when the current character doesn't match any of the character edge's characters. For reasons to be discussed later, nodes can only have one epsilon edge.

The nodes/vertices in this graph are a "state" which tells you whether you have matched a pattern at this point or not (an integer is sufficient to store this "state" information, along with the convention that when it's zero it means it's a non-matching state).

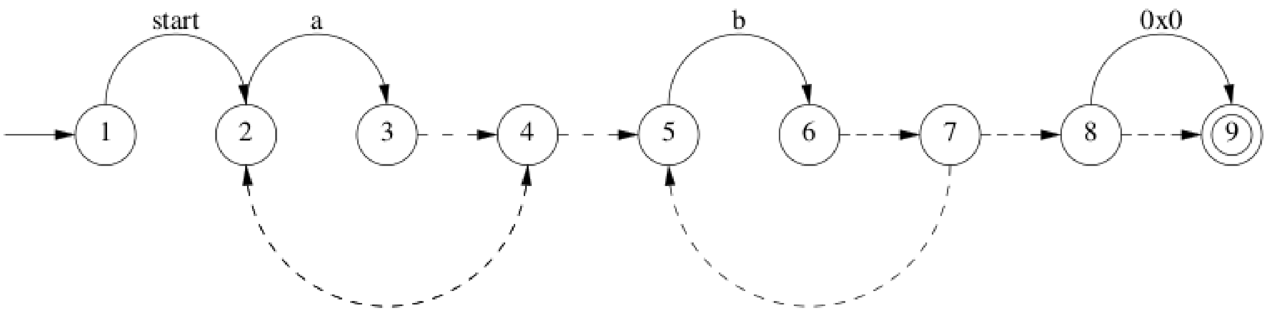
The following `struct`, found in `tokenizer.c`, defines an NFA node:

```
typedef struct nfa_struct {
     /* character edges */
     struct nfa_struct* map[256];

     /* epsilon edge */
     struct nfa_struct* link;

     /* accept state, if any (0 otherwise) */
     int valid_token;
} nfa;
```

As an example, below is an illustration of an NFA for the `egrep`-style regex pattern "`^a*b+$?`":



Epsilon edges are represented using dashed arrows, nodes using numbered circles, and character edges with labelled arrows. The unique accept-state of this NFA is a doubly-circled node (the 9th one).

Note that in the above example, there are multiple epsilon edges for some nodes, unlike in the convention used in the tokenizer, which has a maximum of one epsilon edge per node, but that's a trivial distinction anyway.

Anyway, in the tokenizer, there are about three or four big NFA structures which each contain several patterns. These patterns are split among the NFAs such that any edge traversal of any given NFA will deterministically lead to either a matching node or fail, without the need for backtracking. This is possible by maintaining patterns which are sub-strings of each other (like the operators `&` and `&&`) in different NFAs. The reason for this is time efficiency: backtracking is slow and bad. In an earlier version of the code the NFAs were backtracking; converting the code to this non-backtracking scheme made the tokenizer faster by some non-arbitrary amount.

The routine `setup_tokenizer()` manually splits up the patterns, which are written in a pseudo-regex language, into three big NFA's. It consists in a series of calls to the routine `add_token()` which "compiles" a pseudo-regex string pattern into a bunch of NFA edges and nodes which are injected into a provided NFA node pointer.

Each of these big NFA's is stored as a distinct index of the array "`t`" and is initially allocated in a little for-loop at the beginning of the routine:

```
for (i = 0; i < 3; ++i)
     t[i] = new_nfa();
```

