Implementation of Fork-Merge Parsing in OpenRefactory/C

by

Kavyashree Krishnappa

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 10, 2015

Keywords: Refactoring, Parsing, Fork-Merge

Approved by

Abstract

C preprocessor directives are extensively used in C programs. Due to this, performing advanced program transformations on C programs is difficult. To obtain correct transformations, all possible configurations have to be considered. Modern IDEs ignore multiple configurations of the C preprocessor due to its complexity. OpenRefactory/C is an infrastructure that builds correct and complex transformations for C programs.The preprocessor and the program representation were modified to support multiple configurations in OpenRefactory/C. The C grammar was extended to include conditional directives in the program representation. The extended grammar allows conditional directives to appear in between certain complete C constructs. However, the conditional directives can appear in any part of the program. We modified the OpenRefactory/C parser to handle conditional directives that appear at an unexpected location in C programs and hence support multiple configurations.

Acknowledgments

I take pleasure in thanking many people for making my thesis possible. Firstly, I would like to thank Dr. Jeffrey Overbey and Dr. Munawar Hafiz for their guidance and support throughout this work. I owe much gratitude to Dr. James Cross for serving as member of my advisory committee.

I thank Farnaz Behrang for her help throughout my thesis work. I would like to thank my sister Hemamamatha Krishnappa for her constant support and encouragement. I express my deepest gratitude to my family and my friends for their love, compassion and support in my endeavor. Without them, I know that none of this would have ever been possible. Finally, I am truly grateful to GOD for giving me good opportunities.

Table of Contents

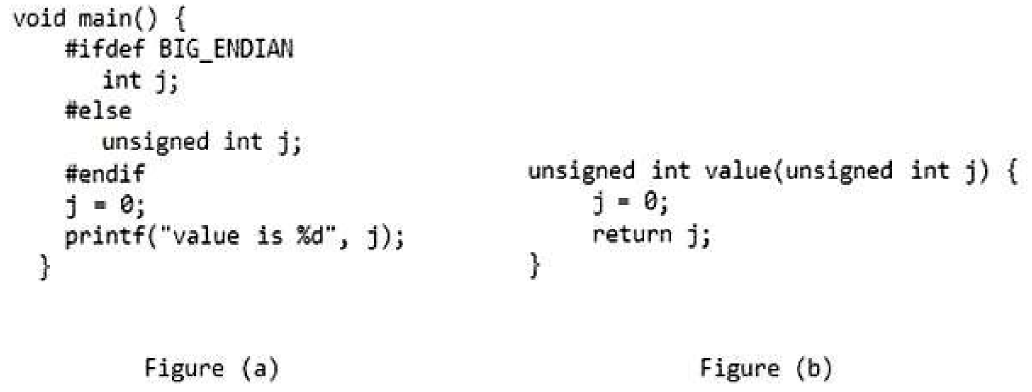
List of Figures

Chapter 1

Introduction

Refactoring a source code improves its structure and design without changing the external behavior. It results in more readable and maintainable code and hence modifying the refactored code is easier. Many modern IDEs such as Eclipse support refactoring. There are many refactoring tools for C which support rudimentary transformations. However, these tools do not support advanced refactorings such as extract method refactoring. Due to the extensive use of preprocessor directives advanced refactoring of C is difficult.

The C grammar does not include preprocessor directives and hence the C code should be preprocessed before parsing. Refactoring a preprocessed version of code might result in inaccurate transformation. Suppose in the example shown in figure 1.1 (a), BIG_ENDIAN is not defined, refactoring the preprocessed version of source code results in inaccurate transformation as shown in figure 1.1 (b). Correct transformation of the code can be performed by supporting multiple preprocessor configurations. In the example, if both branches of #ifdef are considered, refactoring will accurately transform the code as shown in figure 1.2.



```
void main() {
   #ifdef BIG_ENDIAN
      int j;
   #else
      unsigned int j;
   #endif
   j = 0;
   printf("value is %d", j);
 }
```

```
unsigned int value(unsigned int j) {
     j = 0;
     return j;
}
```

Figure 1.1: Example illustrating the importance of multiple configuration transformations.

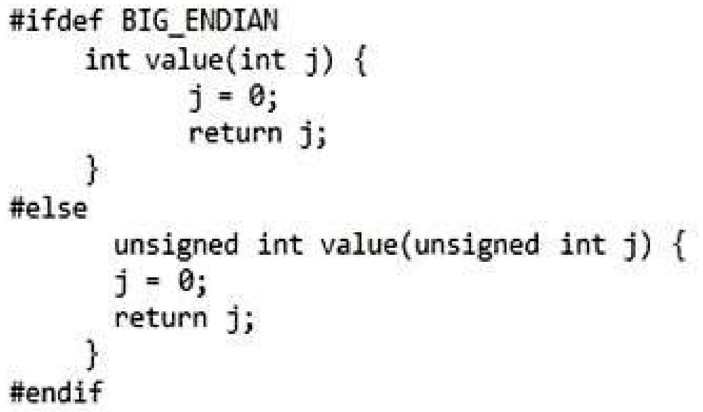

```
#ifdef BIG_ENDIAN
     int value(int j) {
            j = 0;
            return j;
     }
#else
       unsigned int value(unsigned int j) {
       j = 0;
       return j;
     }
#endif
```

Figure 1.2: Result of refactoring with multiple configurations.

1.1 OpenRefactory/C

OpenRefactory/C is a framework that builds correct and intricate program transformations for C. Every part of the OpenRefactory/C infrastructure is modified to support configuration aware analysis and transformation.

1.2 Modifications to preprocessor and extending C grammar

As explained in the previous section, the preprocessed code can not be transformed as it will result in losing branches of conditional directives except one. To overcome this problem, we use the pseudo-preprocessing model by Garrido [3]. In pseudo-preprocess model, the abstract syntax tree generated is similar to code after preprocessing and contains all possible configurations of the code.

Behrang [22] [23] made changes to the preprocessor and program representation to support multiple configurations. She designed an algorithm to keep track of all guarding conditions associated with each macro definition. Therefore, when a macro call is expanded, every possible expansion is included with a guarding condition.

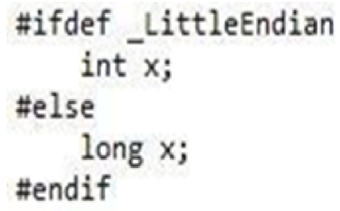
The modified preprocessor feeds the lexical analyzer and the lexical analyzer produces the stream of tokens that includes static conditional directives. However, the C grammar does not include the static conditionals constructs. To parse the C code that contains conditional directives, Behrang [22] [23] extended the C grammar using the specifications given in ISO standard for C99 [24].

1.3 A Problem

The extended C grammar allows preprocessor directives in between complete C constructs such as statements, function definitions and declarations. Despite that, the preprocessor directives may appear anywhere in the code breaking a complete C construct and violating the syntax of extended grammar. The LALR(1) parser used in OpenRefactory/C is not capable of handling preprocessor directives that appear in inside a complete C construct.

1.4 Our solution

The parser generated using extended C grammar can successfully parse the code in figure 1.3, since it has complete C constructs inside each static conditional branch. The resulting abstract syntax tree for code in figure 1.3 is as shown in figure 1.4. The parser fails



```
#ifdef _LittleEndian
    int x;
#else
    long x;
#endif
```

Figure 1.3: Preprocessor directives appearing in between complete C constructs .

to parse the code in figure 1.5, since it violates the rules of extended C grammar. We need to modify the OpenRefactory/C parser to handle such cases. We modified the LALR(1) parser in OpenRefactory/C to handle the conditional directives that appear inside the complete C constructs by implementing Fork-Merge technique. The Fork-Merge algorithm was introduced by Gazzillo and Grimm [8]. In Fork-Merge technique, a new subparser is forked for

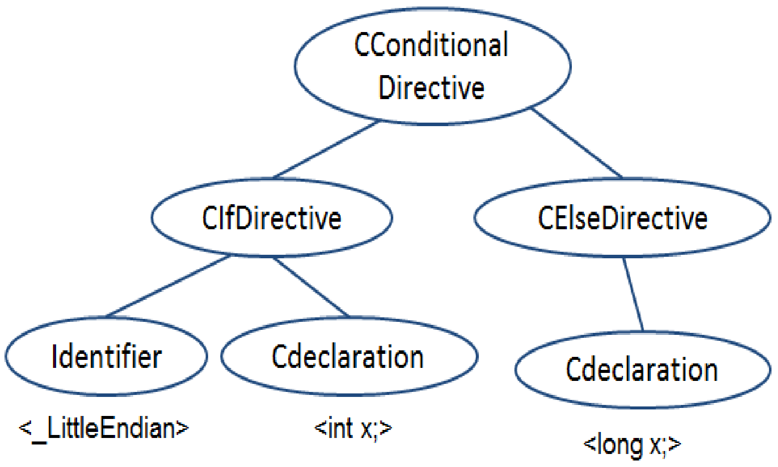
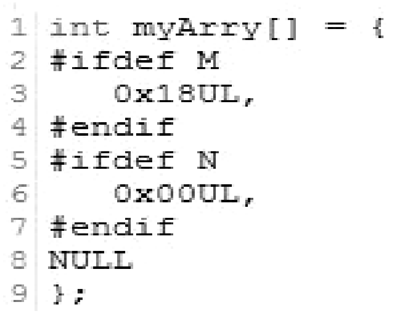


Figure 1.4: Abstract syntax tree constructed by parsing code in figure 1.3.



```
int myArry[] = {
#ifdef M
   0x18UL,
#endif
#ifdef N
   0x00UL,
#endif
NULL
};
```

Figure 1.5: Preprocessor directives in the middle of complete C constructs.

every static conditional. All subparsers together recognizes all configurations, hence supports multiple configurations.

1.5 Thesis Organization

- Chapter 2 introduces basic concepts such as abstract syntax trees, parsing, different classes of parsers, refactoring and automated refactoring tools. It also includes review on existing literature.

- Since our work is based on fork-merge parsing algorithm, in Chapter 3 we explain how fork-merge parsing algorithm works, optimizations introduced to fork-merge algorithm to improve its performance and finally we explain the same using an example.

- Chapter 4 describes how we implement fork-merge algorithm in OpenRefactory/C. The changes made to fork-merge algorithm to accommodate requirements of OpenRefactory/C. This chapter gives the implementation details of the parser.
- Chapter 5 discusses the conclusion and future work.

Chapter 2

Literature Review

2.1 Abstract Syntax Trees

A programming language specification should include the syntax and the semantics of the language. A parser for a programming language verifies that its input obeys the syntactic rules of the language specification. Compilers often use data structures called Abstract Syntax Trees.

