

Forum:[Hack Forums](#)

Thread: [\[Advanced\] \[C\] Writing a Metamorphic Engine / Real USG; Part 2](#)

movsx - July 23, 2017, 11:51 am

Introduction

Part 2 of this tutorial is about parsing and use of syntax directed translation to convert our parse tree into an abstract syntax tree. In parsing, we take a sequence of tokens from the lexer stage and use it to build a parse tree. Not only that, but we have to confirm that the input forms a valid grammar for the language (we will discuss semantic analysis in later tutorials).

So, last tutorial, we ran our input through a lexer. It then broke this input into tokens. Say we want to lex the math expression:

"1 + 3132 1337" This is a character string of length 15

A lexer would spit out {NUMBER, +, NUMBER, *, NUMBER}, an array of size 5. It's also classified 1, 3132, and 1337 as "NUMBER" Now our job as a parser is easy, because we've broken down the input into something smaller. Now we feed this input into the parser to generate a parse tree.

The idea behind parsing is we define our language grammar in terms of a context free grammar. However, we need to do a bit more than that, because C isn't a context free grammar, it actually depends on context. For example, let's look at the following C code:

Code Token * A;

Is it an expression or is it a declaration? We can't determine this from the context alone. It is an expression if Token is an identifier. If Token is a type, then it's a declaration. We need a symbol table to answer this specific question (discussed in part 3). So to build a C parser, we use context free grammars in addition with some extra code thrown in to help us determine the difference between issues as discussed above.

Crash Course on Context-Free Grammars

Some definitions you'll want to understand:

Terminal symbols: Symbols which cannot be replaced.

Non-terminal symbols: Symbols which can be replaced.

To break this down even more, let's say a "terminal symbol" is a token that comes out of our lexer. A "nonterminal symbol" is a group of tokens in our parser. Production rules are also non-terminal symbols.

A production rule is a rule which tells us we can take a group of nonterminal and terminal symbols and replace them with this specific nonterminal. I.e.,

E -> NUMBER + NUMBER

This production rule says that if we see NUMBER + NUMBER in our input, we can replace that with E. So how do we parse something? Simple. We have a start derivation rule and collection of derivation rules. In a top-down parser, our goal is to go from the start derivation rule and apply derivation rules until we match the input sequence. There's also a special symbol called "epsilon" which means the empty token. I.e., we can use this derivation to match essentially nothing.

As an example, suppose we have this grammar, with the start derivation rule being "S" (NOTE the pipe "|" is a special character which means "OR". It means E derives E + E OR E E OR the terminal number.

```
S -> E
E -> E + E | E E | number
```

Now say we have the input "1 + 1 2"

We have a tuple that represents our current state. So (S, "") represents we've started from the start symbol, and "" represents our current match.

```
(1) (S -> E, "E")
(2) (E -> E + E, "E + E")
(3) (E -> E E, "E + E E")
(4) (E -> number, "number + E E")
(5) (E -> number, "number + number E")
(6) (E -> number, "number + number number")
```

As you can see, we've also generated our parse tree while doing this. The parse tree is essentially the rules we've applied to go from the start symbol to the matching output!

!!!!!! In step (2) what happens if we applied E -> E E instead of E -> E + E? Then our parse tree would change the expression from 1 + 1 2 to (1 + 1) 2. This is very bad. We address this later in this tutorial.

But there are FOUR things to notice:

- (1) We've applied our rules and replacements from LEFT to RIGHT.
- (2) There are multiple derivations we can use to reach the same output! (Try the above on 1 + 2 + 3 by yourself) This is BAD as it means our grammar is ambiguous! We need to write an unambiguous grammar.
- (3) How to handle operator precedence? (Discussed later)
- (4) This grammar is LEFT RECURSIVE. Look at E -> E + E. It means essentially the leftmost symbol in its production is itself. I've highlighted this symbol in red. This will cause problems in recursive descent parsers.

