Component-based LR Parsing

Xiaoqing Wu[a,*], Barrett R. Bryant[b], Jeff Gray[b], and Marjan Mernik[c]

[a] *Bank of America Corporation*
CH20, 4500 Park Granada, Calabasas, CA 91302
wuxi@cis.uab.edu
Phone: +1 818-225-3945
Fax: +1 205-934-5473

[b] *The University of Alabama at Birmingham*
Department of Computer and Information Sciences
115A Campbell Hall, 1300 University Boulevard,
Birmingham, Alabama 35294-1170, United States
{ bryant, gray}@cis.uab.edu
Phone: +1 205-934-2213
Fax: +1 205-934-5473

[c] *University of Maribor*
Faculty of Electrical Engineering and Computer Science
Smetanova ulica 17, 2000 Maribor, Slovenia
marjan.mernik@uni-mb.si
Phone: +386 2 220 7455
Fax: +386 2 220 7272

Abstract

A language implementation with proper compositionality enables a compiler developer to divide-and-conquer the complexity of building a large language by constructing a set of smaller languages. Ideally, these small language implementations should be independent of each other such that they can be designed, implemented and debugged individually, and later be reused in different applications (e.g., building domain-specific languages). However, the language composition offered by several existing parser generators resides at the grammar level, which means all the grammar modules need to be composed together and all corresponding ambiguities have to be resolved before generating a single parser for the language. This produces tight coupling between grammar modules, which harms information hiding and affects independent development of language

* Corresponding author.

features. To address this problem, we have developed a novel parsing algorithm that we call Component-based LR parsing (CLR), which provides code-level compositionality for language development by producing a separate parser for each grammar component. In addition to shift and reduce actions, the algorithm extends general LR parsing by introducing switch and return actions to empower the parsing action to jump from one parser to another. Our experimental evaluation demonstrates that CLR increases the comprehensibility, reusability, changeability and independent development ability of the language implementation. Moreover, the loose coupling among parser components enables CLR to describe grammars that contain LR parsing conflicts or require ambiguous token definitions, such as island grammars and embedded languages.

Keywords: Component-based software development, LR parsing, parser generator.

1 Introduction

Among various classical parsing algorithms, LR parsing [1] is a linear-time table-driven algorithm used in a large set of parser generation tools, such as YACC (Yet Another Compiler-Compiler) [2]. The theory of LR parsing has become a standard in compiler design courses and LR parsing tools are used widely in practice. However, in most LR parser generators such as YACC and CUP [3], there is little compositionality provided at the syntax level. Because large grammars may consist of several hundred or even thousands of productions (e.g., the GOLD grammar [4] of Cobol 85 is 2500 lines[1]), compiler developers are forced to place a significant amount of production rules inside one module and develop the parser as a whole. This usually yields poor comprehensibility, reduced changeability, limited reusability, and hampers independent development of the language implementation. These challenges are described as follows:

[1] Available at http://www.devincook.com/GOLDParser/grammars/index.htm

- Comprehensibility. Clearly, too many lines of productions intertwined together may result in poor readability. The lack of modularity means all variables (i.e., nonterminal and terminal symbols) have to share a single namespace, which leads to the generation of very verbose variable names. For instance, the IBM VS-COBOL grammar contains 1028 variable names [5]. Identifying what each symbol stands for in a global scope is difficult.
- Changeability and Reusability. A change to any variable name may propagate errors to hundreds of lines and modifications to any production may generate numerous conflicts with other productions across the language specification file. Furthermore, a language specification is difficult to reuse for embedded languages and languages with many variants or extensions. For example, COBOL has 300 dialects and several extensions for embedded applications such as SQL [6]. To change from one dialect to another, the whole grammar must be rewritten or extensively modified manually [7].
- Independent development. Because all the productions reside in one module, the grammar initially has to be written all together. Tangled relations between productions make it impossible for multiple developers to edit the same grammar concurrently, resulting in a lengthened development cycle for parser implementation. Because other parts of the language development depend heavily on the parsing results, the delay of the parser implementation impedes the overall development progress. In our previous development experience of extending a legacy language, it took a developer over six months to design a grammar with 600 nonterminals and implement the parser and tree structure, significantly limiting the progress of five other engineers working on semantic analysis and the editing environment.

The problems caused by a lack of modularity in a language specification can become more severe when tree construction and semantic analysis are involved. As a result of the poor modularity of the grammar, parse tree construction will be modularized poorly as well. If a semantic phase is implemented using the Visitor pattern [8], a single visitor class may contain a large number of visiting methods for all node classes (e.g., a JavaCC [9] + JJTree [10] sample program contains 85 visiting methods in a single unparse visitor[2]).