An abstract syntax tree represents the syntactic structure of the program. Each node of the abstract syntax tree represents a construct within the program. The word "abstract" is used since an abstract syntax tree does not include every detail appearing in the real code. A simple if-then-else statement is represented using a single node with three branches. The below figure represents the abstract syntax tree (AST) of a simple if-then-else statement:

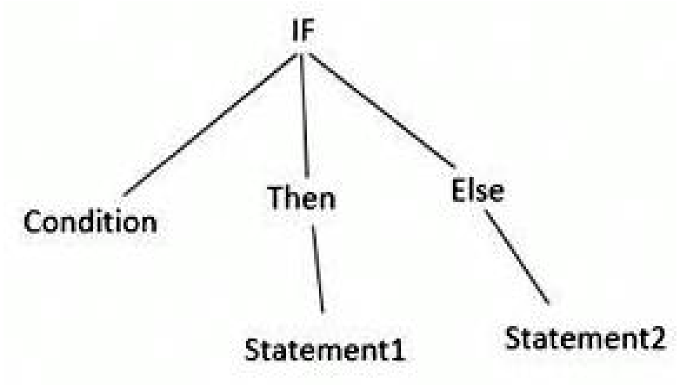


Figure 2.1: Abstract syntax tree of a simple if-then-else statement.

Abstract syntax trees often mimic the grammatical structure of a language, but do not include punctuation and delimiters such as semi-colon, parenthesis, etc. Note that, as we progress from the root to the leaves of the AST, the nodes represent progressively fine-grained language constructs. The process of building a syntax tree (logically) for a given input using the syntactic rules of a programming language is called parsing.

2.2 Parsers

One of the simplest parsers is called a shift-reduce parser. A shift-reduce parser is a table-driven nondeterministic stack machine [18]. A shift-reduce parser is categorized as a bottom-up parser since it identifies and processes the input's lowest-level details first, then the mid-level structures, and finally the highest-level overall structure. The shift-reduce parser attempts to construct a parse tree by performing a series of shift and reduce actions. The action of pushing an input symbol onto its stack is called shifting, and popping the input symbols from the stack and replacing the symbols with the left-hand side of the production is called reducing.

A grammar in formal language theory is used to specify the rules of a programming language, and it can be defined using a set of production rules. Most languages are defined using context-free grammars. The production rules in a context-free grammar are represented in the following format:

Where a single nonterminal appears on the left-hand side and a string of terminals or/and nonterminals appears on the right-hand side of each production rule. The right-hand side of the production rule can be empty, and such productions are called -productions.

In 1965, Donald E. Knuth [17] invented LR parsing, a variation of shift-reduce parsing. LR parsers are capable of recognizing languages defined by deterministic context-free grammars [9]. LR is an acronym, where L means the parser reads the input from left to right and R means the parser produces the right-most derivation. An LR parser is a type of shift-reduce parser that adds an additional finite state machine and look ahead symbols.

In an LR parser, there are four types of actions that can take place: shift, reduce, accept, or error. The shift and reduce actions are the same as any other shift-reduce parser. An accept action takes place after completely reading the input string, and the parsing is successfully completed. An error arises if the input string does not obey the rules of the grammar.

An LR(0) parser is the simplest of all LR parsers. It does not use lookahead tokens. A lookahead token is the incoming token which a parser uses to make shift/reduce decisions. In LR(0) parser, the shift/reduce decisions are made based on the tokens that are already read. Figure 2.2 shows an example for LR(0) grammar.

$$\begin{aligned} S &\longrightarrow S + E \mid E \\ E &\longrightarrow \text{num} \end{aligned}$$

Figure 2.2: Example for LR(0) grammar.

The LR(0) parsing table is constructed using LR(0) items which are the grammar rules with an additional period symbol on the right-hand side of the rule. The possible LR(0) items for the grammar in Figure 2.2 is shown in figure 2.3.

$$\begin{aligned} S &\longrightarrow S + E \\ S &\longrightarrow \bullet S + E \\ S &\longrightarrow S \bullet + E \\ S &\longrightarrow S + \bullet E \\ S &\longrightarrow S + E \bullet \end{aligned}$$

Figure 2.3: LR(0) items for LR(0) grammar in Figure 2.2.

The symbols on the left-hand side of the period symbol correspond to the input string that is recognized by the parser. The symbols on right-hand side of the period symbol correspond to possible next input.

With an LR(0) grammar there are no shift/reduce conflicts or reduce/reduce conflicts. A shift/reduce conflict occurs when the grammar allows a rule to be reduced for a particular token and allows another rule to be shifted for the same token. A reduce/reduce conflict occurs when the grammar allows two different rules to be reduced for a particular token. The grammar in figure 2.4 has shift/reduce conflict and is not an LR(0) grammar and the figure 2.5 shows an example for non-LR(0) grammar with reduce/reduce conflict.

$$S \longrightarrow E+S \mid E$$
$$E \longrightarrow \text{num}$$

Figure 2.4: An example for non-LR(0) grammar with shift/reduce conflict.

$$S \longrightarrow A2 \mid B3$$
$$A \longrightarrow 1$$
$$B \longrightarrow 1$$

Figure 2.5: An example for non-LR(0) grammar with reduce/reduce conflict.

With a lookahead in LR parsers, both shift/reduce and reduce/reduce conflicts in the above examples can be solved. A lookahead is a terminal representing the next possible input to the parser after the right-hand side of the grammar rule. An LR parser with a single lookahead is called an LR(1) or canonical parser. These have the potential to recognize all deterministic context-free languages. LR(1) items are used to construct the canonical parsing table. An LR(1) item is a LR(0) item annotated with a lookahead. For LR(1) grammar in figure 2.6(a) the LR(1) items are as shown in figure 2.6(b). The lookahead and the symbols on the right-hand side of the grammar is separated by a comma.

$$S \longrightarrow E+S \mid E$$
$$E \longrightarrow \text{num} \qquad\qquad E \longrightarrow \bullet\text{num}\,,+$$

(a) LR(1) grammar productions (b) LR(1) items

Figure 2.6: An example LR(1) grammar and its LR(1) items.

Canonical parsers have a high memory requirement. To overcome this limitation of LR parsers, Frank DeRemer in 1969 [16] devised two variations on LR parsers: SLR parsers and LALR parsers.

Unlike LR parsers, the SLR parser does not use the lookahead to construct the parser table. The SLR parser considers the follow-set of a symbol to decide whether to shift or reduce. The follow-set of a symbol is the set of all terminal symbols that can follow the occurrence of the symbol. SLR and LALR parsers have the same size tables and same parser states.

LALR parsers accept more grammars than SLR parsers. LALR parsers lie in between SLR and canonical parsers in terms of the grammars accepted. In a canonical parser, multiple copies of any particular state in an LR(0) automaton may exist, each annotated with a different lookahead information. LALR parsers can be implemented by starting with a Canonical parser and integrating all the states that have the same LR(0) items but different lookaheads. The lookaheads can be aggregated together, resulting in a parser with the same number of states as LR(0) and the same information as the Canonical parser. LALR parsers are capable of handling more shift/reduce conflicts than SLR parsers [18]. Figure 2.7 shows the hierarchy of grammar classes for bottom-up parsing.

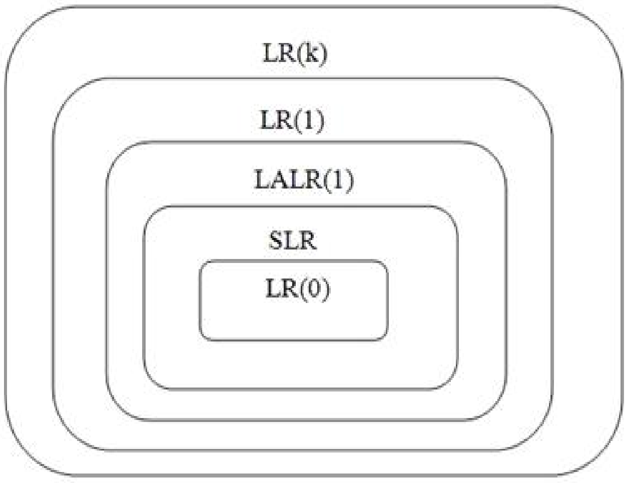


Figure 2.7: Hierarchy of grammar classes for bottom-up parsing. [25]

2.3 Challenges in C parsing

The C language consists of the C language proper as well as the C preprocessor. Including the preprocessor makes the C language more expressive, it has facilities such as file inclusion macros and static conditionals.

In practice, the use of the preprocessor in the C language is extensive, and preprocessor code generally does not conform to the syntactic rules of the base C language. The C processor operates at the lexical level, and this makes the parsing C language more challenging. Preprocessor conditionals are frequently used in many software systems. These conditionals are usually used to declare and define the configuration variables. Preprocessing and parsing these conditional directives have been a significant challenge.

2.4 Analyzing C preprocessor usage

Ernst et al. [4] analyzed C preprocessor usage to determine how frequently the preprocessor is used in practice; this provides significant insights on practical usage of the C preprocessor. The primary motive of their study was to evaluate the potential for preprocessor usage reduction and to find out the difficulty in creating a framework for preprocessor aware tools. They analyzed 26 packages that consist of 1.4 million lines of C code. Through their analysis, Ernst et al. found that about 8.4% of program lines are preprocessor directives and across packages it varies from 4.5% to 22%. The lines that expand a macro or the lines whose inclusion is controlled by if statements was not included while computing these ratios. They also found that about 48% of the total directives in all packages they analyzed were conditional compilation directives. The remaining 32% of the total directives were macro definitions, and file includes making up to 15% of total directives.

Ernst et al. conducted a detailed analysis of macro definitions that may complicate the program understanding. A macro can have multiple definitions inside a package, or it can be even redefined while preprocessing. Redefining a macro makes it harder to find which definition is used in a particular expansion. Ernst et al. examined the frequency of macro redefinition in the 26 packages they analyzed. They found that about 10% of the macros were defined twice, 2% of the macros were defined thrice, and 2% of the macros were defined four or more times. They also reported data regarding the prevalence of preprocessor directives, macro body categorizations, use of the C preprocessor to achieve, inconsistencies and errors in macro definitions and uses, and dependencies of the code upon macros.

2.5 Preprocessor conditionals

Preprocessor conditionals are often used to make the software compatible with various hardware platforms and operating systems. Preprocessor macros are used as configuration variables, and then tested using preprocessor conditionals to configure the software accordingly. Aversano et al. [6] in their work handling preprocessor conditioned declarations propose an approach for finding out all possible types a configuration variable can be declared in a software system. They also analyze the effect of using preprocessor conditionals for declaring configuration variables in 17 different software systems.

The authors proposed a tool that finds all possible types of a configuration variable declared, and that detects inconsistent variable declarations for all possible configurations. The tool that they created parses and creates an abstract syntax tree and then builds an enhanced symbol table. The enhanced symbol table consists of the variable name, scope, type of the variable and the conditional expression which decides the type of the variable. The software system can be compiled and type-checked for all possible configurations using the enhanced symbol table. This is unlike the traditional compilation where only one possible configuration is type-checked at a time. After type checking, a list of all possible type errors that can be produced by the software system is created.

There are many software systems that have a lifetime of more than ten years, for such software systems, a new configuration is required, and old ones may not be used anymore due to the evolution in hardware and software platforms. These old configurations which are no longer needed are considered to be dead configurations. Removing such dead configurations would save programmers from wasting their time to understand and analyze these configurations. Removing these old configurations is tedious tasks as it takes a lot of time to check for these old configurations, edit, compile, debug and test millions of lines of source code.