Once these structures are set up, the tokenizer's job is to take the input code buffer, which is a sequence of characters, and convert it to a sequence of tokens. This is where the algorithms operating upon the NFA data structure come into play. Well, there's only one main one, and it's the matching algorithm, implemented in the routine `match()`. What it does is pretty straightforward: given an NFA structure (recall that there are 3 big ones), it tries to reach a matching-state node by traversing character edges or epsilon edges, and either makes it to such a node or fails. The main lexing loop just calls it in a loop on the 3 big NFA's until a match if found, at which point the corresponding token is adding to the out-coming sequence of tokens. If and when a match is found, the character pointer is advanced in the main lexing loop and the process is repeated from the new offset. All-in-all, the matching algorithm code is not so bad:

```
/*
 * nfa* t                :     matching automaton / current node
 * char* full_tok        :     full text
 * char* tok             :     where we are now
 */
match_t match(nfa* t, char* full_tok, char* tok)
{
      int i = 0, str_const = 0;
      char c;
      match_t m;

      /* Iterate the matching loop as long as there are
       * characters left */
      while (i <= strlen(tok)) {
            /* Syntax sugar for "current character" */
            c = tok[i];

            /* HACK to deal with \" in string constants */
            if (c == '"')
                  str_const = 1;
            if (c == '\\' && str_const) {
                  i += 2;
                  continue;
            }

            /* If a character edge for the
             * current character exists,
             * follow it. */
            if (t->map[c]) {
                  t = t->map[c];
                  ++i;
            }
            /*  Otherwise, check for an epsilon edge
             * and follow it if it exists */
            else if (t->link) {
                  t = t->link;
            /* Otherwise: no epsilon, no character, no match;
             * stop the loop */
            } else
                  break;

            if (t->valid_token)
                  break;
      }

      /* Set up the result-report structure */
      if (t && t->valid_token) {
            /* Match succeeded */
            m.success = t->valid_token;
            m.pos = tok + i;
      } else {
            /* No match */
            m.success = 0;
      }

      return m;
}
```

Well, other than that, the tokenizer module has some ugly state machine stuff to deal with being either inside or outside of a comment, and it’s not so clean in doing the main control loop part that calls the matching algorithm, unfortunately. Anyway, the result of the whole process is a pointer to an array of `token_t` structures, which are defined as follows in the header file `tokenizer.h`:

```
typedef struct token {
     char type;
     char* start;
     int len;

     /* for elegant parser diagnostics */
     int from_line;   /* originating source line */
     int from_char;   /* originating offset in that line */
} token_t;
```

`start` and `type` give the token string, which is useful for things like identifiers where it matters, but not useful for operators like + which have their own token type code. The `start` pointer is simply an index into the original input code buffer, and `len` gives the length of the token string in the buffer from this offset pointer. Finally, `from_line` and `from_char` are used to track where a given token came from, which allows pretty error diagnostics like this:

```
int int int;
    ^
error: line 1: identifier expected
```

Note that these kinds of fancy errors sometimes may occur at any of the compilation stages, which are (here) lexing, parsing, and codegen’ing. In the specific example above, it was at the parsing stage, but it could have been one of the other two.

Anyway, the next step is to parse the stream of tokens into a parse tree. This is programmed thusly in `main.c`:

