Abstract Syntax Tree (AST) for C Programming Language

Overview of compilation

Broadly speaking, compilation of C programs refers to producing executable binary files containing machine code from source files. It consists of the following procedures:

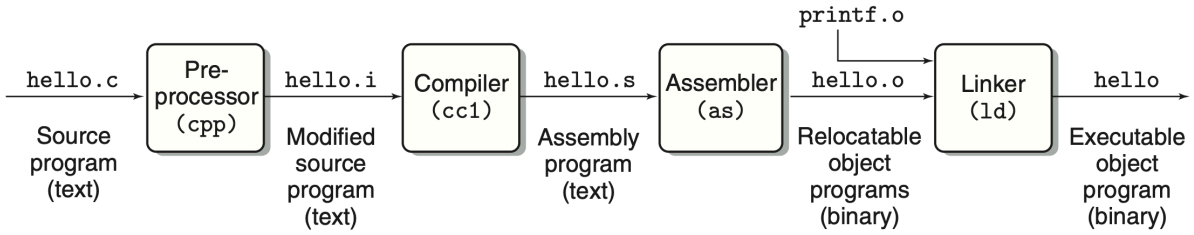
1. Macros in a C file are expanded, and a modified C file (referenced as an expanded C file later) is generated. This stage is also known as preprocessing. To be more specific, a preprocessor handles a C source file by recursively searching header files, expanding macros in found header files, and replacing macros in the source file with expansion results.

2. A C compiler translate a expanded C file into an assembly file which contains human-readable machine code and assembly directives. Compilation in the narrow sense refers to this procedure.

3. An assembler produce an object file from an assembly file. An object file includes binary machine code but can not executed directly at the this stage.

4. An static linker collects multiple object files and creates an executable binary file.

The above processes can be well illustrated by the following image (adapted from *Computer Systems: A Programmer's Perspective 3nd Edition*).



Handling C Macros

Different from programming languages that do not support macros (such as Java and Python) or have macros as part of their language syntax (for example Rust), C has loose specifications about how macros can be defined and used. Thus, a C source file may seem to have syntactic errors even ignoring macro definitions. For example, the following code snippet was extracted from stdlib/strtol.c:

```
INT
INTERNAL (strtol) (const STRING_TYPE *nptr, STRING_TYPE **endptr,
           int base, int group)
{
  return INTERNAL (__strtol_l) (nptr, endptr, base, group, false,
                _NL_CURRENT_LOCALE);
}
libc_hidden_def (INTERNAL (strtol))
```

According to the section *6.9 External definitions* of the C11 specification, the first part of above the code is fundamentally a function definition if `INTERNAL (strtol)` is considered as a declarator. By contrast, `libc_hidden_def (INTERNAL (strtol))` is seemingly a syntactic error because the C language specification requires declarations, which are top-level components of a translation unit (essentially a C source file), to end with `;`. In fact, both `libc_hidden_def` and `INTERNAL` are macros, and the whole line will be expanded to a valid c declaration as:

```
extern __typeof (__strtol_internal) __EI___strtol_internal __asm__(__hidden_as
```

To construct ASTs, there are two approaches to issues related to use of macros. One is to first preprocess a C file and then to parse it. For example, we can use the proprocessors of Clang and GCC to expand a C file to a "clean" one. Later, we could utilze third-party libraries, such as Python binding of libclang or Rust crate lang_c, to build ASTs from it. Nevertheless, it may be significantly inconvenient to require a file to be preprocessed for two reasons. First, for a realistic C project, preprocessing typically requires both system and user-defined header files. In order to obtain accurate preprocessing results, users have to specify search paths for header files either manually or by providing a compilation database which records how a C compiler compiles every file. For the latter manner, a database is generated only after a successful compilation of the whole project. Moreover, preprocessing can only be done on a system that has header files of relied third-party libraries. Therefore, it may fail to build ASTs in a clean system because the required header files are missing. In summary, it is somehow heavy-weight to do both preprocessing and AST construction even though C has a relatively small set of language syntax.