Baxter and Mehlich [1] introduced a tool to remove the configurations that do not have utility anymore. They use the Design Maintenance System (DMS) and a set of rules that specify the configurations to be removed. Using DMS and these set of rules, the dead configurations are removed automatically within a few hours and without much effort. They also proposed a method to simplify and easily evaluate the Boolean expressions.

Kastner et al.[7] proposed a variability aware parser which parses almost all unpreprocessed code within a reasonable time. The variability aware parser not only finds the syntax errors in the source code, it also finds the possible type errors that can occur. The variability aware parser would build a single abstract syntax tree which reflects the code's variability.

The code's variability is represented by using optional subtrees. In their work, Kastner et al.[7] proposed a tool called TypeChef that has a variability aware lexer and a variability aware parser. The variability aware lexer propagates variability from conditional compilation to conditions in the token stream and resolves macros and file inclusion.

The variability aware parser reads the token stream from the lexer and produces an abstract syntax tree which preserves variability. Similar to the enhanced symbol table that was used in Aversano et al.'s work on handling preprocessor conditioned declarations, a conditional symbol table was used during parsing to track the state of the program correctly.

Although TypeChef seems to solve the problems with the preprocessor conditionals, TypeChef misses several interactions with preprocessor and it has few drawbacks. TypeChef evaluates the constraints only if the macros are defined using #define keyword.

2.6 Fork-Merge parsing

Gazillo and Grimm [8] proposed an open source tool called SuperC that completely parses programs with high variability. They propose a configuration preserving preprocessor that resolves the preprocessor directives such as includes and macros. This leaves out static conditionals sustaining the variability of the program. They also propose a configuration preserving parser which builds an abstract syntax tree with a static node for each static conditionals. The configuration preserving parser forks a new subparser each time it encounters a static conditional. The authors propose algorithms for configuration-preserving preprocessing and configuration-preserving parsing.

To limit the number of subparsers that are forked for a static conditional, they implement techniques such as early reduces, lazy shifts, and shared reduces. Early reduces make sure that the subparsers merge as soon as possible. Lazy shifts decrease the number of subparsers by delaying the forking of subparsers that will shift, and shared reduces eliminate the duplicate work done by subparsers by reducing single stack for several heads at the same time. Chapter 3 describes the complete functionality of a Fork-Merge parser.

2.7 Refactoring

Refactoring is the process of removing the unwanted or duplicate code in the existing code and improving the design without affecting the external behavior of the code [12]. Opdyke first introduced the term refactoring in 1990. Refactoring results in a more readable, efficient and maintainable code. By refactoring the code, fixing the bugs in the code becomes easier. Refactoring code makes it readable and understandable, it removes any existing duplicate code or logic and simplifies programs that have complex conditional logic. It is easier to find and fix bugs in programs that are readable, easy to understand, having less lines of code and simple logic. Refactoring increases the code reusability. If we need to build something on the existing code, we have to understand the code and by refactoring, it is easy to read, understand and modify or build on the existing code. Refactoring can be either done manually by the software developer or by using an automated tool. To perform the refactoring manually, one has to go through the code manually and make changes, this may lead to errors, and it is a frustrating and tedious process if refactoring has to be performed on enormous code. William Opdyke in his Ph.D. thesis [19], came up with the idea of refactoring tool that can implement the chosen refactoring by itself. Don Roberts, John Brant, and Ralph Johnson introduced a refactoring tool for smalltalk [10] which is the first refactoring tool ever.

2.8 Automated refactoring tool

There are many Interactive Development Environments (IDEs) and software editors that support automated refactoring. Eclipse, NetBeans, Xcode, Visual Studio are some of the IDEs which support automated refactoring. Automated refactoring involves following steps:

1. Discovering the parts of the code to be refactored.
2. Finding which refactoring to be applied to the code.
3. Make sure that the refactoring to be applied preserves the behavior of the code.
4. Transforming the existing code that is selected by applying the refactoring.
5. Evaluate the effect of refactoring on quality of code.
6. Make changes in the documentation of the code according to the changes made in the code due to refactoring.

The automation of refactoring tool can be partial or full [15]. In partially automated refactoring, the developer recognizes the part of the code to be refactored and selects the appropriate refactoring to be applied, and the selected refactoring is performed automatically. Most of the modern IDEs that support refactoring use partial automation approach. There are some researches that support fully automated refactoring. In a fully automated refactoring, the entire process is automated. Guru, a refactoring tool for SELF programs, is an example of full automated refactoring tool [20]. Automated refactoring can be used to convert object-oriented programs into aspect-oriented programs by marking the aspects in the existing code, generating the corresponding aspect code and removal of the marked code through refactoring [13]. Automating application of design patterns to the existing code is one of the vital use of refactoring [14]. Application of design patterns to the code makes the software system more flexible and with more flexibility the system can be extended according to the changing requirements.

2.9 CRefactory

Refactoring a C source code with conditional directives is challenging [11]. Refactoring is usually performed after preprocessing that means that the refactoring is performed on the preprocessed version of the code, and the results of refactoring is based on a single configuration. But, a refactoring tool cannot consider a single configuration of the source code because changing source code for one configuration might not support compiling the source code for other configurations. Therefore, it is important that the refactoring tool makes sure that it preserves all possible configurations.

Garrido and Johnson [3] [5] proposed a refactoring tool called CRefactory, which handles conditional directives correctly. In their work, they successfully proposed a way of integrating conditional directives in to the C grammar and program representation. They preferred to complete the branches of conditional directives with the text that is before the conditional, after the conditional statement, or both. By doing this, the conditional statements can be added to the C grammar and hence parse all conditional branches in a single pass.

Chapter 3

Overview Of Fork-Merge Parsing

Gazzillo and Grimm [8] proposed a configuration preserving parser which builds an abstract syntax tree with a static choice node for each static conditional. The configuration preserving parser forks a new subparser each time it encounters a static conditional. To improve the performance of the parser, they introduced four optimizations: token-follow set, early reduces, lazy-shifts and shared reduces. In the next section, we explain how Fork-Merge Parsing algorithm works, following which we describe the optimizations that were introduced to improve the performance of the parser and finally we illustrate Fork-Merge Parsing using an example.

3.1 Fork-Merge LR Parser: How It Works?

In Fork-Merge LR (FMLR) parsing, a priority queue of LR subparsers is used. Each subparser in the queue consists of an LR parser stack, a presence condition and the head. The head of a subparser is the next token to be processed. It can be either a conditional or an ordinary token. A subparser is represented as follows:

$$p := (c, a, s)$$

where p is the subparser, c is the presence condition, a is the head of subparser, and s is the LR parser stack.

At first, the priority queue is initialized with a subparser containing the initial token as head, true as presence condition and an empty LR parser stack. The token-follow set is computed for the subparser in the queue. The token-follow set captures the variability of source code. It includes the first language token on each path and the corresponding static conditional.

For example, consider a subparser with p.c as presence condition and p.a as head. If p.a is an ordinary token, the resulting token-follow set contains a single element $T = \{(p.c, p.a_0)\}$, where a_0 is the next language token in the source code. An LR action is performed on the subparser and the single element in T. If the result of LR action is not accept or reject, then the subparser is rescheduled.

If the head of the subparser p.a is a static conditional, then the token-follow set returns more than one element. In this case, a fork action is performed on the current subparser. The fork action results in a set of subparsers, where each subparser in the set represents a distinct element from token-follow set.

$$Fork(T, p) := \{(c, a, p.s) \mid (c, a) \in T)\}$$

The resulting set of subparsers is rescheduled by inserting them into the queue. These new subparsers replace the current subparser. The presence condition of each subparser is distinct and the presence condition of all subparsers together recognizes all configurations.

Merge action combines subparsers that have the same head and LR parser stack. The merged subparser replaces the original subparsers in the priority queue.

$$Merge(Q) := \{(\bigvee p.c, a, s) \mid a = p.a \ and \ s = p.s \ \forall \ p \in Q\}$$

The presence condition of the merged subparser is the disjunction of the presence conditions of the original subparsers. The priority queue makes sure that the subparsers are merged at the earliest point. The next section explains the fork-merge LR parsing with an example.

3.2 Optimizations to FMLR parsing

In MAPR [26], the configuration-preserving parser forks a subparser for every branch of every static conditional, including empty branches. This will lead to many unnecessary subparsers. To limit the number of subparsers, Gazillo and Grimm [8] came up with two optimization techniques: use of token-follow set and lazy-shifts.

As explained in previous section, the token-follow set captures the actual variability of the source code by finding the tokens that follow each branch of a static conditional. The token-follow set consists of a presence condition and token pair corresponding to each branch of static conditional, except for an empty branch. This restricts the number of subparsers that are forked.

The static conditionals with empty or implicit branches usually results in a follow-set whose token requires a shift as next LR action. FMLR steps subparsers by position of head. And the subparser with first such token performs its shift and can merge even before the next subparser can perform its shift. In such cases, forking the subparsers eagerly is not useful. The lazy-shift delays the forking of subparser by producing multiple heads for the subparser. It forks off a single-headed subparser p' for earliest head, performs shift action on p' and reschedules p' and the multi-headed subparser.

$$Lazy(T, p) := \{ \bigcup \{(c, a)\} \ \mid \ Action(a, p.s) = \text{`}shift\text{`} \ \forall \ (c, a) \in T\}$$

To improve the performance of the parser further, Gazillo and Grimm [8] introduced two more optimization techniques: early reduces and shared reduces.

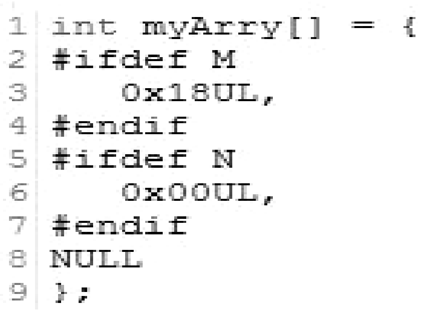
Early reduces prevent the subparsers from outrunning each other and increase the merging opportunities. Since reduces does not change the head of a subparser, when all subparsers in a priority queue have same head, the subparsers which reduce are given higher priority than the subparsers that shift.

Shared reduces eliminate the duplicate work done by the parser. If all tokens in a follow-set reduces to same nonterminal, instead of forking the subparser and reducing their stacks in the same way, it is more efficient to reduce a single stack for several heads. Shared reduce results in multi-headed subparser.

$$Shared(T, p) := \{ \bigcup \{(c, a)\} \mid Action(a, p.s) = \text{`}reduce\ n\text{`} \ \forall \ (c, a) \in T\}$$

3.3 An Example Illustrating FMLR Parsing

Consider a sample C code that has static conditional statements:



```
int myArry[] = {
#ifdef M
    0x18UL,
#endif
#ifdef N
    0x00UL,
#endif
NULL
};
```

Figure 3.1: An example code that has static conditional statements.

The FMLR parser initializes the queue with a subparser for the initial token int. The initial state of priority is shown in figure 3.2.

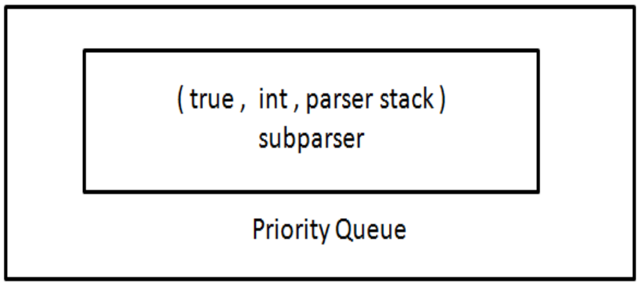


Figure 3.2: Initial state of the priority queue.