LL(1) Grammars

LL(1) grammars mean we can scan the input from left to right, deriving the parse tree from top-bottom and using at most 1 lookahead symbol at the same time. This introduces a problem. We cannot use recursive descent parsers to parse left recursive grammars. Say we have the grammar from above:

```
S -> E
E-> E + E | E E | number
```

How we would implement this in a recursive descent parser, is we'd have a function for each production rule. For instance, in pseudocode

```
Code bool E() {
    if (E() && '+' && E()) // BIG PROBLEM!!!
        ....
}
```

So above we can see the issue is E calls itself without consuming any input. Oh oh. This will cause a stack overflow. We can fix this issue by converting our left recursive grammar into a right recursive one. There's also another issue we need to fix: precedence. We do this by wrapping lower precedence production rules at "higher" levels than higher precedence production rules.

So above, to fix precedence:

```
S -> E
E -> E + E | E E | number
```

We convert this into

```
S -> E
E -> E + T | T
T -> T number | number
```

But we still have a problem. This grammar is left recursive. Think hard about this. You'll note that the below grammar is equivalent AND right recursive.

```
S -> E
E -> T + E | T
T -> number T | number
```

CFG for a subset of C

parser.h contains the CFG for a subset of C. Read my comments in the parser.h code, it explains the issue in more detail. It also discuss the dangling else problem, and how the top down parser automatically solves this issue.

The dangling else problem is this: In the following code, which "IF" does the "ELSE" belong to? Line 1 or 2? (It is line 2, but as you'll note, depending on our grammar, it is ambiguous, this causes problems, but we can fix it in our actual parser, without modifying the grammar (both solutions work fine, though))

```
Code 1. if ()
2. if ()
3. else ...
```

parser.h

<https://pastebin.com/C3dsqYpS>

Syntax Directed Translation

We want to directly output an abstract syntax tree from our recursive descent parser. Not a parse tree. So we use syntax directed translation to do this. I won't discuss the theory, it can span multiple CS/EE courses. But I'll tell you what it does, and you can see from my code how it works.

Parse trees are ugly as fuck and hard to work with. Just imagine how ugly it'd be for an even larger input. We want to make it pretty and transform the right tree into the left one.

We define a struct for our AST node to make it convenient to work with. Now might be a good time to look into unions if you're not familiar with them.

astnode.h

<https://pastebin.com/xEqu0mTU>

And now, here's the code for the recursive descent parser. Read the comments, they offer some light. And take note to comments about error checking. Also specifically look into backtracking. What is backtracking? Our parser has to backtrack. Why? This isn't covered in the tutorial, I expect you to figure it out, and perhaps talk about it in the comments. Look at my code. See how I backtrack.

parser.c

<https://pastebin.com/cfyESE4B>

Here's an example input:

```
Code {  
    int x;  
    x = 0;  
    while (x < 10) {  
        x = x + 1;  
    }  
}
```

Our lexer spits out:

```
Code [{] [INT] [Identifier x] [;] [Identifier x] [=] [Int Literal=0] [;] [WHILE ] [{] [Identifier x] [<] [Int  
Literal=10] [}] [{] [Identifier x] [=] [Identifier x] [+] [Int Literal=1] [;] [}] [}]
```

Our parser spits out an AST:

```
Code BLOCK  
  DECLARATIONS  
    DECLARATIONS  
      DECLARATION  
        TYPE INT  
        ID x  
  STATEMENTS  
    ASSIGN  
      ID x  
      LITERAL 0  
  STATEMENTS  
    WHILESTMT  
      BINOP <  
        ID x  
        LITERAL 10  
      BLOCK  
        DECLARATIONS  
        STATEMENTS  
          ASSIGN  
            ID x  
            BINOP +  
              ID x  
              LITERAL 1  
          STATEMENTS  
        STATEMENTS
```

Next Tutorial

The next tutorial will be posted next weekend, when I have more time. My last tutorial got a lot of hate. If people are going to hate on my tutorials, and expect them to be something they can just copy paste into their own projects, without any effort whatsoever to learn the topic at hand then there won't be a third. I've already written the metamorphic engine. This tutorial took me 2 hours to write. Time is valuable, and I won't spend it where it's not appreciated.