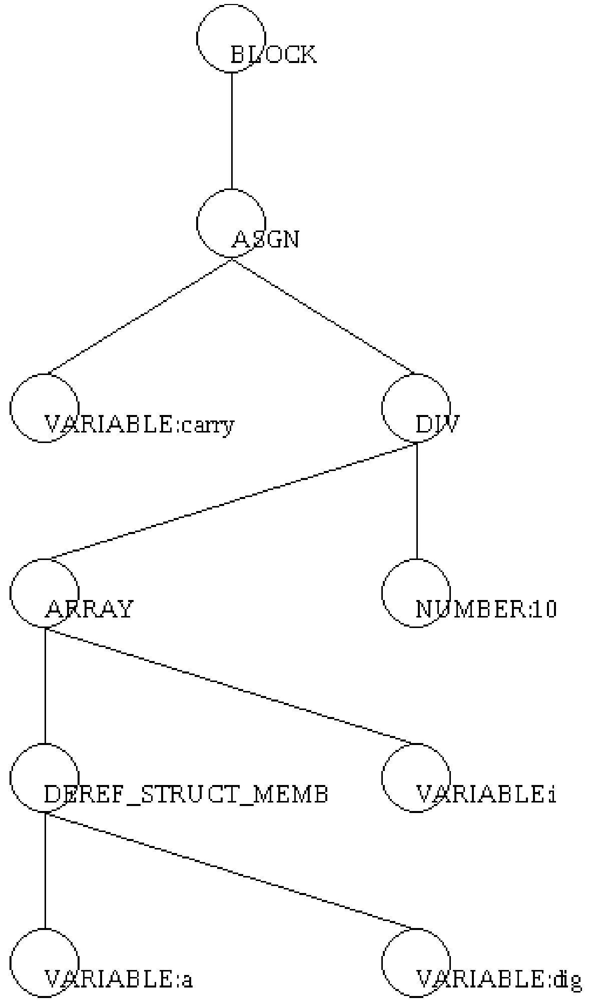
```
tree = parse(tokens);
```

The parser resides in `parse.c` and is a pretty classical recursive descent thing, hand-implement by my noobself, probably noobishly, based on a grammar described in `grammar.txt`.

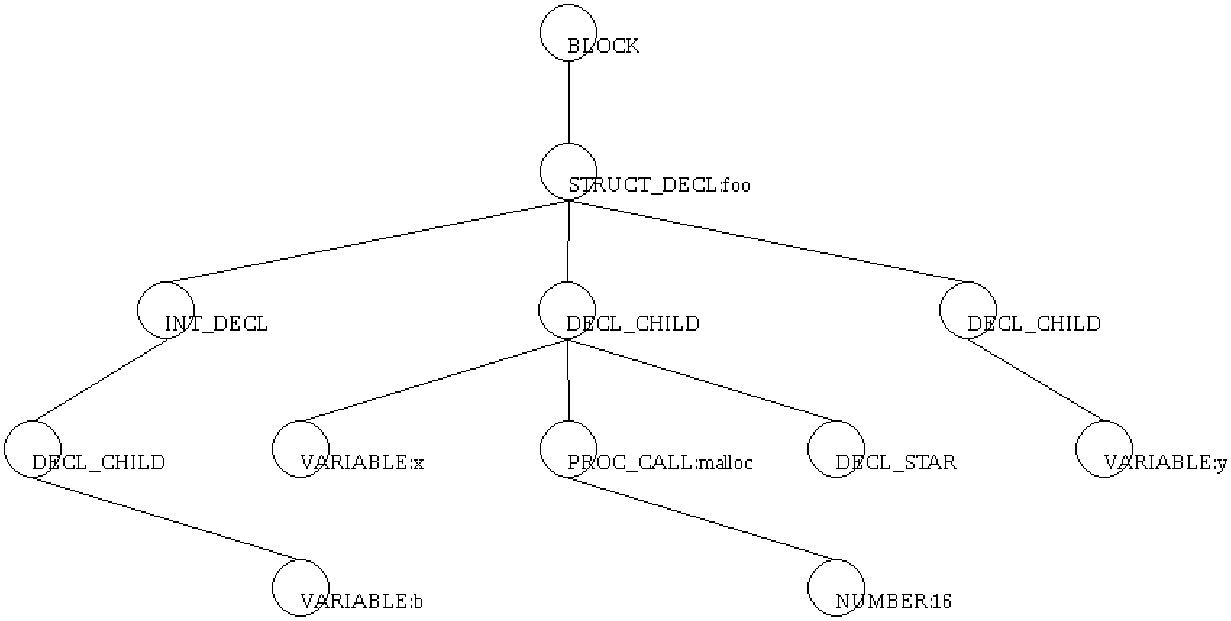
Each of the constructions in the grammar maps to one routine in the parser code module. Some backtracking happens sometimes.