The parser stack in the initial state is empty, with int as lookahead. For each token until #ifdef the token-follow set method returns a single element. Hence, the FMLR parser performs an LR action on each of these tokens. The state of parser stack at this point, is as shown in figure 3.3.

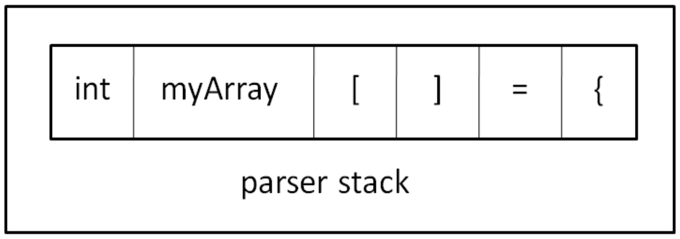


Figure 3.3: Parser stack before shared reduce.

When the #ifdef token is encountered, the token-follow set T1 is computed, the starting symbol of each branch and its presence condition are returned.

$$T1 = \{(M, 0X18UL), (!M\&\&N, 0X00UL), (!M\&\&!N, NULL)\}$$

Since all tokens in token-follow set T1 reduce to the nonterminal InitializerList, shared reduces turns the current subparser into a multi-headed subparser, p1 = (H1, parser stack) with H1 = T1 (Figure 3.4).

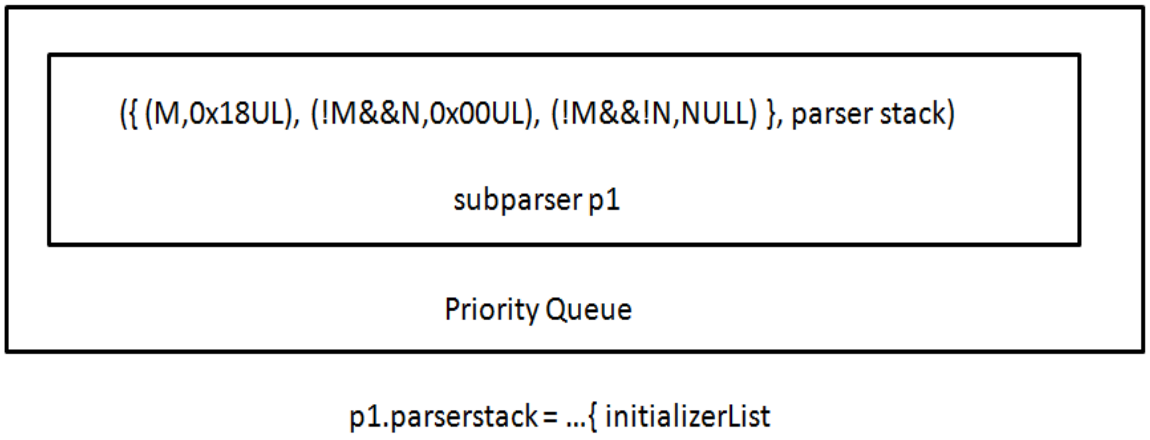

({ (M,0x18UL), (!M&&N,0x00UL), (!M&&!N,NULL) }, parser stack)
subparser p1
Priority Queue
p1.parserstack = ...{ initializerList

Figure 3.4: Priority queue after shared reduces.

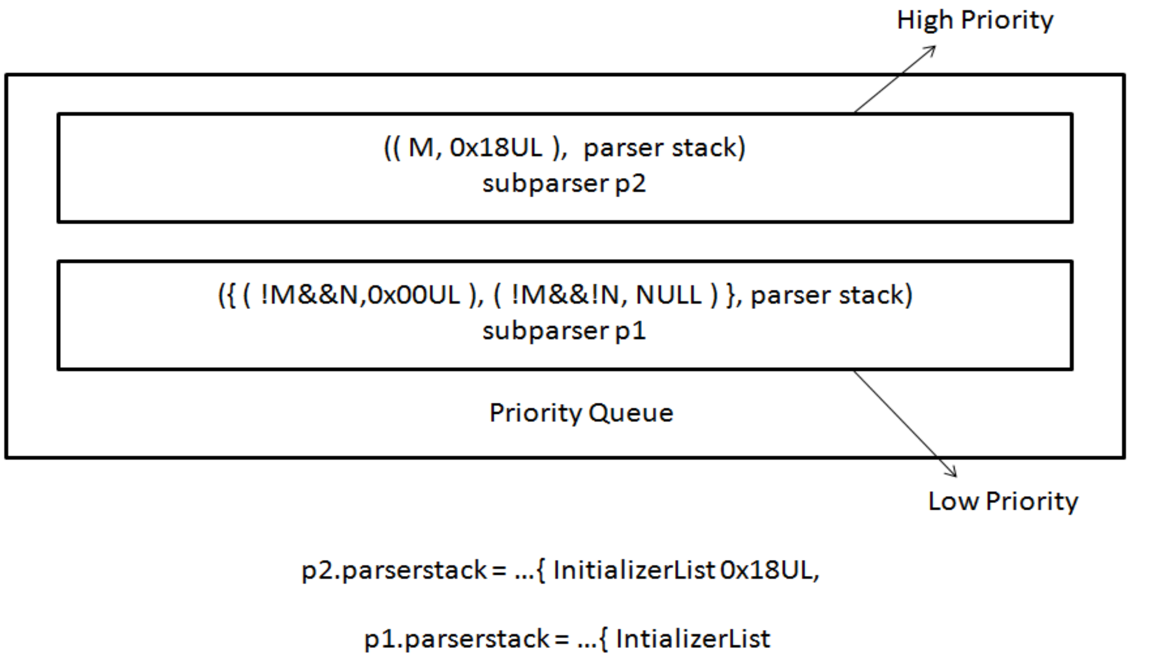

High Priority
((M, 0x18UL), parser stack)
subparser p2
({ (!M&&N,0x00UL), (!M&&!N, NULL) }, parser stack)
subparser p1
Priority Queue
Low Priority
p2.parserstack = ...{ InitializerList 0x18UL,
p1.parserstack = ...{ IntializerList

Figure 3.5: State of priority queue after fork action.

In the next iteration, the parser stack reduces and now since all tokens in H1 shifts, lazy-shift produces same multi-headed subparser. FMLR parser now forks a single headed subparser p2 for presence condition M and shifts 0X18UL onto its stack. The current state of the priority queue is as shown in figure 3.5. Next, the FMLR performs a shift on subparser p2, the comma after the token 0X18UL is shifted onto the stack.

The FMLR now updates the head of p2 to conditional on line 5-7 and computes the token-follow set as

$$T2 = \{(M\&\&N, 0X00UL), (M\&\&!N, NULL)\}$$

Now since all tokens in T2 reduces to InitializerList, shared reduce produces a multi-headed subparser p3 = (H2, s), where H2= T2. At this point the subparsers p1 and p3 have same head but different stack elements (Figure 3.6).

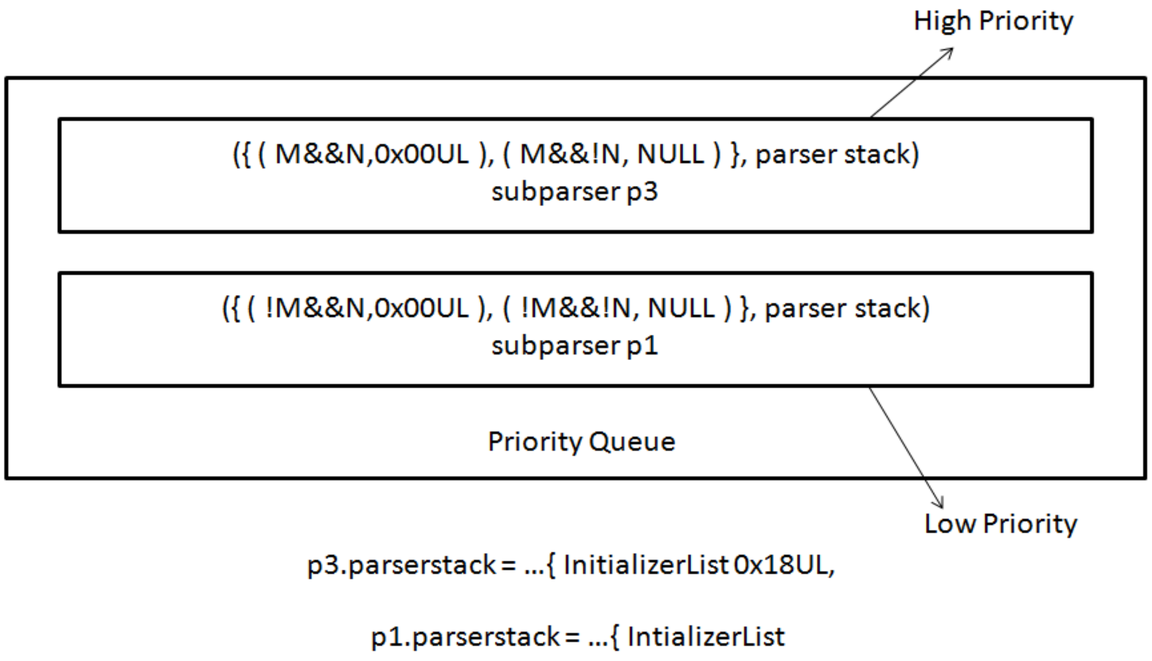

High Priority
({ (M&&N,0x00UL), (M&&!N, NULL) }, parser stack)
subparser p3
({ (!M&&N,0x00UL), (!M&&!N, NULL) }, parser stack)
subparser p1
Priority Queue
Low Priority
p3.parserstack = ...{ InitializerList 0x18UL,
p1.parserstack = ...{ IntializerList

Figure 3.6: Priority queue with subparsers having same head but different parser stacks.

In the next iteration, p3 reduces due to early reduces and yields same stack as p1. The subparsers p1 and p3 are now allowed to merge since they have same stack and head. The merge action disjoins M and !M for all presence conditions and therefore M is eliminated from presence condition.

The FMLR parser repeats the same actions for the conditional N at line 5-7. This way, FMLR parses 22 configurations with two subparsers. Grimm and Gazzillo [8] in their work

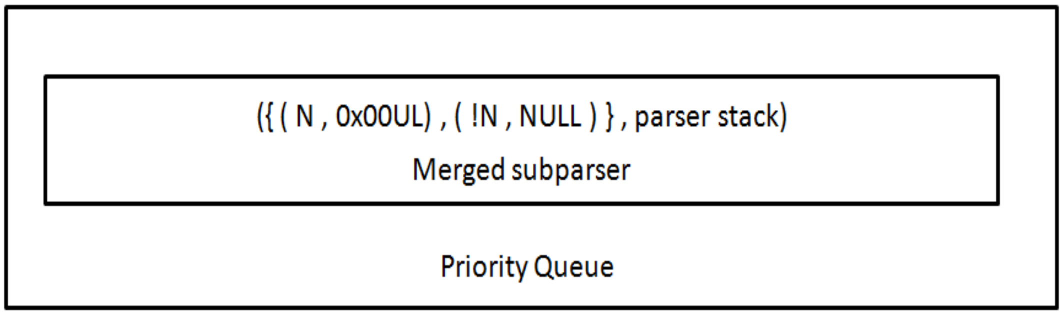


Figure 3.7: State of priority queue after merging.

evaluated FMLR parser by parsing Linux kernel. They found that FMLR requires less than 40 subparsers while MAPR [26] fails to parse most of the source files.