Zens - July 28, 2017, 5:42 am

Great tutorial, bookmark this. Will read this later

HiddenEyE - July 28, 2017, 7:21 am

great tutorial OP. nice to see good thread this days in c++

Hoodie+ - July 29, 2017, 6:29 am

Very interesting tutorial.

Good job and thank you!

6577657534735 - July 29, 2017, 3:18 pm

Cool thread

Does this actually work? I'd imagine this will easily bypass signature based AV but it would probably have a hard time bypassing runtime heuristic based AV since the majority of malware actions are not computational or logic based but system call based and the runtime profile of system calls used will still be the same regardless of how much junk code and business logic is rearranged

movsx - July 29, 2017, 9:51 pm

(Jul 29, 2017 03:18 PM)Abraluna Wrote: [\[CITE\]](#) Cool thread

Does this actually work? I'd imagine this will easily bypass signature based AV but it would probably have a hard time bypassing runtime heuristic based AV since the majority of malware actions are not computational or logic based but system call based and the runtime profile of system calls used will still be the same regardless of how much junk code and business logic is rearranged
Unfortunately, it is a major problem for us because malware authors can easily not copy paste or use obvious syscall sequences that flag antiviruses. So the metamorphic engine would easily get past antiviruses.

Metamorphic engines are a real threat to computer security and hopefully this motivates people to find ways to defeat them.

If people are interested, I might write up the third part of the tutorial which would cover syntax directed translation and semantic analysis. It will be a very wordy tutorial, though, unfortunately.

Hoodie+ - July 30, 2017, 6:58 am

(Jul 29, 2017 09:51 PM)movsx Wrote: [\[CITE\]](#) Unfortunately, it is a major problem for us because malware authors can easily not copy paste or use obvious syscall sequences that flag antiviruses. So the metamorphic engine would easily get past antiviruses.

Metamorphic engines are a real threat to computer security and hopefully this motivates people to find ways to defeat them.

If people are interested, I might write up the third part of the tutorial which would cover syntax directed translation and semantic analysis. It will be a very wordy tutorial, though, unfortunately. There isn't a simple or easy solution to bypass metamorphic engines.

That's why everything is moving from blacklisting technique to whitelisting technique.

For example Smart Screen or the new Windows 10 versions which allows only to download software from the App Store and your software needs to be verified to be there.

Personally, I'm not a fan of the white-listing approach.

movsx - July 30, 2017, 7:50 am

There's also a really big problem that needs to be solved here that I'm shocked nobody caught. Its regarding associativity. Can anyone catch it and propose a solution? (There's 2 solutions).

HiddenEyE - July 30, 2017, 8:56 am

(Jul 30, 2017 06:58 AM)Hoodie+ Wrote: [\[CITE\]](#) There isn't a simple or easy solution to bypass metamorphic engines.

That's why everything is moving from blacklisting technique to whitelisting technique.

For example Smart Screen or the new Windows 10 versions which allows only to download software from the App Store and your software needs to be verified to be there.

Personally, I'm not a fan of the white-listing approach.

i agree with you mate. But microsoft will always suffer from blackhat guys. cause each security measure they implement results to new Oday from they bad guys.

0xDEAD10CC - July 30, 2017, 6:27 pm

Code static inline void format(int num)

```
{
    int i;
    for (i = 0; i < num; i++)
    {
        printf("\t");
    }
}
```

static void printast_helper(struct astnode *node, int depth)

```
{
    if (node == NULL)
        return;

    switch (node->node_type)
    {
        case T_BLOCK:
            format(depth);
            printf("BLOCK\n");
            printast_helper(node->block.declarations, depth + 1);
            printast_helper(node->block.statements, depth + 1);
            break;

        case T_DECLARATION:
            format(depth);
            printf("DECLARATION\n");
            printast_helper(node->declaration.type, depth + 1);
            printast_helper(node->declaration.identifier, depth + 1);
            break;
    }
}
```

...