Here are renderings of a few example parse tree for some example code snippets (made by using the `--ast` flag of the compiler and piping into `drawtree` another program I wrote)

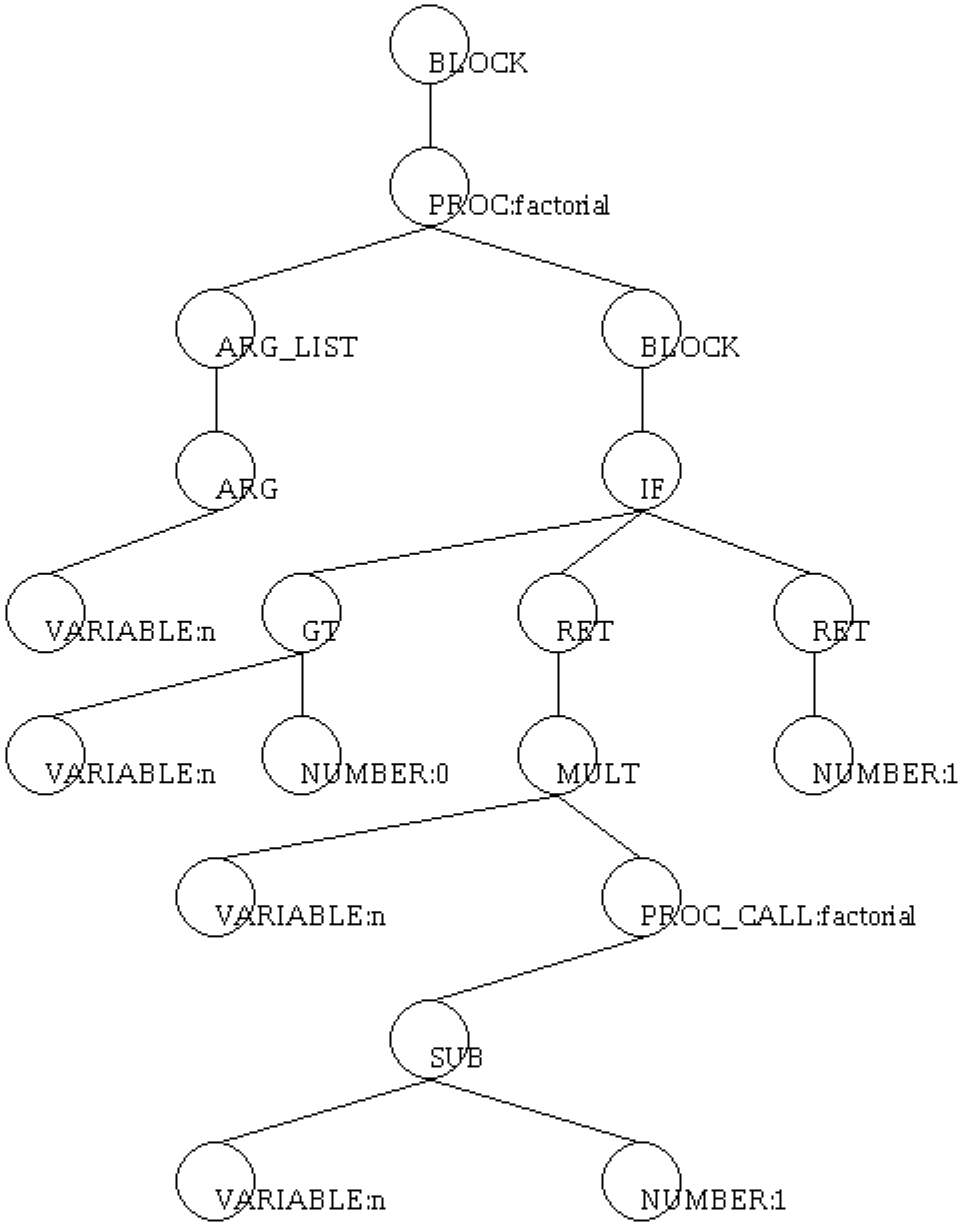
```
carry = a->dig[i] / 10;
```



```
struct foo {
      int b;
} *x = malloc(16), y;
```

