Chapter 4 Building rc-Promela Parser

In order to automate refactoring, we found that the key is to design a front-end language, because CFSM is too low level, providing little guidance for refactoring automation. For that reason, we choose Promela as our front-end language. Promela is a design language of SPIN that is used to describe behaviors of a system. We select a subset of Promela's syntax and add refactoring commands to its syntax. The language is called *rc-Promela*, where "*r*" stands for "*refactoring*" and "*c*" stands for "*ccs*." In this chapter, we describe how the *rc-Promela* parser is built.

4.1. The architecture of *rc-Promela* parser

In Figure 4.1, we illustrate the data flow diagram of *rc-Promela* parser. Given a model written in *rc-Promela*, *Scanner* first translates the input characters into tokens. Second, in Parser stage, tokens are checked to see if they violate the grammar. In third stage, the *Parser* generates an abstract syntax tree（AST） for latter stage. In the last stage, we use AST to generate CCS state graphs for refactoring transformation. In next sections we will introduce every stage of how we implement every component of rc-Promela parser. In these stages, the functionality of Scanner is well-known, so we ignore it in this parser.

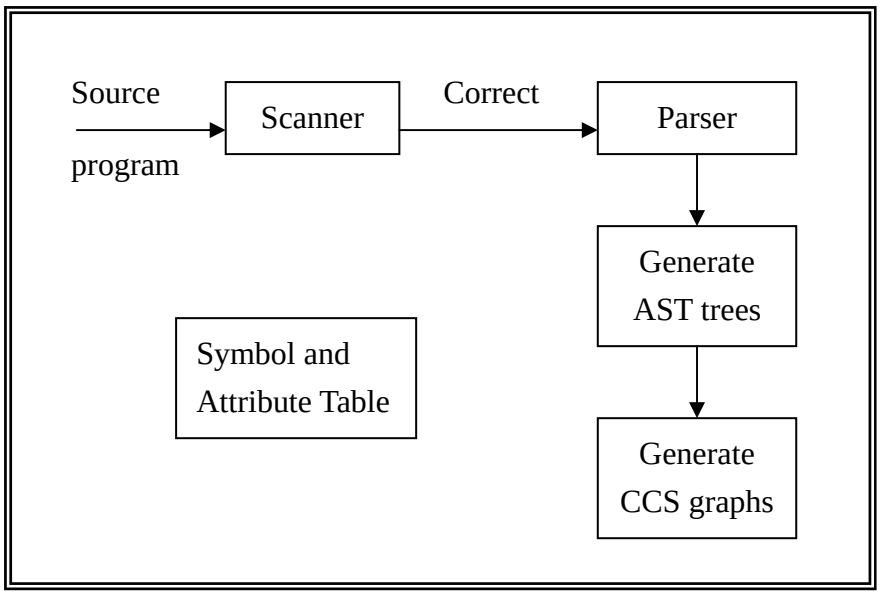

Figure 4.1. Structures of building rc-Promela parser.

4.2. *rc-Promela* grammar

In SPIN, the grammar of Promela has been specified in http://spinroot.com/spin/Man/grammar.html. Current version is SPIN version 3.0. We select a subset of this grammar and add new statements for refactoring, and we call it *rc-Promela*. We use YACC to specify grammar of *rc-Promela*. The whole grammar is listed in Appendix A.

In this section, we list the semantics and statements that *rc-Promela* doesn't provide:

- Process declaration with parameters: The parameters in process declaration should be empty in *rc-Promela*, as in the following:

  proctype NAME ( ) {

  // process's body

}

This implementation tools would make our implementation easier, so we exclude the functionality from *rc-Promela*.

- Statements related to temporal logic properties, such as *never claim statements, event assertions, and assert statements*: This functionality are currently unavailable because verifying temporal logic in compositional analysis need further research. Thereby, these statements are not supported in *rc-Promela*.

- *inline statement, printf( )*: Our *rc-Promela* produces CCS state machine as output, it doesn't produce executable C code like SPIN. So we excluded these statements from *rc-Promela*.

- Share variables: The usage of share variable is excluded from *rc-Promela.* How to do compositional analysis for share variables need further research.

- The buffer length of channel: Promela allows arbitrary number of buffer length in a channel, however, our refactoring focuses on rendezvous style communication. So, the buffer length should be zero in channel declaration.

4.3. Parser—parsing grammar

After scanner program and YACC file are done, we need only **main( )** that calls the Parser **yyparse( )** function to combine scanner and parser program. On a UNIX system, YACC takes the grammar and creates y.tab.c, the C language parser, and y.tab.h, the include file with the token file definitions. yyparse( ) will call yylex( ) to scan input and return tokens to the parser. In y.tab.c that LALR (1) parser will check these tokens to see if they satisfy by grammar rules.