Just some updates on this code, there's other things of course... You can use printf without that

loop to get what you're looking for using width specifiers. And because the `format()` function is called within all case statement blocks, you can probably move that out of the switch statement scope to avoid that boilerplate.

Specific # of spaces:
Code `#include <stdio.h>`

```
int main(void)
{
    int depth = 10;
    printf("%*c%s\n", depth, 10, "testing 1");
    printf("%*c%s\n", depth, 10, "other 1");
    return 0;
}
```

Something like:

```
Code void print_p(const char *msg, int depth)
{
    printf("%*c%s", depth, ' ', msg);
}
```

Varadics would be another step, but not needed here.

movsx - July 30, 2017, 7:09 pm

(Jul 30, 2017 06:27 PM)0xDEAD10CC Wrote: [\[CITE\]](#) Just some updates on this code, there's other things of course... You can use `printf` without that loop to get what you're looking for using width specifiers. And because the `format()` function is called within all case statement blocks, you can probably move that out of the switch statement scope to avoid that boilerplate.

Varadics would be another step, but not needed here.

Sure, I could have done it like that. It's really not high priority for me though.

Since you're interested in reviewing the code, let's talk about what is wrong about the parsing method (it will give us incorrect results for some inputs) and how we will have to solve it in the next semantic analysis phase of the front end. Or how we could change the grammar to fix this issue, and not have to solve it in the next phase.

Do you see it?

SIM-1725686 - July 30, 2017, 7:18 pm

This tutorial thusfar has been pretty decent, though as of now, it's basically just a (very) vague description of writing compilers.

My point here, though: Why should the only point of compiler theory be to create malware and wreck other people's computers? Concepts like self-mutating programs can be applied to topics of far greater merit (of course, after more research). To name one - artificial intelligence, which is rapidly becoming one of the most studied areas within computer science...

I guess what I'm getting at is more of a dilemma of philosophy, but... Go do something meaningful. Study computer science at a top university or something. Do research. Win scholarly awards and accolades. Why set the bar at just malware???

movsx - July 30, 2017, 7:25 pm

(Jul 30, 2017 07:18 PM)thosakwe Wrote: [\[CITE\]](#) This tutorial thusfar has been pretty decent, though as of now, it's basically just a (very) vague description of writing compilers.

My point here, though: Why should the only point of compiler theory be to create malware and wreck other people's computers? Concepts like self-mutating programs can be applied to topics of far greater merit (of course, after more research). To name one - artificial intelligence, which is rapidly becoming one of the most studied areas within computer science...

I guess what I'm getting at is more of a dilemma of philosophy, but... Go do something meaningful. Study computer science at a top university or something. Do research. Win scholarly awards and accolades. Why set the bar at just malware???

These tutorials are to motivate people to find ways around metamorphic engines and show that they are a real threat. I don't code malware.

I did a triple major in ECE/math/physics at a top uni and did plenty of research during my undergrad. I'm currently working as an FPGA compiler engineer and I'm considering doing a PhD in computer science or ECE. But not in AI, that field is completely overrated.

c_-m - July 30, 2017, 7:27 pm

(Jul 30, 2017 07:18 PM)thosakwe Wrote: [\[CITE\]](#) This tutorial thusfar has been pretty decent, though as of now, it's basically just a (very) vague description of writing compilers.

My point here, though: Why should the only point of compiler theory be to create malware and wreck other people's computers? Concepts like self-mutating program can be applied to topics of far greater merit (of course, after more research). To name one - artificial intelligence, which is rapidly becoming one of the most studied areas within computer science...

I guess what I'm getting at is more of a dilemma of philosophy, but... Go do something meaningful. Study computer science at a top university or something. Do research. Win scholarly awards and accolades. Why set the bar at just malware???

Not speaking for OP but I assume it is because the nature of this forum will naturally attract people to these sort of subjects, the initial thought of becoming some "powerful haxor" attracts many to the idea(probably some psychological shit behind this?), the thought that staying long enough to realise the risks of true bh activity in most cases isn't for them doesn't cross peoples minds in the initial stages.