The second approach is to model common macro uses as an extended C syntax without requiring preprocessing. For example, we could parse `libc_hidden_def (INTERNAL (strtol))` as a declaration (or a less common function definition) since a top-level element of a C file is either a function definition or a declaration. Obviously, this approach has several downsides. As C macros can be defined and used in an extremelly flexible manner, the set of extended C syntax could be infinitely large. It is impossible to anticipate all the ways C developers use macros and construct an AST for each use. Therefore, it may fail to parse a C file to ASTs if the file is not yet preprocessed and an unexpected macro use is encountered. Nonetheless, this approach enables parsing a C file independently, without relying on other header files or preprocessing.

To conclude, preprocessing-free ASTs construction provides a good starting point for static analysis of realistic software projects. As a superset of standard C syntax, the extended C syntax should work with C files that have been preprocessed. Thus, preprocessing can be integrated in the workflow of static code analysis to obtain semantically legitimate C structures, which could be further transformed into low-level representations for data-flow and control-flow analysis.

Implementation based on tree-sitter

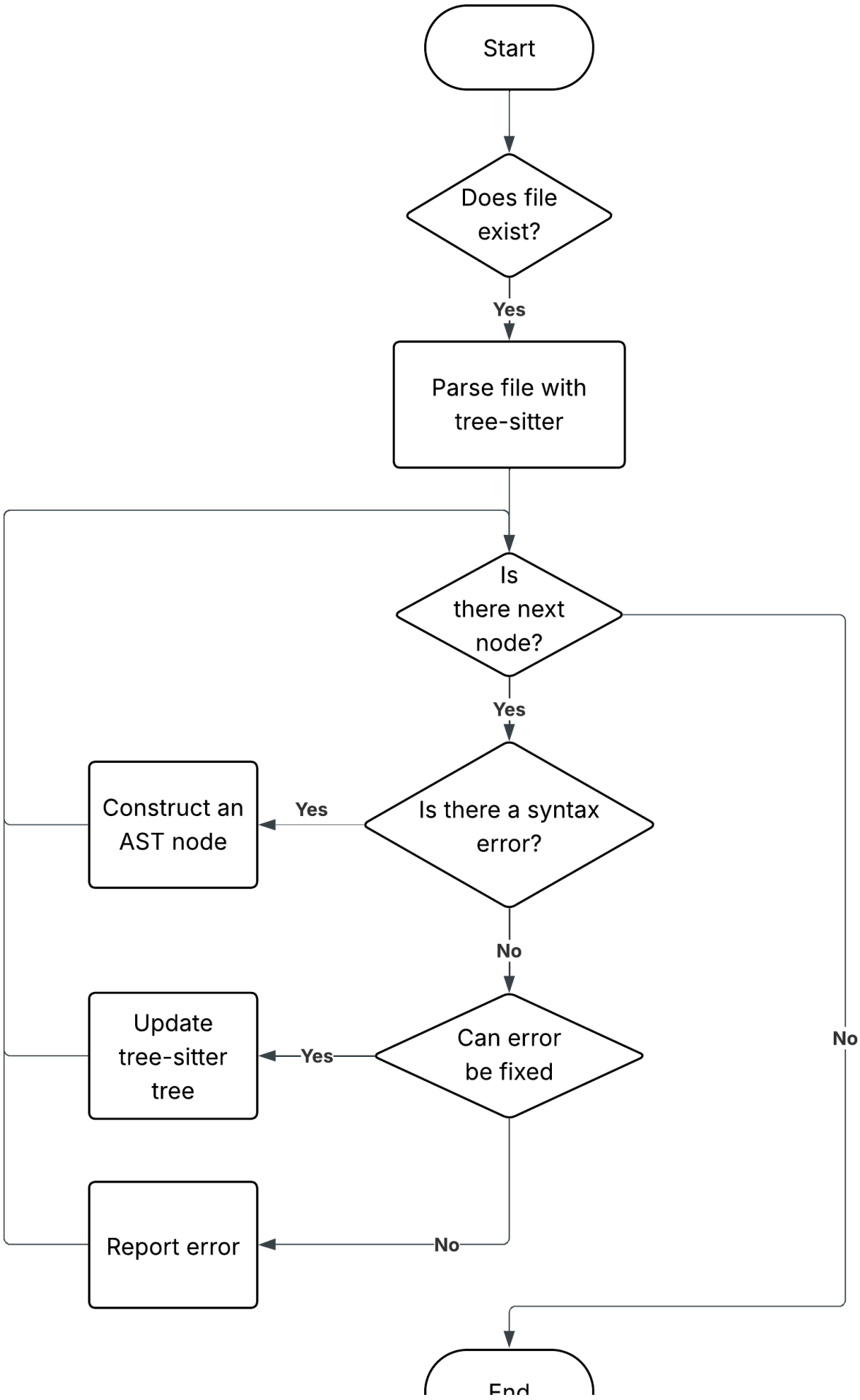
Tree-sitter is an incremental parsing system for programming tools. It offers Python bindings for many supported languages and supports C programming langauge. Hence, we utilize tree-sitter and tree-sitter-c to construct abstract syntax tree for C. Instead of directly using tree presentations provided by tree-sitter, we create our own AST for two reasons. First, tree-sitter syntax trees are essentially concreate syntax trees (CSTs, also known as parse trees), which keep comments, punctuator tokens, new lines, and etc. While it could be ideal for online code editing to keep these syntax component, it may be annoyingly demanding to analyze CSTs. Secondly, while manipulated programmingly, every tree node of a tree-sitter tree is of the same type and too vaguely plain to do analysis on as tree-sitter has support for a very large base of programming langauges. Therefore, a more specified and concise tree representation is necessary. Our design of abstract syntax tree for C is primarily inspired by tree-sitter-c and adopt some syntax structure of lang_c to better reflect C standard syntax.

