GraphComp

jordisp82

April 27, 2023

Contents

1 Introduction

There are several source code analysis tools on the market, some of them proprietary (usually with a really high price tag), some free software, each of them with a varying degree of features and analysis power. There is something they have in common: the problems they find are shown on the source code, the offending piece highlighted and with accompanying explanations about why a problem has been found and maybe some suggestions about how to fix it.

At first glance, there shouldn't be anything wrong with that. What should those tools do, otherwise? After all, developers write source code, and that souce code sometimes has issues, and the developer needs to see the offending source code as it is: source code.

I've been working with such tools for some time, and they are really great, very powerful. They find mistakes in the code, dangerous uses of the C language, violations of MISRA rules and directives, some even point you at possible runtime failures such as accessing an array out of bounds, divisions by zero, and so on.

The creators of these tools are people with a great knowledge of the C language, they understand how a compiler works and how it analyses and translates the source code, and they spend plenty of time making their tools very powerful.

Alas, power is not the only point to consider. If a tool is very powerful but does a poor job telling the developer what and where the problem is, that power of the tool may become next to useless to the developer. We want, first, a tool that is powerful in analysing the source code; second, we want a tool that does a good job explaining the developer what problem the code has, where it is, and maybe even suggesting ways to fix it.

This is where this application comes. In my experience, source code alone is, sometimes, maybe even often, not enough to get a good understanding of the problems found by the analysis tools. Not all developers have the same level of knowledge of the C language, not all of them have been taught — or have had the opportunity lo learn — how a compiler works, and sometimes the tools have diagnostic messages a bit cryptic and hard to understand.

It seemed to me that showing the source code as an abstract syntax tree (AST) could be helpful to the developer to understand what's going on, for several reasons.

1. An AST is a graphical representations, and seeing the code as a tree may make it easier for the developer to understand how the expressions and statemens are evaluated and run, and therefore why there is a problem in something that, at first glance, looks completely harmless.
2. An AST is the way compilers see and translate source code. This lets the developer to be fully aware of the intricacies of the C language and what's going on "under the hood" (such as type conversions, either explicit or implicit), and that too helps the developer to understand the problems pointed out by the analysis tool.

3. A tree representation allows to condense the source code and represent succintly big chunks of code.

This doesn't mean that an AST representation is always better than the "plain" source code. It just means that it shows the code in a different way, and I hope that this can provide extra information and hints to the developer, so he or she can better understand the results of an analysis and find it easier to fix the problems.

To accomplish this goal, the application needs to:

1. Analyze the source code submitted to it, and this means that a compiler front-end has to be developed.
2. Graphically show the source code as an AST, with the results of the analysis visible in the tree.

In the next chapters I'll explain how I designed and developed the application to accomplish its goals, by carefully analyzing the source code submitted to it and representing it as a syntax tree.

2 The compiler front-end

2.1 Introduction

A compiler front-end performs lexical, syntax and semantic analysis, either to perform syntax-directed translation or, much more typically, an AST and an intermediate representation, on which several optimizations can take place before generating the target code (assembler).

When it comes to C, this poses an additional problem: the pre-processor, which is, in practical terms, a language inside the language. I had no doubts about how to deal with this: calling gcc to perform this step. Ah, yes, a parser with an integrated pre-processor would have been awesome, but this is not the core of my application. I didn't want to spend time and effort on this.

Fully knowing what I lose by dealing with "just" a translation unit, then I focused on the lexical analysis. Using flex for that was my first and only option, because implementing a lexer by hand would have given me almost no benefit — meaning, my tool will not be what I intend it to be just because it has an awesome scanner implemented by hand.

In contrast, the parser gave me much trouble. First, I needed a grammar. The C standard luckily has an LALR grammar as one of its appendix, and I found a Bison file with it[1]. I had three big options to choose from:

1. Use the Bison grammar from the Internet and add the code to create the AST, since that file only creates a parser that tells whether a source code file conforms, or does not conform, with the C grammar.
2. Use the Bison grammar from the Internet to create the action and goto tables of the LR parser, and implement the algorithm by hand.
3. Use the Bison grammar from the Internet to get the productions to implement other kind of parsers, more precisely the Earley parser, which is far more tolerant with grammars than the LR parser.

I started implementing the algorithm of an LR parser, using Bison to create the *action* and *goto* tables for the grammar. The reason was that I don't have much experience with Bison (which also means that I don't yet trust it enough), and I preferred to implement the parser myself to have more control over it.

But then I realised that if I want to modify the grammar, at some point in time, to include some language extensions (such as those of the GreenHills compiler) that require doing so, an LALR parser could easily become a pain in the ass — shift-reduce and reduce-reduce conflicts. I remembered having read, in some book, a pletora of other parsing algorithsm that pose almost no restrictions on the grammars, and one of these algorithms was the Earley parser.

Long story short: I ended up implementing the parser with Bison, which was my least preferred option. Why? When I read the Earley parser in the book, all seemed to be wonderful. I even looked at some pages on the Internet to better understand it so I could successfully implement it[2].