Chapter 4

Fork-Merge parsing in OpenRefactory/C: Implementation

IDEs for C support only limited refactoring, making programmers do the additional refactoring manually. The existing refactorings for C tend to be slow and buggy [21]. The C programming IDEs ignore the multiple configurations of the C preprocessor, which result in inaccurate transformations. Due to the presence of preprocessors and multiple configurations, static analysis of C programs is complicated and hence IDEs for C does not support sophisticated refactorings. OpenRefactory/C is a refactoring tool that resolves these challenges of building C program transformations.

OpenRefactory/C uses custom preprocessor and custom parser to support analysis and transformation of code containing preprocessor configuration. The main goal of OpenRefactory/C is to produce correct results of transformations and with all possible configurations. OpenRefactory/C uses mutable abstract syntax trees (ASTs) as its primary program representation for code transformation. The entire framework of OpenRefactory/C is written in Java. Source code transformation is performed by modifying the AST and later traversing the tree to output the revised source code. Behrang in her thesis [23] explained the changes made to the preprocessor and the lexical analyzer to achieve the required results. The modifications made to parser are explained in this chapter.

4.1 Modifications made to Parser

The grammar productions for C99 were extended to support C preprocessor directives. The changes to the C99 grammar were made based on the section 6.10 in ISO standard for C99. An LALR(1) parser for extended C grammar was generated using Ludwig [2]. The extended grammar allows the conditional directives to appear in between complete C constructs such as declarations, statements, and function definitions. In many cases, the conditional directives appear in the middle of complete C construct and the parser fails to construct an abstract syntax tree. We modified the OpenRefactory/C parser to successfully handle and parse the source code when the conditional directives appear at an unexpected location in the program.

We modified the OpenRefactory/C parser to implement Fork-Merge LR parsing algorithm designed by Gazzillo and Grimm [8]. Using the Fork-Merge technique, we can handle the conditional directives that violate the syntax of extended grammar.

4.2 Modifications made to Fork-Merge parsing

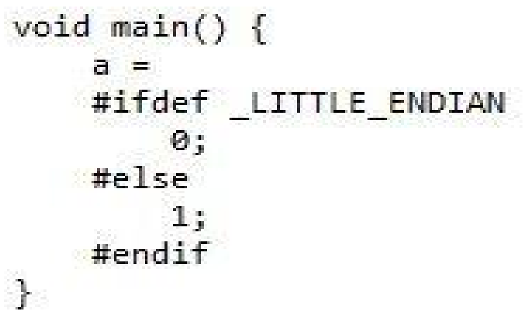
As explained in chapter 3, Fork-Merge parser is capable of preserving multiple configurations efficiently. We use similar technique in our parser to handle multiple configurations. In FMLR parsing, the entire parser was forked for every non-empty branch of a static conditional. While forking, the entire parser stack of the original parser was copied into the forked subparsers. In OpenRefactory/C, we fork the parser stacks instead of parsers and include all required information such as guarding conditions inside the parser stack.

In OpenRefactory/C the extended grammar allows conditional directives to appear in between certain complete C constructs such as declarations, statements and function definitions. Therefore, the AST constructed while parsing should have the choice nodes only at the nodes that represent these C constructs. To satisfy this requirement, we have modified the merge action of FMLR parser. In FMLR parsing, two or more subparsers are merged when they have same heads and parser stacks. In our parser, we merge parser stacks only when their lookaheads are same and when both parser stacks have reduced to either a statement, a declaration or a function definition.

4.3 Implementation of graph-structured stack

In OpenRefactory/C, an LALR(1) parser was generated for the extended grammar. We modified LALR(1) parser to implement fork-merge technique. As explained in chapter 2, an LALR(1) parser uses a parser stack to keep track of elements that are parsed. The parser stack contains a stack of states and a stack of symbols. We included more information in the parser stack to support the fork-merge actions.

For each branch of #ifdef statement, the fork action creates a new parser stack. Every new parser stack created holds a reference to the original parser stack object and we call the reference as parentstack to the new parser stack. The presence condition for each branch of #ifdef statements is stored in their respective parser stack object. For example, when the piece of code in figure 4.1 is parsed, two new parser stacks are created and they have a reference to the parentstack as shown in the figure 4.2. All three stacks together create a graph like structure.



```
void main() {
    a =
    #ifdef _LITTLE_ENDIAN
        0;
    #else
        1;
    #endif
}
```

Figure 4.1: An example of code with simple conditional statements.

Using graph structured stack, we avoid duplication of parsing actions. To perform a single action on multiple parser stacks parallely, we created a new class called parser stackList. The parser stackList represents the graph structured stacks. Each parser stackList object contains a priority queue. The priority queue consists of parser stacks that has same parentstack. To perform LR actions on parser stacks in the priority queue, we need to consider the following scenarios:

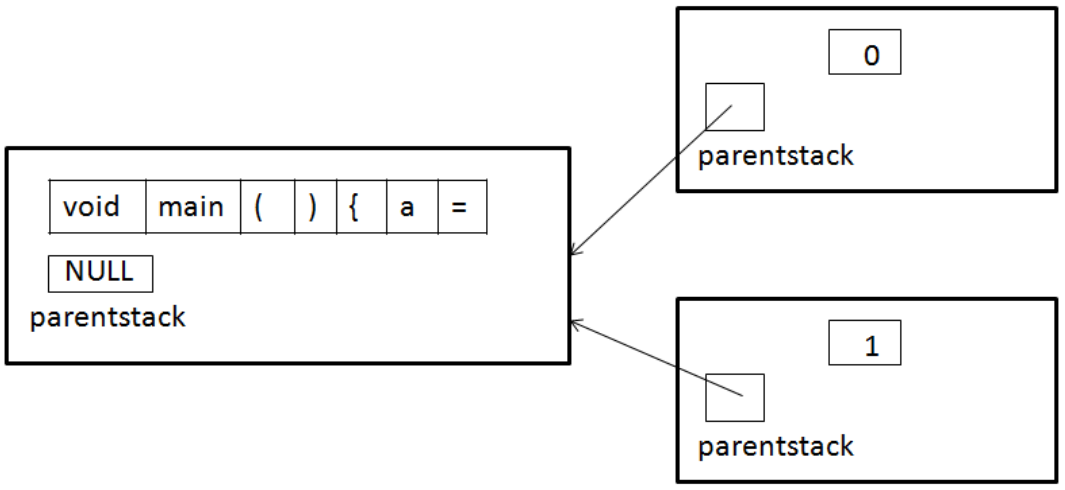


Figure 4.2: Graph structure stacks formed due to fork action.

- Every parser stack in the queue reduces: In this case, we reduce every parser stack by calling reduce action. Most of the times, after forking, the parser stack needs to have elements from parentstack to perform reduce action. In section 4.7. we explain in detail how we handle this situation.

- Every parser stack in the queue shifts: In this case, we shift the lookahead for each parser stack and read next token from token stream to set the next lookahead.

- One or more parser stack in the queue reduces, while remaining parser stacks shift: when one or more parser stack in the queue reduces, higher priority is given to the parser stacks that reduce. The reduce action is performed for the parser stacks that reduce while parser stacks that shift wait.

- One or more parser stack has syntax error and should be rejected: In this case, we try to recover the parser stack from syntax error, if it fails to recover then the parser is halted.

- One or more parser stack has to be completely parsed and should be accepted:

  – When every parser stack in the queue is completely parsed we can simply merge all the parser stacks, return the top element and end parsing.

  - When two or more, but not all parser stacks have completely parsed then those which are completely parsed are merged and wait for other parser stacks to complete.
  - If it is just one parser stack that finished parsing, the parser stack waits for other parser stacks to complete.

We came up with the following strategy to handle the above scenarios:

```
1.  code = calculateActionCode
2.      if code == shift then
3.          call shift method for parser stackList
4.      else if code == reduce then
5.          call reduce method for parser stackList
6.      else
7.          Merge all parser stacks in the queue and accepted the merged parser stack.
8.      endif
9.
10. function calculateActionCode
11. for each parser stack in the queue do
12.     find the action to be executed on the parser stack
13.     if action == reduce then
14.         return reduce code
15.     else if action == shift then
16.         shiftValue = action code
17.     else if action == accept then
18.         add the parser stack to acceptlist
19.         acceptValue = action code
20.     else
21.         attemptToRecoverFromSyntaxError
```

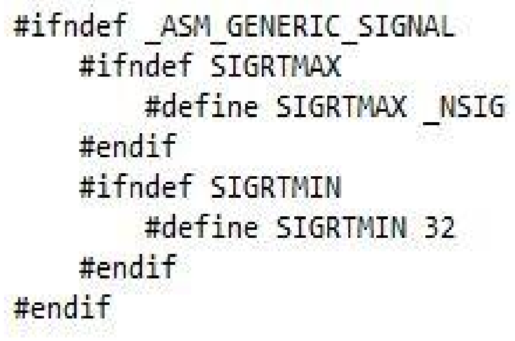
```
22.             if attemptToRecoverFromSyntaxError fails then
23.                 halt the parser
24.             endif
25.         endif
26.         if acceptlist.size == priorityQueue.size then
27.             return acceptValue
28.         else
29.             return shiftValue
30.         endif
31. end function
```

After finding the appropriate action code for parser stackList, based on the action code either shift, reduce, accept or reject action is performed. The token-follow set in FMLR parser captures the actual variability of the source code. It avoids the unnecessary forking by ignoring the empty branches of conditional directives. In our parser, instead of finding token-follow set for each branch, we do not fork for a conditional branch until we find an ordinary token inside the branch or one of its nested branches. In the next section we explain the factors we considered to decide whether to fork a conditional branch or not.

4.4 Making fork decision

The lexical analyzer feeds a stream of tokens to the parser. The parser reads these tokens one at a time. If the token read is an ordinary token, then it is set as lookahead for current parser stack. If the token read is a conditional directive, then the fork action is performed. When the code has a simple conditional statement as in the example shown in Figure 4.1, the fork action is simple and we need to keep track of one level of conditionals. However, in real time conditional directives appear haphazardly. Generally, the following cases may occur:

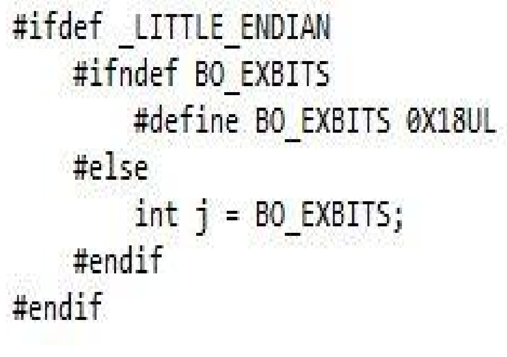
- Static conditionals with empty branches: The empty branches are sometimes nested as shown in Figure 4.3. In the example, the macros are expanded with the conditional statements during preprocessing. Hence forking the conditional directives in figure 4.3 will result in redundant parser stacks. We need to identify such cases and avoid forking.



```
#ifndef _ASM_GENERIC_SIGNAL
    #ifndef SIGRTMAX
        #define SIGRTMAX _NSIG
    #endif
    #ifndef SIGRTMIN
        #define SIGRTMIN 32
    #endif
#endif
```

Figure 4.3: Example for static conditionals with empty branches.