As mentioned previously, although tree-sitter-c already handle several macro uses via extended syntax, it is still possible to see syntax errors reported by tree-sitter as unknown irregular macro uses could potentially exist. We provide an extendable mechanism for handling unrecognized syntax via a plugin system. If a syntax error is detected, registered plugins are searched for a code fixing solution. Some key interfaces for the plugin system are as follows:

```python
@dataclass(frozen=True)
class CodeFix:
    byte_start: int  # inclusive
    byte_end: int  # exclusive
    replacement: bytes


class ParsingFixerInterface(metaclass=ABCMeta):
    @abstractmethod
    def node_type(self) -> str:
        pass

    @abstractmethod
    def can_fix(self, tree: Tree, curosr: TreeCursor) -> bool:
        pass

    @abstractmethod
    def fix(self, tree: Tree, cursor: TreeCursor) -> CodeFix:
        pass


class ParsingFixerRegistry:
    _registry: Dict[str, List[ParsingFixerInterface]] = defaultdict(list)
```

```
    @classmethod
    def register(cls, fixer: ParsingFixerInterface) -> bool:
        node_type = fixer.node_type()
        fixer_list = cls._registry[node_type]

        if fixer in fixer_list:
            return False
        else:
            fixer_list.append(fixer)
            return True

    @classmethod
    def lookup_fixer(
        cls,
        tree: Tree,
        cursor: TreeCursor,
    ) -> Optional[ParsingFixerInterface]:
        if cursor.node is None:
            return None

        node_type = cursor.node.type
        for fixer in cls._registry[node_type]:
            if fixer.can_fix(tree, cursor):
                return fixer

        return None
```

Following is a flowchart representing the AST parsing process:


Start
Does file exist?
Yes
Parse file with tree-sitter
Is there next node?
Yes
No
Is there a syntax error?
Yes
Construct an AST node
No
Can error be fixed
Yes
Update tree-sitter tree
No
Report error
End

End

Known limitations

Currently, parsing will fail for the following C language features:

1. Inline assembly statements.
2. Type aliasing using `typedef` keyword.
3. Type qualifiers exclusive to MSVC, including but not limited to `__cdecl`, `__clrcall`, `__stdcall`, `__fastcall`, `__thiscall`, `__vectorcall`, `__declspec`, `__based`.
4. C11 generic selection expressions.
5. C11 static assertion.