SIM-1725686 - July 30, 2017, 7:28 pm

(Jul 30, 2017 07:25 PM)movsx Wrote: [\[CITE\]](#) These tutorials are to motivate people to find ways around metamorphic engines and show that they are a real threat. I don't code malware.

I did a triple major in ECE/math/physics at a top uni and did plenty of research during my undergrad. I'm currently working as an FPGA compiler engineer and I'm considering doing a PhD in computer science. But not in AI, that field is completely overrated. I doubt it will go anywhere, honestly.

Fair enough.

In that case, I personally think it might be better to post a tutorial like this publicly on the Internet, and then reshare it on sites like Reddit, Twitter, and here (if you so wish). You might not really get an educated response from HackForums, ever.

Anyways, just my two cents. Good luck to you.

(Jul 30, 2017 07:27 PM)c_-m Wrote: [\[CITE\]](#) Not speaking for OP but I assume it is because the nature of this forum will naturally attract people to these sort of subjects, the initial thought of becoming some "powerful haxor" attracts many to the idea(probably some psychological shit behind

It wasn't a question of why he posted it here, but about the nature of the tutorial. But I understand what you're saying.

(Jul 30, 2017 07:09 PM)movsx Wrote: [\[CITE\]](#) Sure, I could have done it like that. It's really not high priority for me though.

Do you see it?

I was not too concerned with this kind of stuff because I already did my research on LR(1) parsers a long while back. The code (now that I've actually taken the time to collect all the header and source files) actually has a few more things wrong that I didn't notice just by first glance.

The parsing will still fail under quite a few conditions, but this would be a couple of them:

Some of these are issues in the code.

(Jul 30, 2017 08:24 PM)0xDEAD10CC Wrote: [\[CITE\]](#) I was not too concerned with this kind of stuff because I already did my research on LR(1) parsers a long while back. The code (now that I've actually taken the time to collect all the header and source files) actually has a few more things wrong that I didn't notice just by first glance.

The parsing will still fail under quite a few conditions, but this would be a couple of them:

Code struct obj_t x = { 0 };
Bit advanced but these don't yet work:
Code void *(*f)() = NULL;
More:
x=y works, x9=y doesn't.

Some of these are issues in the code.

These are not actual issues, the language only represents a subset of the C language. I'd use a parser generator to represent the entire C language, with a different CFG.

The issue I was talking about relates to operator associativity.

0xDEAD10CC - July 31, 2017, 6:29 pm

(Jul 31, 2017 05:32 PM)movsx Wrote: [\[CITE\]](#) These are not actual issues, the language only represents a subset of the C language. I'd use a parser generator to represent the entire C language, with a different CFG.

The issue I was talking about relates to operator associativity.

Ohh.. Well this subset would still need to address ' as a char in some way. I don't know what kind of language you're trying to build exactly though so not sure what I'm supposed to be looking for. If you gave me or someone else the documentation or standard for the subset you were building then it might have helped, but I was thinking of this mainly as C89.

movsx - July 31, 2017, 6:31 pm

(Jul 31, 2017 06:29 PM)0xDEAD10CC Wrote: [\[CITE\]](#) Ohh.. Well this subset would still need to address ' as a char in some way.

That is trivial. How would you handle it?

Do you know what associativity is? How would we solve the problem with associativity in our next phase, or by changing our CFG?

0xDEAD10CC - July 31, 2017, 6:40 pm

(Jul 31, 2017 06:31 PM)movsx Wrote: [\[CITE\]](#) That is trivial. How would you handle it?

Do you know what associativity is? How would we solve the problem with associativity in our next phase, or by changing our CFG?

<http://eli.thegreenplace.net/2012/08/02/parsing-expressions-by-precedence-climbing>

There's tons of information out there. If you're going to use the actual character as the identifier then you need an escape sequence, which is why I had assumed '\'. I'm not the one writing the tutorial.

movsx - July 31, 2017, 6:46 pm

(Jul 31, 2017 06:40 PM)0xDEAD10CC Wrote: [\[CITE\]](#)

<http://eli.thegreenplace.net/2012/08/02/parsing-expressions-by-precedence-climbing>

There's tons of information out there. If you're going to use the actual character as the identifier then you need an escape sequence, which is why I had assumed '\'. I'm not the one writing the tutorial.