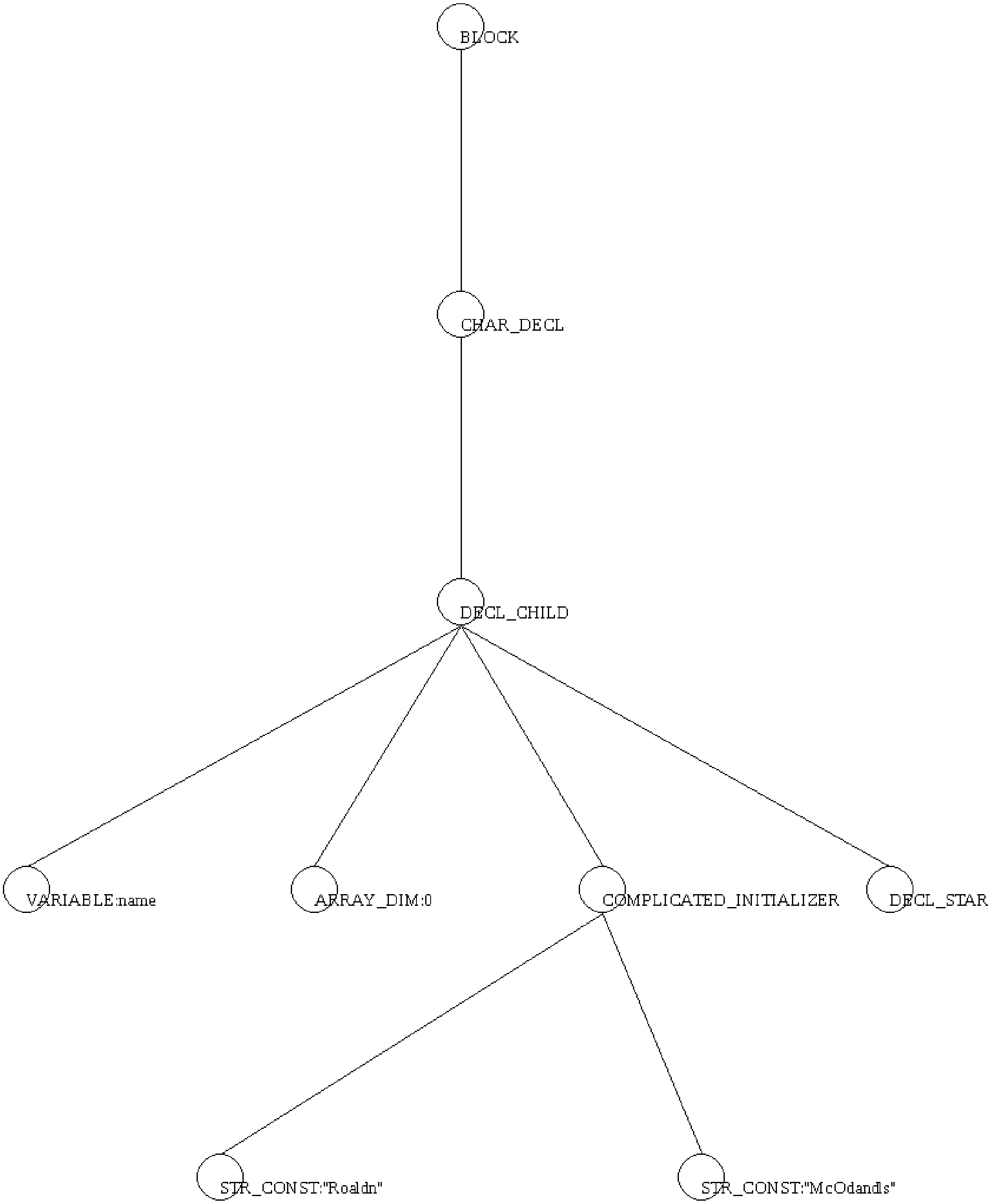


```
factorial(n) {
     if (n > 0)
          return n * factorial(n - 1);
     else
          return 1;
}
```