A possible solution to this problem is to separate the LR grammar into several related grammar modules, as employed in some compiler-compilers such as PPG (Polyglot Parser Generator) [11] and LISA (Language Implementation System based on Attribute grammars) [12]. These modules are composed together afterwards in a textual way to generate a single parser. However, syntax entities (nonterminals and terminals) are not constituents of the hidden part of a module. When importing a module for expressions, some nonterminals may clash with existing non-hidden nonterminals producing undesirable effects. Moreover, an LR grammar is not closed under composition. Adding an LR grammar module into an existing LR grammar may result in generating shift-reduce or reduce-reduce conflicts for the overall grammar. Hence, although modularizing the grammar specification helps for separation of concerns, the tight cohesion between modules may yield numerous problems when grammars are composed together in a textual way. This will be further discussed in Section 6 in comparison to related work.

To avoid the coupling between different modules, the ideal solution is to decompose a large language into several smaller language components (e.g., components for expressions and statements), with each component generating its own individual parser. Because the LR grammars of the various components do not co-exist in the same parser,

[2] Available at https://javacc.dev.java.net/source/browse/javacc/examples/VTransformer/UnparseVisitor.java.

composition at the code level does not generate any internal conflicts. In this way, a language could be implemented simply by a plug-and-play mechanism for parser components. The language developer will not have to worry about the dependency issues among different components. This approach is common in component-based programming [13] and invasive software composition [14] where components can be simple plug-ins that are integrated during a composition phase.

In order to implement the above mechanism, we have developed a new parsing algorithm, Component-based LR parsing (CLR), which provides code-level compositionality for language development by producing a separate parser for each grammar component. In addition to shift and reduce actions, the CLR algorithm extends general LR parsing by introducing switch and return actions to empower the parsing action to jump from one parser to another. CLR decreases the workload of constructing a large language parser into developing a number of smaller language parsers, where each decomposed parser component can be implemented and tested independently. This further benefits the development of the following semantic phases.

Experimental evaluation has demonstrated that CLR increases the comprehensibility, reusability, changeability and independent development ability of the syntax analysis phase. Moreover, the loose coupling among parser components enables CLR to describe grammars that contain LR parsing conflicts or require ambiguous token definitions, such as embedded languages. It is also possible to extend component-based parsing into other parsing algorithms, where a language can be implemented by multiple parsers each of which employs the most suitable parsing algorithm.

The paper is organized as follows. Section 2 details the definition of Component-based Context Free Grammar and introduces the CLR algorithm. Section 3 describes the difficulties and strategies of implementing the algorithm and presents a CLR parser generator. A case study of CLR parsing is introduced in Section 4, where sample CLR implementations of the Java language are presented. Section 5 demonstrates the evaluation of using CLR parsing and Section 6 discusses the difference between the compositionality employed in CLR and the modular grammar utilized in other language implementation practices. Future work and conclusions are presented in Sections 7 and 8, respectively.

2 Component-based Context Free Grammar and LR Parsing

In order to enable language implementation in a component-based manner, we have invented a new type of context free grammar called Component-based Context Free Grammar (CCFG), which is as expressive as conventional CFG, yet has a different representation format. In this section, a component-based LR parsing algorithm is presented to conduct parsing on top of CCFG specifications.

2.1 Component-based Context Free Grammar

Within a language specification, it is common to find a group of productions in the grammar that are tightly cohesive with one another. Inside the group of productions, most grammar symbols are not used by the outside productions, whereas one symbol may be constantly referred to by other groups. This symbol can be regarded as the point of entry to this group; in other words, the start grammar symbol of this sub-language. Figure 1 illustrates the grouping situation in the grammar of the Java Language Specification (JLS) [15]. The diagram is in the style of the Design Structure Matrix [16], which can be used to