No, I'm talking about associativity, and how you would solve it. I'm trying to get a discussion going on in these tutorials.

Escape sequence recognition is trivially handled in the lexical analysis phase.

BeNT- - July 31, 2017, 6:52 pm

Pressed for time OP so didn't read the whole thing.

Looks like you managed to consolidate a shit ton of info into reasonable size that makes sense.

Good job.

Cheers. B

0xDEAD10CC - July 31, 2017, 7:07 pm

(Jul 31, 2017 06:46 PM)movsx Wrote: [\[CITE\]](#) No, I'm talking about associativity, and how you would solve it. I'm trying to get a discussion going on in these tutorials.

Escape sequence recognition is trivially handled in the lexical analysis phase. That article was talking about associativity as well. Scroll down to "Handling associativity" it's part of the algorithm.

movsx - July 31, 2017, 7:19 pm

(Jul 31, 2017 07:07 PM)0xDEAD10CC Wrote: [\[CITE\]](#) That article was talking about associativity as well. Scroll down to "Handling associativity" it's part of the algorithm. I already know how to handle associativity.

The grammar is right recursive, and right associative.

If I decide to release tutorial #3, I'll describe how we fix it in the next phase. Alternatively, we can just modify our grammar. The issue with modifying the grammar is each expression derivation will be broken into 2 derivations. It's uglier. But it would save us work in tutorial 3.

ALSO: Just a general poll. Would you guys like me to jump into the backend for my next tutorial or continue with the frontend? I feel like people here would probably want to see some backend action, which would likely be 2 tutorials away from here unless I just jump straight into it.

0xDEAD10CC - July 31, 2017, 10:16 pm

(Jul 31, 2017 07:19 PM)movsx Wrote: [\[CITE\]](#) I already know how to handle associativity.

The grammar is right recursive, and right associative.

If I decide to release tutorial #3, I'll describe how we fix it in the next phase. Alternatively, we can just modify our grammar. The issue with modifying the grammar is each expression derivation will be broken into 2 derivations. It's uglier. But it would save us work in tutorial 3.

ALSO: Just a general poll. Would you guys like me to jump into the backend for my next tutorial or continue with the frontend? I feel like people here would probably want to see some backend action, which would likely be 2 tutorials away from here unless I just jump straight into it.

You explained my point though. The article does talk about associativity, and the algorithm allows for you to handle it in code. I'm more interested in the code aspect.

tulamba - August 1, 2017, 7:18 pm

movsx, great work. Have you looked at Lex/Yacc or Flex/Bison. I believe they are more elegant for tokenization and processing.

movsx - August 1, 2017, 7:56 pm

(Aug 1, 2017 07:18 PM)tulamba Wrote: [\[CITE\]](#) movsx, great work. Have you looked at Lex/Yacc or Flex/Bison. I believe they are more elegant for tokenization and processing.

Yes, but I don't think they'd be suitable for this tutorial because the goal is to have people learn about these concepts rather than use lexer or parser generators.

D3RP - August 7, 2017, 3:27 pm

(Jul 30, 2017 06:58 AM)Hoodie+ Wrote: [\[CITE\]](#) There isn't a simple or easy solution to bypass metamorphic engines.

That's why everything is moving from blacklisting technique to whitelisting technique.

For example Smart Screen or the new Windows 10 versions which allows only to download software from the App Store and your software needs to be verified to be there.

Personally, I'm not a fan of the white-listing approach.

I agree it's kind of odd.. because :

- blacklisting technique is innocent until proven guilty
- whitelisting technique is guilty until proven innocent

Hoodie+ - August 8, 2017, 6:34 am

(Aug 7, 2017 03:27 PM)D3RP Wrote: [\[CITE\]](#) I agree it's kind of odd.. because :

- blacklisting technique is innocent until proven guilty
- whitelisting technique is guilty until proven innocent

Exactly!

It should be innocent until proven guilty ?