A stranger example:

```
char* name[] = {"Roaldn", "McOdandls"};
```



The next phase after parsing is a single codegen module per target architecture. (There are no further intermediate representations between the parse tree and the final assembler code, since this is my first time doing this and I'm kind of lazy and unexperimented, "eager to get a quick babby's first compiler running fast", etc).

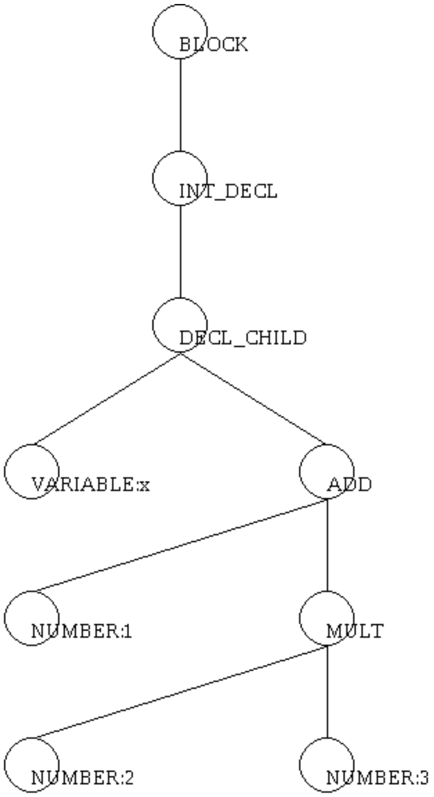
Supported targets currently are:

- intel i386 aka x86 (32-bit) assembler, to be assembled using gnu assembler ("`gas`")

- "BPF VM" (a crippled register-based VM I designed when I was 13; unfortunately the codegne for this target doesn't support later-added-to-the-project stuff like procedures, recursion, pointers, and structs, and it wouldn't be easy or possibly even feasible anyway) – the output is a kind of pseudo-assembler for which i wrote an assembler ("`bpf-asm`") available in another directory of the source tree. In the earlier stages of programming the compiler (the first 1 week or so in Dec. 2012) this was the only target.

Well, the main algorithm in the x86 codegen (which lives in `codegen_x86.c` and `typeof.c`) is a recursive tree walk routine called `codegen()`, which writes codes for operations and returns a string for an a register or stack index address containing the result of the computation. In general, the code for the leaves gets written first, and from there on addresses percolate up to increasingly higher parser nodes which are coded after the leaves. More specifically, semantics of various constructs guide the specific order of traversal. For example, statement-sequencing block coding ("`{a; b; c;}`") leads to the traversal of the block node's children one-by-one in order.

Consider for example the simple code fragment "`int x = 1 + 2 * 3;`". It produces the following parse tree:



Which the x86 codegen turns to the following GNU i386 assembler fragment (which is kind of dumb and inefficient but there you go this is my first and crude compiler):

```
1           movl -8(%ebp), %eax
2           movl $1, %eax
3           movl %eax, -8(%ebp)
4           movl $2, %eax
5           imull $3, %eax
6           movl -8(%ebp), %ebx
7           addl %eax, %ebx
8           movl %ebx, -8(%ebp)
9           movl -8(%ebp), %eax
10          movl %eax, %esi
11          movl %esi, -4(%ebp)
```

Instruction 1, `movl -8(%ebp), %eax`, seems like some kind of error from the register / stack memory juggling mechanism that tries to prevent running out of registers when evaluating arithmetic. Oh well.