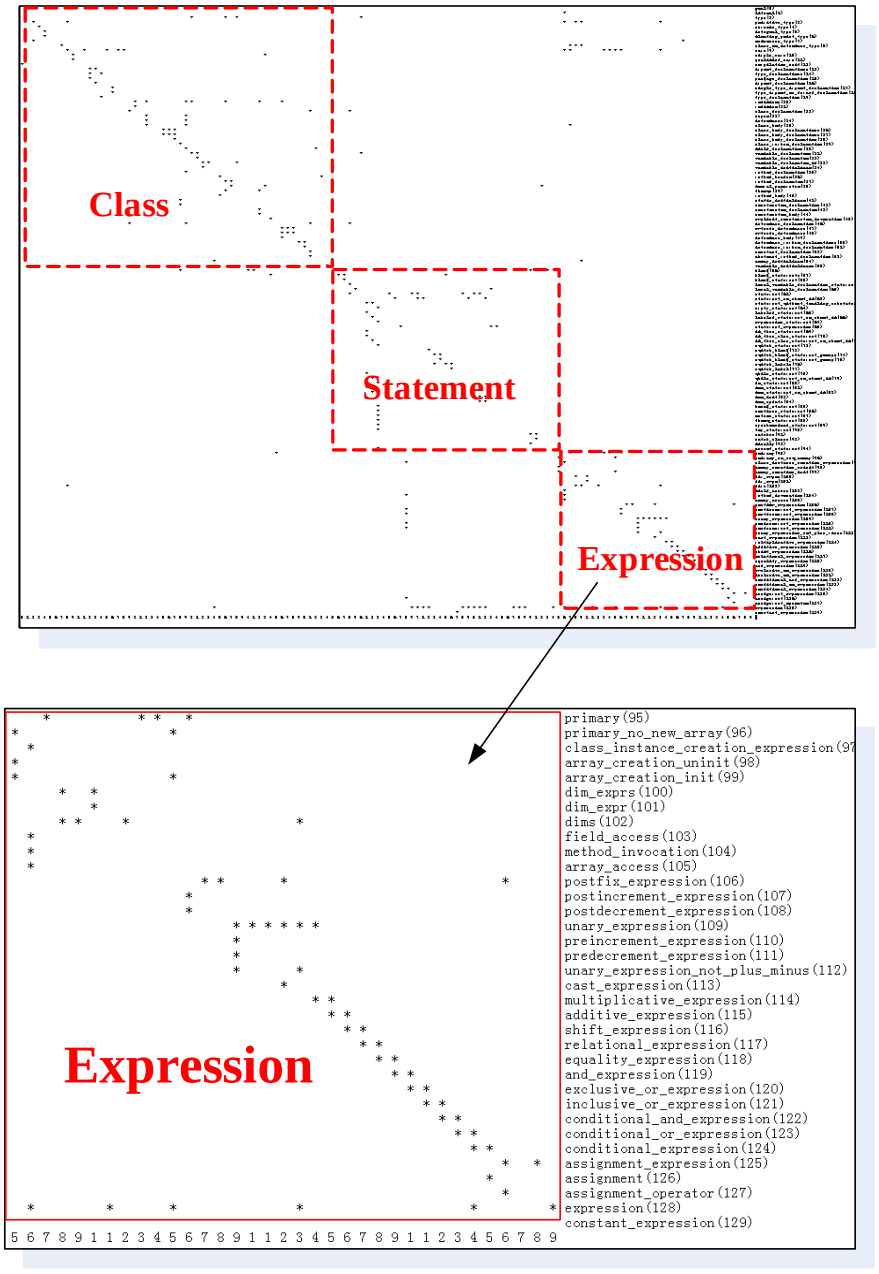


Figure 1: Illustration of production grouping in the JLS grammar

highlight the dependencies among modules. A star (*) in position (i, j) represents that there exists a production where the i^{th} nonterminal is the Left-Hand Side (LHS) symbol and the j^{th} nonterminal is one of the Right-Hand Side (RHS) symbols. The symbol names are attached to the indices along the right-most column. Notice that those expression-related

productions form a sub-matrix, inside which each symbol is mainly referred to by symbols inside the matrix, with one exception, the nonterminal `expression`. We call these expression-related productions a Grammar Component (GC) and treat the nonterminal `expression` as the start symbol of the component. The language produced from a grammar component is called a Language Component (LC), which can be reused to build other languages (e.g., the Expression component can be utilized to build the Statement component).

These groups of productions suggest the basic idea of language components. We formally define Component-based Context Free Grammar as follows:

Definition: A CCFG component G is a quintuple (N, T, C, P, S), where N is a set of nonterminal symbols, T is a set of terminal symbols, C is a set of symbols representing grammar components (called G's *child components*, which may include G itself) with $T \cap N = T \cap C = N \cap C = \varnothing$; the relation $P \subseteq N \times (N \cup T \cup C)^*$ is a finite set of production rules; S is the start symbol with $S \in N$. A production of the form $A \rightarrow \alpha$ means A derives α, where $A \in N$ and $\alpha \in (N \cup T \cup C)^*$. The Component-based Context-Free Language (CCFL) produced from grammar G is denoted by L(G). The union of CCFLs produced from G's child components is denoted by L(C). Let $\sigma \in (N \cup T \cup C \cup L(C))^*$, $\tau \in (N \cup T \cup C \cup L(C))^*$, and let $\gamma_1=\sigma B \tau$ and $\gamma_2=\sigma \beta \tau$. Then, γ_1 directly derives γ_2, denoted $\gamma_1 \Rightarrow \gamma_2$, if one of the following two conditions is met: 1) $B \rightarrow \beta$ is a production in P; 2) $B \in C$ and $\beta \in L(B)$. Furthermore, if there is a sequence of zero or more strings $\delta_1 \ldots \delta_n$ such that $\gamma_1 \Rightarrow \delta_1 \ldots \Rightarrow \delta_n \Rightarrow \gamma_2$, we say that γ_1 derives γ_2, denoted $\gamma_1 \Rightarrow^* \gamma_2$. If $S \Rightarrow^* \alpha$, we say that α is a *sentential form* of G. A sentential form with no nonterminal and component symbols is called a *sentence*. Therefore, L (G) is the set of all sentences that can be derived from the