- Static conditionals that has an empty branch and a non-empty branch: In this case it is tedious to keep track of the conditionals that should to be considered for forking. For the example shown in Figure 4.4, we need to fork the #else branch and avoid forking for nested #ifndef condition.



```
#ifdef _LITTLE_ENDIAN
    #ifndef BO_EXBITS
        #define BO_EXBITS 0X18UL
    #else
        int j = BO_EXBITS;
    #endif
#endif
```

Figure 4.4: Example for nested static conditionals.

To handle the above cases, we designed an algorithm that keeps track of all conditionals and determines the level of nesting to be considered for forking the parser stack. We created a class called Directive that keeps track of directives and their conditions. We use list as main data structure to store the objects of Directive class. Each object of Directive class represents a #if-[#elif-][#else-]#endif statement.

The variable isDirective is used to represent whether the last tokenRead is an ordinary token or a conditional directive. It is initially set as false and is set as true if the tokenRead is a conditional directive. The do-while loop executes until the next token read is not a conditional directive. For every if-directive read from the token stream, we create a new Directive object and is stored in Directive list, dirList. The directive object consists of a list of directives and a list of conditions corresponding to the directives. When the tokenRead is an elif-directive or an else-directive, we add the directive and the condition to the latest object that was added to dirList. When the tokenRead is an endif-directive there are two cases we need to deal with:

- endif-directive for an empty #if-#elif-#else statement - In this case we should ignore the endif-directive and remove the last Directive object from dirList.
- When endif-directive occurs after forking #if, #elif or #else directives - In this case we should consider the endif-directive and perform the required actions.

To handle these two cases, we use isFork variable in the Directive class to represent if at least one of the directives in the object has been forked. When an endif-directive is encountered and isFork is false, this means that the conditional statement has empty branches and hence forking should be avoided. We use readDirective method to handle conditional directives that are supposed to be forked. In the algorithm, when the next token read is an ordinary token, the do-while exits at line 23 and for every conditional directive in dirList, appropriate fork action is performed by calling readDirective method.

```
1. do
2.     isDirective = false
3.     if tokenRead is if-directive then
4.         isDirective = true
```

```
5.          dirList.add(new Directive(tokenRead, condition))
6.          tokenRead = next token from lexer
7.      endif
8.      if tokenRead is elif-directive OR tokenRead is else-directive then
9.          isDirective = true
10.         dir = dirList.getLastElement()
11.         dir.add(tokenRead, condition)
12.         tokenRead = next token from lexer
13.     endif
14.     if tokenRead is endif-directive then
15.         isDirective = true
16.         if dirList.getLastElement().isFork then
17.             readDirective(tokenRead, )
18.         endif
19.         dirList.removeLastElement()
20.         tokenRead = next token from lexer
21.     endif
22. while isDirective
23. for each directive object dir in dirList do
24.     for each (directive, condition) pair in dir do
25.         readDirective(directive, condition)
26.         dir.isFork = true
27.         if directive is endif-directive then
28.             dirList.removeLastElement()
29.         endif
30.     end for
31. end for
```

```

function readDirective(directive, condition)
    if directive is if-directive then
        fork-for-ifs()
    else if directive is elif-directive OR directive is else-directive then
        fork-for-else()
    else
        parser stack = parser stack.parentlist
    endif
end function
```

The readDirective method takes a directive and a condition as parameter and based on the type and nesting level of the directive different fork actions is performed. Next section explains different kinds of forking we implemented in our parser.

4.5 Different fork actions

Following are the different fork actions that we implemented based on the type and nesting level of the directives:

- Forking for if-directive: when an if-directive is passed as a parameter, the parser stack is forked based on the nesting level of the if-directive. Following cases may occur with forking an if-directive.
  - For simple conditional statements similar to example in figure 4.1, we need to fork the current parser stack. The fork action creates new parser stack, a reference to the current parser stack is assigned as parentstack to the new parser stack. The new parser stack is added to the priority queue.
  - When the if-directive occurs inside a conditional statement, we need to fork for nested if-directive. In this case, we replace the current parser stack from priority

queue with the new parser stack that is created and current parser stack is assigned as the parentstack of new parser stack.

- The elif and else directives are forked in a similar way. These two directives always occur after if-directive or an elif-directive itself. The fork action for else and elif directives also creates a new parser stack. However, the parentstack for new parser stack is the parentstack of current parser stack.

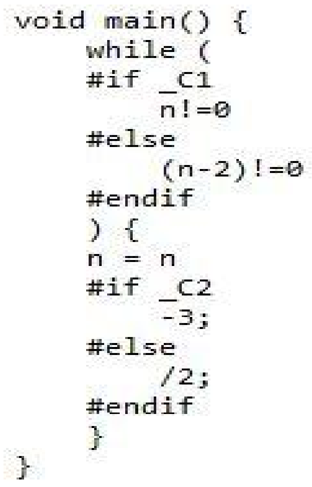
- endif-directive: When an endif-directive is encountered, we need to parse all parser stacks in the priority queue parallely. Every action that was executed on a single parser stack must be executed on multiple parser stacks. For this purpose, we override a few methods defined for parser stack in parser stackList class. Next section explains the changes made to these methods.

4.6 Parsing multiple parser stacks

In section 4.3, we have explained how LR actions are executed on multiple stacks. In this section, we explain the implementation of various fork actions in parser stackList class to handle consecutive conditional statements. Generally, there are two cases the consecutive conditionals might occur.

- For simple consecutive conditional statements as shown in figure 4.5 we fork every branch in the preceding conditional statement. The resulting graph-structured stack will have four branches as shown in the figure 4.6.

- Consecutive conditional statements nested in a conditional statement as shown in figure 4.7. In this case, we apply the same strategy as nested fork method explained in section 4.5, and fork every branch in the preceding statement.



```
void main() {
    while (
    #if _C1
        n!=0
    #else
        (n-2)!=0
    #endif
    ) {
    n = n
    #if _C2
        -3;
    #else
        /2;
    #endif
    }
}
```

Figure 4.5: Example of simple consecutive conditional statements.

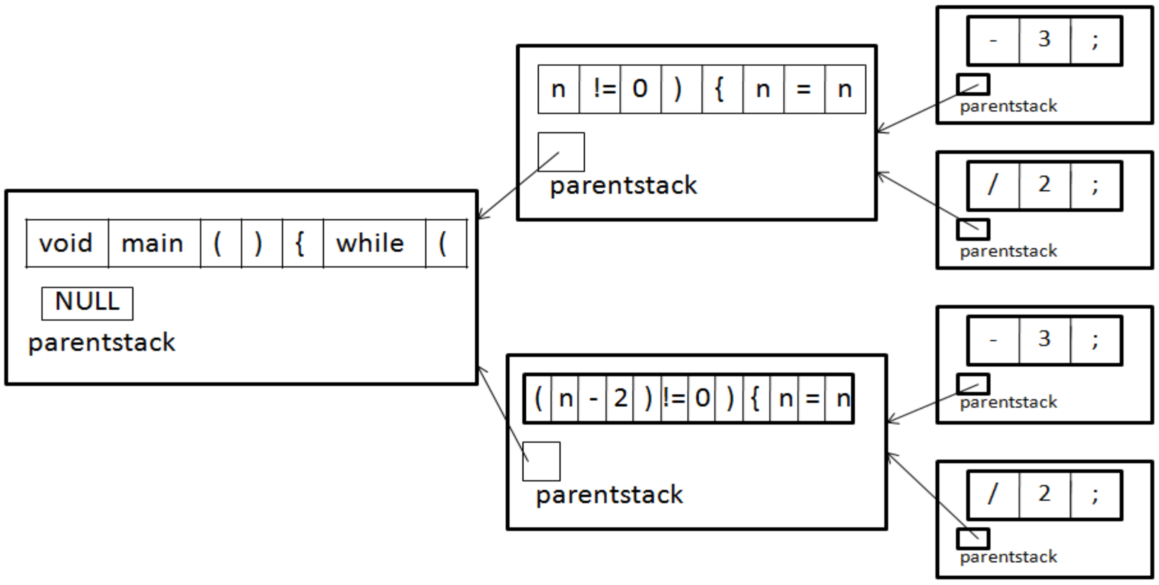
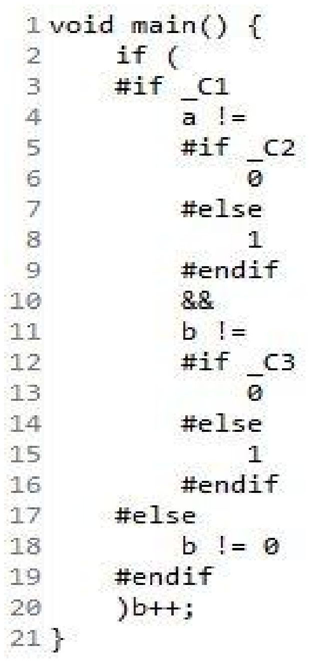


Figure 4.6: Graph structured stack for simple consecutive condition statements.



```
void main() {
    if (
    #if _C1
        a !=
        #if _C2
            0
        #else
            1
        #endif
        &&
        b !=
        #if _C3
            0
        #else
            1
        #endif
    #else
        b != 0
    #endif
    )b++;
}
```

Figure 4.7: Consecutive conditional statements nested in a conditional statement.

4.7 Performing reduce action after forking

The reduce action in LALR(1) replaces one or more symbols from the top of the stack to the left-hand side of the grammar production. While forking we created an empty stack with same state as topstate of the parser stack that was forked. We continue to parse the new parser stack that was forked. The shift action on the new parser stack does not cause any problem, however the reduce action requires one or more symbols on the parser stack.

In example 4.1, after forking the parser stack for if-directive, the new parser stack p1 shifts the symbol '0' and then reduce. Due to fork action, the top state of the parentstack is copied as starting state for p1. The state of p1 now reflects that it should reduce to an assignment-statement. Since there are not enough symbols on the p1 to reduce, we should consider the symbols from parentstack. One way to do this is to pop the symbols required to reduce from parentstack and use them to complete the reduce action. In the example, p1 needs two more symbols from its parentstack, so we pop the symbols 'a' and '='. Using these two symbols and '0' the p1 reduces to assignment-expression. However, since fork actions usually result in two or more parser stack with same parentstack, wrong symbols are read when the other parser stacks try to read from the same parent. In our example, for else-directive a new parser stack p2 is created with same parentstack as p1, after popping the symbols ’a’ and ’=’. Now when the p2 has to reduce to assignment-expression, it has the wrong symbols ’)’ and ’{’ on top of its parentstack as shown in figure 4.8.