4.4. Building abstract syntax trees

When parser checks input tokens, it will build *abstract syntax tree* (AST) for this source program. An *abstract syntax tree* is a condensed form of parse tree which omits elements such as parentheses and semicolons, which are syntactically important but have no semantic value. For example, it need not include intermediate nodes in expression subtrees for the non-terminal symbols that are usually introduced in your grammar to describe operator precedence and associativity. The advantage of creating an abstract syntax tree is that once the whole tree is available, more sophisticated analysis can be done than can be done while parsing. You can traverse the tree as many times as necessary to collect various kinds of data to do other analysis.

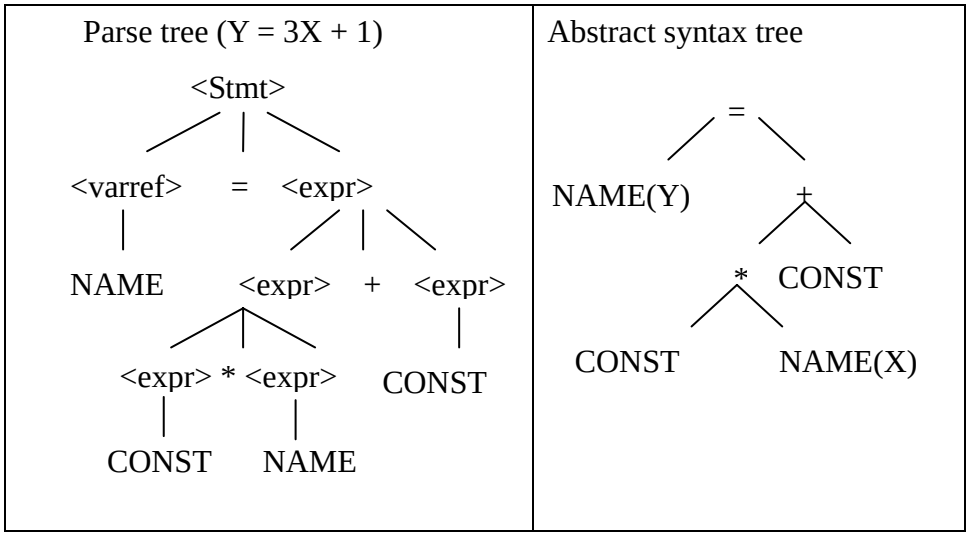

Figure 4.2. The difference between parse tree and abstract syntax tree.

In Figure 4.2 is an example of the difference between parse tree and AST, it represents statement “Y = 3X + 1”. In the left side of Figure 4.2 is parse tree of the statement, where words marked with “< >” are non-terminal symbols. The other symbols with upper case are tokens. In the right-hand side of Figure 4.2 is AST of the statement, non-terminal symbol are omitted, and initially only leaves with attributes.

In YACC, it allows you to insert semantic action anywhere for the user. We can use this action to construct abstract syntax tree representation of the input program. For example, In Figure 4.3 is YACC file in *rc-Promela* with semantic action to construct AST. The action code can refer to the values of the right-hand side symbols as $1, $2,…, and can set the value of the left-hand side by setting $$.

```
expr :
        '(' expr ')'          { $$ = $2; }
      | expr '+' expr         { $$ = nn(ZN, '+', $1, $3);}
      | expr '-' expr         { $$ = nn(ZN, '-', $1, $3); }
      | expr '*' expr         { $$ = nn(ZN, '*', $1, $3); }
      | NAME                  { $$ = $1; }
      | CONST                 { $$=nn(ZN,CONST,ZN,ZN);}
```

Figure 4.3. YACC file with semantic action for constructing AST.

In Figure 4.3 nn( ) function means that constructing a tree for the production rule. First parameter is symbol name of this production rule, second parameter is its type, third parameter is left leave of the tree, and last parameter is right leave of the tree.

We use this structure from bottom to up to construct an AST for source program written in *rc-Promela*. That is, an AST consists of a root node and subnodes. Each subnode may, in turn have its own subnodes. Each node in AST has a type field which specifies what kind of node it is and each node has left-hand side subnode and right-hand side subnode. Meanwhile, each node has a symbol field to specify its value. Constructing AST makes other analysis can be independent of the parser. Each time we need more analysis, we don’t need to call parser again to get data information; we just only need to traverse AST to get data to do analysis. We use AST to do data flow analysis to generate compact CCS state graphs.