Instruction 2, `movl $1, %eax`, puts the "NUMBER:1" leaf into the EAX register temporarily, then instruction 3, `movl %eax, -8(%ebp)`, stashes the result from the EAX into the stack offset address `-8(%ebp).`

Instructions 4 and 5 are the result of coding the multiplication node. They compute the result of the multiplication node into register EAX and return that to their parent node as the address of the result.

Instruction 6 is back up at the ADD node. It reloads the old half-computed addition result (at this point, 1) from stack offset `-8(%ebp)` back into a register, EBX.

Instruction 7 adds the result of the multiplication node (held in EAX) to the EBX register which held the half-completed addition. Now the addition is complete and the result is in EBX.

The next few instructions spend a stupid time getting that result from EBX into the stack offset `-4(%ebp)` which is actually what was probably allocated as the stack offset for the variable "`i`".

Clearly, this is stupid as shit output code, and with some more intermediary phases, perhaps some data structure for sequences of register operations, and some kind of optimization algorithm for which the data structure would be well designed, there wouldn't be code as bad as this. But this is my first time writing a compiler and I kept things stupidly simple.

The register allocation is also very noob retarded stupid. It could basically be described as "stupid" and "greedy" and "unplanned". No graph colouring, planned register allocation etc. to be seen in this code! As such, the x86 codegen just barfs out very-unoptimised x86 assembler, without much planning, and without any kind of backtracking or backpatching.

Anyway, apart from this simple tree walk algorithm the codegen does some more dirty nasty stuff, like keep track of function declarations, keep track of and compute the types of arbitrary nodes (lots of this is in `typeof.c`), keep track of local and global symbols, allocate stack offsets for local variables, issue assembler directives to write core data for global variables with initializers, etc.

As a sample, here's a code snippet for probably one of the least messy-to-code codegen nodes:

```
if (tree->head_type == CC_AND) {
      my_ccid = ccid++;
      sto2 = get_temp_reg_siz(4);
      printf("movl $0, %s\n", sto2);
      for (i = 0; i < tree->child_count; ++i) {
            sto = codegen(tree->child[i]);
            printf("cmpl %s, %s\n", sto, sto2);
            printf("je cc%d\n", my_ccid);
            free_temp_reg(sto);
      }
      printf("movl $1, %s\n", sto2);
      printf("cc%d:\n", my_ccid);
      return sto2;
}
```

Another short but interesting example is the code for a generic IF-statement node:

```
if (tree->head_type == IF) {
      lab1 = intl_label++;
      lab2 = intl_label++;
      /* codegen the conditional */
      sto = codegen(tree->child[0]);
      /* branch if the conditional is false */
      str = registerize(sto);
      str2 = get_temp_reg_siz(4);
      printf("movl $0, %s\n", str2);
      printf("cmpl %s, %s\n", str, str2);
      free_temp_reg(sto);
      free_temp_reg(str);
      free_temp_reg(str2);
      printf("je IL%d\n",     lab1);
      /* codegen "true" block */
      codegen(tree->child[1]);
      /*
       * Check for an "else-return" pattern;
       * if it is encountered, the "else"
       * label and jump are not necessary
       */
      else_ret = tree->child_count == 3
       && tree->child[2]->head_type == RET;
      /*
       * Jump over else block if there
       * is one
       */
      if (tree->child_count == 3
            && !else_ret)
            printf("jmp IL%d\n", lab2);
      printf("IL%d: \n", lab1);
      /* code the else block, if any */
      if (tree->child_count == 3)
            codegen(tree->child[2]);
      if (!else_ret)
            printf("IL%d: \n", lab2);
      return NULL;
}
```

As such, the assembler is doing a lot of the work (for example, *it* deals with forward label jumps).

A lot of the messier aspects of the codegen have lots of annotated comments in the relevant source files. And the entry point and overall call graph is pretty well marked.

But maybe i should write about some of the messier arcane aspects like how it compiles structs or declarations ... or declarations of structs

TODO: maybe write more stuff.
This is WIP !...