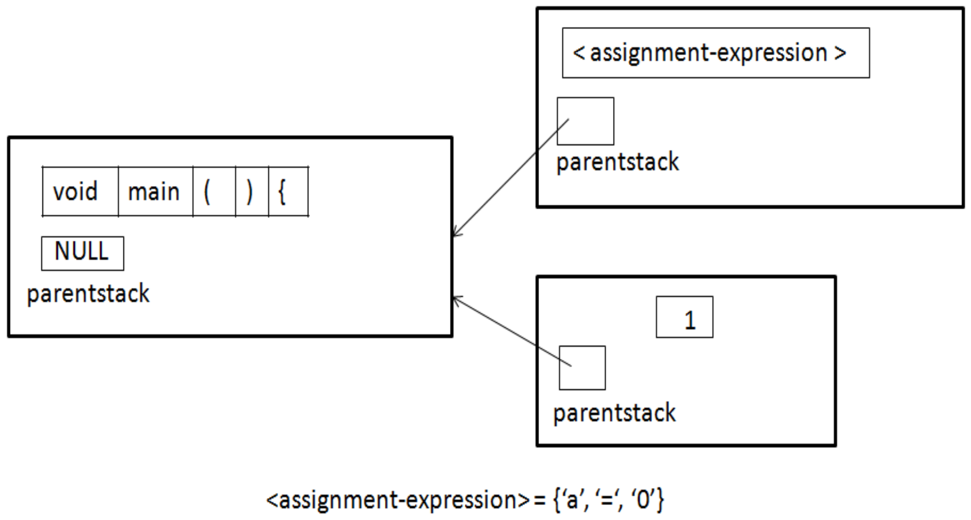

< assignment-expression >
parentstack
void
main
(
)
{
NULL
parentstack
1
parentstack
<assignment-expression> = {‘a’, ‘=‘, ‘0’}

Figure 4.8: Reading symbols by popping from parentstack.

To handle this problem, instead of popping the symbols from parentstack, we copy them into child stack to perform reduce action. Now when p2 reads from parentstack it still contains the expected symbols. We allow merging only for statements, declarations, function definitions and translational unit. After reducing to assignment-expression, p1 has to reduce to statement before merging with p2. This reduce action requires more symbols from parentstack to be copied into p1 stack. Since we are not popping the symbols from parentstack, the next symbols to be read from parentstack are not known. To solve this problem, we use a variable called readParent to keep track of the number of symbols that are read from parentstack for each parser stack. In the above example, when the parser stack p1 copies the symbols ’a’ and ’=’ from parentstack and reduces the p1 to assignment-expression, the readParent variable for p1 is set to 2. Now when we need to read more elements from parentstack, we skip the top 2 symbols on top and copy the required number of symbols and update the readParent value.

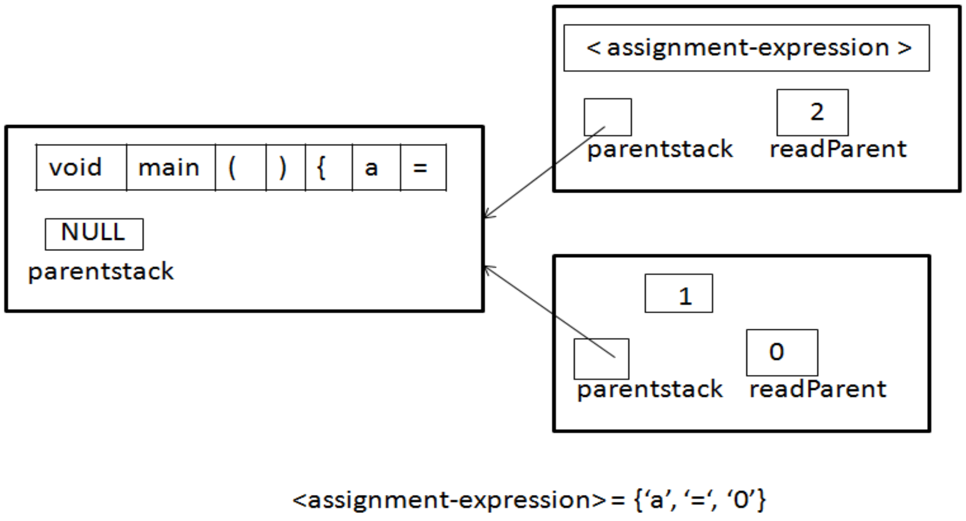


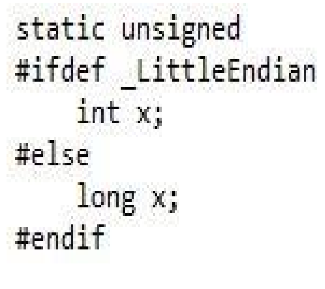
Figure 4.9: Copying from parentstack using readParent variable.

For nested conditional statements, it is obvious that a parentstack will inturn have another parentstack. In some scenarios it is not only enough to read from the immediate parentstack but also from higher levels of the parentstacks. Consider the sample code in figure 4.6, we name the parser stacks created due to forking as pX, where X is the line number at which the conditional causing the fork action appears. The parser stack for condition _C2, p5 reduces first to a conditional-expression, at this point it copies the symbols 'a' and '!=' from its parentstack p3. In the next iteration, parser stack has to copy the symbols 'if' and '(' from p3 to reduce to a statement, it. We came up with a strategy to readback symbols that are required for reducing from the parentstack. We call the readback method recursively to readback from higher levels of the parentstacks.

```
1. find the number of symbols to be popped to perform reduce action
2. if symbolsToPop from parser stack ¿ size of parser stack then
3.     readBack(symbolsToPop)
4. endif
5. pop the symbols from parser stack.
```

```
6. push the new nonterminal on left-hand side of grammar production.
7. calculate the new state of the parser stack and push the state value.
8.
9. **function** readBack(symbolsToPop)
10.     symbolsFromParent = parentStackSize - readParent
11.     **if** symbolsFromParent ¡ symbolsToPop **then**
12.         parentStack.readBack(( symbolsToPop - symbolsFromParent))
13.         readParent = parentStack.readParent
14.         parentStack = parentStack.parentStack
15.     **endif**
16.     deep copy symbolToPop number of symbols from parentStack
17.     add the deep copied symbols to the parser stack
18.     readParent = readParent + symbolToPop
19. **end function**
```

In java, copying an object does not allocate a new memory location for the copied object, instead it creates a new reference to the object to be copied. Hence the changes made to the copied object are reflected in original object. The parser stack is a stack of objects that may contain tokens, terminals or nonterminals.



```
static unsigned
#ifdef _LittleEndian
    int x;
#else
    long x;
#endif
```

Figure 4.10: Sample code containing static conditionals.

Consider the sample code in figure 4.10. The tokens 'static' and 'unsigned' reduce to declaration-specifiers which is represented as ASTListNode in an AST. After forking for #ifdef-directive, a new parser stack p1 is created. The parser stack p1 reads next three tokens

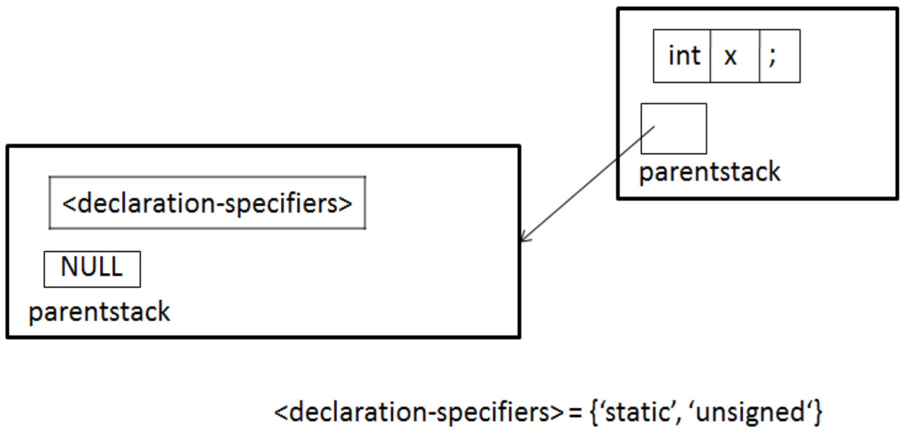


Figure 4.11: State of parser stack before copying symbols from parentstack.

and reduces to CDeclaration by copying the declaration-specifiers from its parentstack. The declaration specifier 'int' which is in #ifdef branch is added to the declaration-specifiers copied from the parentstack. If the declaration-specifiers object is not deep copied, then 'int' added to the declaration-specifiers in p1 is reflected in declaration-specifiers object in parentstack.

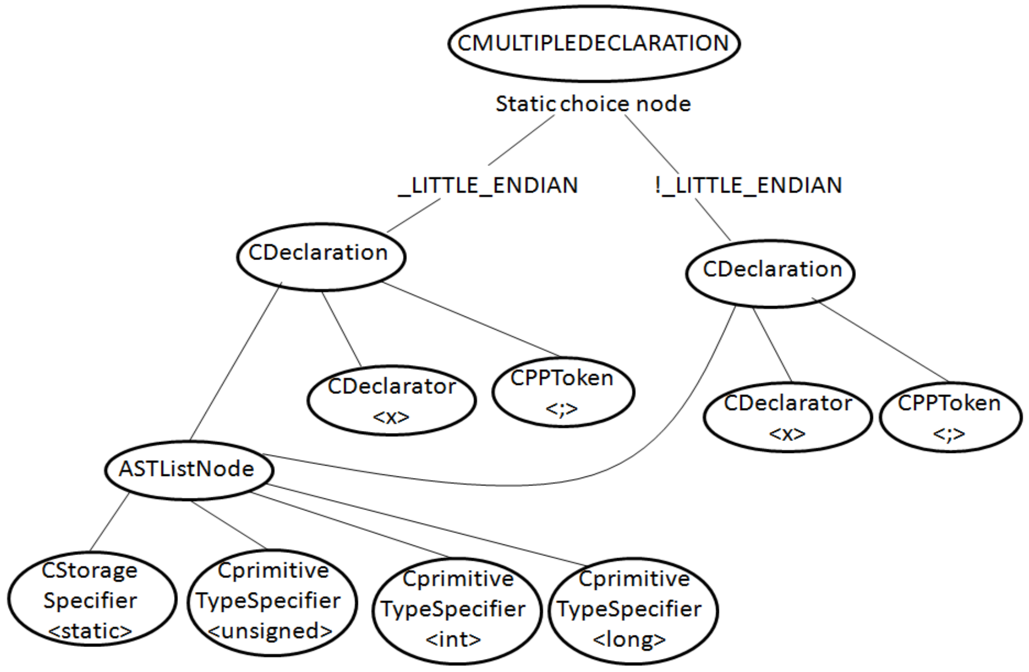
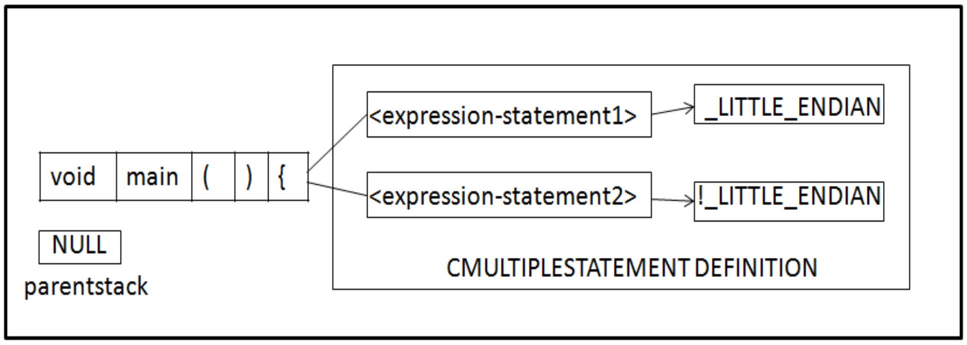


Figure 4.12: Incorrect AST as a result of not deep copying the objects from parentstack.