[1] https://www.quut.com/c/ANSI-C-grammar-y.html
[2] https://loup-vaillant.fr/tutorials/earley-parsing/

Neither the book nor the webpage were enough for me to succeed. I used a pretty simple and small example, and even with that, the parser failed. Then I tried to use the LR implementation I had previously done, with even a more disastrous result.

It was a very shameful experience, me being incapable of successfuly implementing neither of those parser algorithms. I don't know it the tables were wrong, if my code was buggy, or if I was missing something when implementing the Earley parser. Be it as it may, I realised that it would be better for me to stick with the easy road — after all, a fancy parser isn't the core of my application, either, despite the fact that is closer to it than the lexer.

2.2 The parser

The goal of the parser, here, is just to create the abstract syntax tree (AST). It will take for granted that the code successfully compiles, so it won't do any kind of error recovery. If it finds a syntax error it will stop immediately, no apologies made.

For the AST, I created a node with the following fields:

token :	Integer with the token associated with the node, if any. Relevant only for the leaf nodes.
data :	Generic pointer to additional data, such as the string of a literal, or the value of a numeric literal.
n_children :	Number of children of this node — zero of leaf nodes.
func_ptr :	Pointer to the function that created this node. Used mainly to identify the kind of node, not only regarding its associated non-terminal, but also which production of the non-terminal.
parent :	Pointer to the parent node — null for the root.
children :	Array of pointers to the children nodes.

Creating this tree is not difficult nor complex, although not trivial.

Nodes with no children They are the leaf nodes. Some of such nodes only require to store the kind of token, such as an operator. For identifiers, we store its string in the data field of the node. Others leaf nodes also have a string in this field, even if they do not correspond to an identifier (for example, for a typedef name or an enumeration constant, or goto statements). Numeric literals need to store the value of their literals.

Nodes with a single child For most of them, no additional data is stored for such nodes, but there are some exceptions.

Nodes with more than one child In these cases, the data field of the node is always set to null. There is only one case that, in a node with two children, a strig is stored (second production of struct or union specifiers).

It's up to the functions called within the Bison grammar file to create this AST.

2.2.1 Interaction with the lexer

When the lexer sees something that looks like an identifier, it must first check that it's not a typedef-defined type or an enumeration constant, and there is a function to check precisely that, sym_type.

To properly implement it, I created a tiny symbol table. For each scope I have two list of strings, one for the types and another one for the enumeration constants. Also, each scope has a pointer to its parent. There is at list one scope, corresponding to the entire translation unit.

Function sym_type begins checking the current scope, looking for the string in the two lists. If it's found in one of them, it's clearly not an identifier, and it will return what it is depending on the list where the string was found. If it's not found in any of the list, then it does the same check for the parent scope, if any, until there is no parent scope to look for. In this case, we're sure it's an identifier, and sym_type returns accordingly.

But who takes care of creating those scopes and filling in the lists of strings?

- Except for goto labels, in C there are two scopes: file scope, and block scope. For the first one, the two lists are already created; when the parser sees the start of a block ("compount statement") creates a new block with function push_scope, and when it sees the end of the block, it destroys it (with function pull_scope). In both cases, the pointer to the current scope (the one that has to be looked for first) is updated.
- Enumeration constants are seen when the enumeration is created. When the parser sees non-terminal "enumeration constant", it adds its string to the list of constants of the current scope[3].

  The case of typedef-defined types is more complicated, since the syntax for declarations is more complex. There is a production for declarations that includes non-terminal for storage-class specifiers (and typedef is one of them). So, in this production, the parser checks if typedef is among those specifiers[4]. If it is, then the identifier of the declarator is stored as a typedef-defined type for the current scope[5].

2.2.2 Flattening the AST

With all of this made, I then proceeded to "colorize" the AST, that is, to fill it with some attributes. Let's suppose we have a variable declaration: it would be very convenient to have, at the root node of that declaration, all we need to know about this variable: name, type, storage class, and so on. Sooner said than done.

With the AST we have just created, we face a problem. The tree mimics the grammar, which is tailored for an LALR parser. This kind of parsers loves left recursion in the grammar. Let's put an example.

Imagine a translation unit, which is the start symbol of the grammar. A translation unit is just a set of external definitions (which in turn can be either declarations or function definitions). Now, when it comes to the grammar, a translation unit is either an external definition, or a translation unit followed by an external definition.

If our translation unit has, let's say, a thousand external declarations, we won't have a root node with a thousand children. We will have a node with two

[3] Function scope_add_enumeration_constant
[4] Function look_for_typedef
[5] Functions register_id_as_typedef and scope_add_typedef

children, and the first child will, in turn, have two children, until we will reach, almost one thousand generations later, a node with a single child.

Not only this makes giving attributes to the nodes more difficult, but it will also drive mad the developer who sees his or her source code in this way — inducing the developer to believe that there is some hierarchical relationship between the declarations and function definitions in the source code.

The AST created by the parser, therefore, needs to be “flattened”. If there are a hundred external defintions in a translation unit, then the translation unit has to be one node with a hundred children, period — no matter what the LALR grammar says.