Now when fork action is performed for #else-directive, the new parser stack p2 has parentstack with wrong set of declaration specifiers: 'static', 'unsigned', and 'int'. Further when p2 is reduced to CDeclaration, the declaration specifier 'long' is added to the same declaration-specifiers object resulting in wrong AST as shown in figure 4.12. To avoid this, we need to deep copy each element while copying from parentstack. By deep copying the declaration-specifiers object from parentstack, when p1 reduces to CDeclaration, the declaration specifier 'int' is added to copied object and this change is not reflected in the declaration-specifiers of parentstack.

4.8 Merging parser stacks

Whenever the parser stacks in the queue reduces to statement or a declaration or a function definition, we merge the parser stacks. We merge the parser stacks that reduce either to statement, declaration or a function definition into a single parser stack. The merged parser stack replaces the parser stacks in the priority queue.



<expression-statement 1> = {<assignment-expression 1> , ';'}
<expression-statement 2> = {<assignment-expression 2> , ';'}
<assignment-expression 1> = {'a', '=', '0'}
<assignment-expression 2> = {'a', '=', '1'}

Figure 4.13: Merged parser stack for example in figure 4.1.

To merge two or more parser stacks, the parser stacks need to have same parentstack and should be reduced into either a statement, a declaration or a function definition. Based on the type of reduction, we create an object to hold these multiple definitions. We created

3 different classes for this purpose: MultipleStatementDefinition, MultipleDeclarations, and MultipleFunctionDefintions. These three classes implement the MultipleDefinition interface and inherit ASTNode class. Each of these classes has a list of conditionals and list of objects. These classes also define a hashmap object which maps the conditionals to their corresponding statements, declarations, or function definitions. Figure 4.13 shows the merged parser stack for example 4.1.

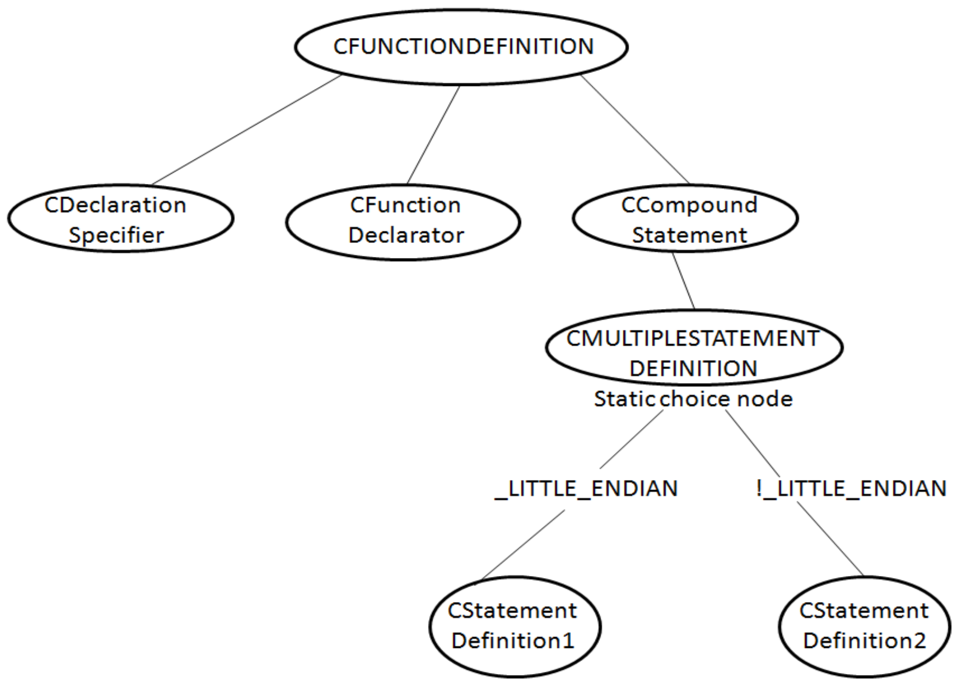


Figure 4.14: AST generated after parsing the example code in figure 4.1.

The resulting merged parser stack is parsed for remaining tokens. The merged parser stack is a single parser stack and is parsed like an ordinary parser stack. In the example, the '}' is shifted onto merged parser stack and is reduced to CFunctionDefinition. The resulted AST, as shown in figure 4.14, contains a MultipleStatementDefinition node which is a choice node.

4.9 Results

The modified OpenRefactory/C parser was tested by parsing the C standard library header files. We created C programs that includes C standard library header files and parsed these programs using the modified parser. The resulting abstract syntax tree is traversed and is compared with the original source code to check if the parser is correctly constructing the abstract syntax tree.

Among 24 C99 standard library header files, 19 header files parses successfully. In the remaining 5 header files, the parser is not able to recognize some of the identifiers that are defined using #define macros. This is due to a problem in expanding macros by OpenRefactory/C preprocessor which was modified to support multiple configurations.

Chapter 5
Conclusion and Future Work

In this thesis, we explained the modifications made to the OpenRefactory/C parser to support multiple configurations and hence build correct program transformations.

The parser was modified to implement Fork-Merge technique. Using Fork-Merge approach, the parser constructs an AST that includes C preprocessor and thereby include all possible configurations of the C program. In our parser, we fork a parser stack which is the main data structure used in a parser. Each parser stack after forking represents a distinct configuration. The parser stacks are merged when they reduce to a statement, a declaration or a function defintion. The parser stacks are merged using a choice node.

We allow only few productions to have choice nodes. This may result in exponential parser stack number. As a future work we will allow more productions to have static choice node in order to decrease the number of active parser stacks.

However, treating too many grammar productions as complete forces the refactoring tool to deal with too many static choice nodes. This complicates the refactoring process. We need to examine C language libraries and carefully select the productions to be considered for choice node.

Bibliography

[1] Ira D. Baxter, Michael Mehlich, Preprocessor Conditional Removal by Simple Partial Evaluation, WCRE '01 Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01), Page 281.

[2] Jeffrey L. Overbey and Ralph E. Johnson, Generating Rewritable Abstract Syntax Trees, Software Language Engineering, Pages 114 133.

[3] Alejandra Garrido and Ralph E. Johnson, Embracing the C preprocessor during refactoring, Journal of Software: Evolution and Process, Volume 25, Issue 12, pages 12851304, December 2013.

[4] Michael D. Ernst, Greg J. Badros, David Notkin, An Empirical Analysis of C Preprocessor Use, IEEE Transactions on Software Engineering, Volume 28 Issue 12, December 2002, Pages 1146-1170.

[5] Alejandra Garrido, Program refactoring in the presence of preprocessor directives, doctoral desertation, 2005.

[6] Lerina Aversano, Massimiliano Di Penta, Ira. D. Baxter, Handling Preprocessor-Conditioned Declarations, SCAM '02 Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation, Page 83.

[7] Kastner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, Thorsten Berger, Variability-aware parsing in the presence of lexical macros and conditional compilation, OOPSLA '11 Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications, Pages 805-824.

[8] Paul Gazzillo and Robert Grimm, SuperC: parsing all of C by taming the preprocessor, PLDI '12 Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, Pages 323-334.

[9] A. V. Aho and S. C. Johnson, LR Parsing, Journal ACM Computing Surveys (CSUR) Surveys Homepage archive Volume 6 Issue 2, June 1974, Pages 99-124.

[10] Don Roberts, John Brant, and Ralph Johnson, A Refactoring Tool for Smalltalk, Theory and Practice of Object Systems - Special issue object-oriented software evolution and re-engineering Volume 3 Issue 4, 1997, Pages 253 263.

[11] Alejandra Garrido and Ralph Johnson, Challenges of Refactoring C Programs, IWPSE '02 Proceedings of the International Workshop on Principles of Software Evolution, 2002, Pages 6-14.

[12] Dave Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, Paolo Tonella, Automated refactoring of object oriented code into aspects, in Proceeding of International Conference on software Maintenance (ICSM), 2005, Pages 2736.

[13] Mel O Cinneide, Automated refactoring to introduce design patterns, in Proceedings of the International Conference on Software Engineering, Pages 722 - 724, Jun 2000.

[14] Tom Mens and Tom Tourwe, A survey of software refactoring, in IEEE Transactions on Software Engineering Volume 30 Issue 2, Pages 126 139, Feb 2004.

[15] Alain Coutu, Automated Refactoring Tool, Masters Thesis, Pages 2-4.

[16] Frank Deremer, Practical Translators for LR(k) Languages, Technical Report, 1969.

[17] Donald E. Knuth, On the translation of languages from left to right, Information and Control Volume 8 Issue 6, December 1965, Pages 607639.

[18] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Prentice Hall 2006.

[19] Opdyke, William F., Refactoring Object-Oriented Frameworks Ph.D. diss., University of Illinois at Urbana-Champaign, 1992.

[20] I. Moore, Automatic Inheritance Hierarchy Restructuring and Method Refactoring, Proc. Object-Oriented Programming, Systems, Languages, Applications Conf., pp. 235-250, 1996.

[21] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. Systematic testing of refactoring engines on real software projects. In ECOOP, Volume 7920 of LNCS, pages 629653, 2013.

[22] OpenRefactory/C: An Infrastructure for Building Correct and Complex C Transformations Munawar Hafiz, Jeffrey Overbey, Farnaz Behrang, and Jillian Hall.

[23] F. Behrang, Static Program Analysis In Presence Of Multiple Configurations, Masters Thesis, Auburn University, Auburn, AL, 2014.

[24] International Organization for Standardization. ISO/IEC 9899:1999: Programming Languages C. Dec 1999.

[25] Nigel P. Chapman, LR Parsing: Theory and Practice, Cambridge University Press, New York, NY, 1988.

[26] M. Platoff et al. An integrated program representation and toolkit for the maintenance of C programs. In Proc. ICSM, pp. 129137, Oct. 1991.

[27] C. Kastner et al. Partial preprocessing C code for variability analysis. In Proceeding 5th VaMoS, pp. 127136, Jan. 2011.

[28] Aho, Ullman, The Theory of Parsing, Translation and Compiling. Prentice Hall, New Jersey, 1972.

[29] M. Tomita, Generalized LR Parsing, Springer Science and Business Media, 2012, pages 1-16.

[30] "LR Parser." From Wikipedia and the free Encyclopedia.
`http://en.wikipedia.org/wiki/LR_parser`

[31] "Abstract syntax tree." From Wikipedia and the free Encyclopedia.
`http://en.wikipedia.org/wiki/Abstract_syntax_tree`