start symbol S by sequential application of production rules and replacing grammar components with their corresponding languages (i.e., L (G) = {x| S $\Rightarrow$*x, x $\in$ (T$\cup$L(C))*}).

CCFG is the specification language for describing a grammar component. Inside a grammar component, the component preferably should have appropriate granularity. In other words, its derived sentences should be rich enough to be treated as a language, such as an expression language. This enables the decomposed unit to be developed and tested independently.

A single language can be described by a set of CCFG components, where the component containing the entry point to the language serves as the *root component*. The set of CCFG components must be closed (i.e., components being referenced in any other component must be defined within the set). Circular reference is allowed across CLR component definitions. The restriction is that the component set should not contain left-recursive references, namely, a component symbol should never be the first symbol of its own sentential forms; otherwise, it may cause an infinite loop in the generated parser. Notice that unlike LL grammars, this restriction is placed at the component level, not the production level. As CCFG components themselves are LR grammars, there is no problem to have left-recursive productions in CCFG.

2.2 Component-based LR Parsing

In CLR parsing, each grammar component in a CCFG set is generated as a separate parser. Each parser has a unique identifier that serves as the symbolic link when parsers are composed at the code level. These parsers are applied sequentially to parse different segments of the same input stream. The first invoked parser, called the *root parser*, is generated from the root component of the CCFG set. It invokes sub-parsers that will

recursively invoke other parsers as needed. Therefore, except for the root parser, each component parser will have a *parent parser* (which invoked this parser) and zero or more *child parsers* (which will be invoked by this parser). Those parsers having no child parsers are called *leaf parsers*. This makes all the parser instances compose in a tree-like structure when a program is parsed.

CLR is an extension to LR parsing. In addition to shift and reduce actions employed in LR parsing, the algorithm adds two new external actions: *return* and *switch*. A switch action transfers the parsing task from a parser to one of its child parsers and a return action returns it back. Although shift and reduce actions are determined by the current stack state and the next lookup (in case of LR (1)), whether to switch or return solely depends on the current state configuration, regardless of the next input. This is because the parser components are independent of each other, so the parent parser cannot pre-compute what is the first token of the child parser; therefore, it cannot make a decision upon external actions based on lookups. As the shift and reduce information is kept in a table, the switch and return conditions are stored in data structures called the *switch map* and *return map*. Given a particular state as the key, the switch map returns a small set of parser components (usually zero or one) that can be switched to at this state, and the return map simply returns a true or false value indicating if the parser is returnable. By consulting these two maps, the external actions are executed to switch the parsing task from one parser to another when internal actions are found ineffective.

Although there are no internal conflicts generated when LR parsers are composed, composition may generate new conflicts between internal actions and external actions, as well as conflicts among external actions themselves. To ensure the correct action is selected,

a set of rules are applied in the CLR parsing algorithm. By enforcing these rules, there is at most one action that can be selected at any particular parsing state. Hence, CLR always produces a deterministic parsing result for any set of CCFG components. These rules are described as follows:

- For conflicts between internal actions and external actions, the internal actions will take place first until an error state is encountered, when external actions will be checked.
- For conflicts between external actions (i.e., return-switch conflicts and switch-switch conflicts), switch actions have higher priority than return actions; the priorities among all possible switch actions are specified at the grammar level by the appearance order of related component symbols.
- Any incorrect selection of internal or external actions should be corrected by recovering the stack and executing other external actions until one of them ultimately succeeds.

Figure 2 demonstrates the flowchart of CLR parsing from one parser's perspective. It shows that CLR parsing extends regular LR parsing by introducing parser dispatching actions. The extended part is invoked when the executing parser reaches a point that there are no internal actions available for the next input token (i.e., an error state). The detailed logic used in parser dispatching is presented below as an algorithm.

Algorithm: Component parsing

Input: The unparsed part of the input program and the stack of the executing parser.

Output: A flag to indicate whether the general LR parsing should be resumed or terminated, or simply tagged an error.

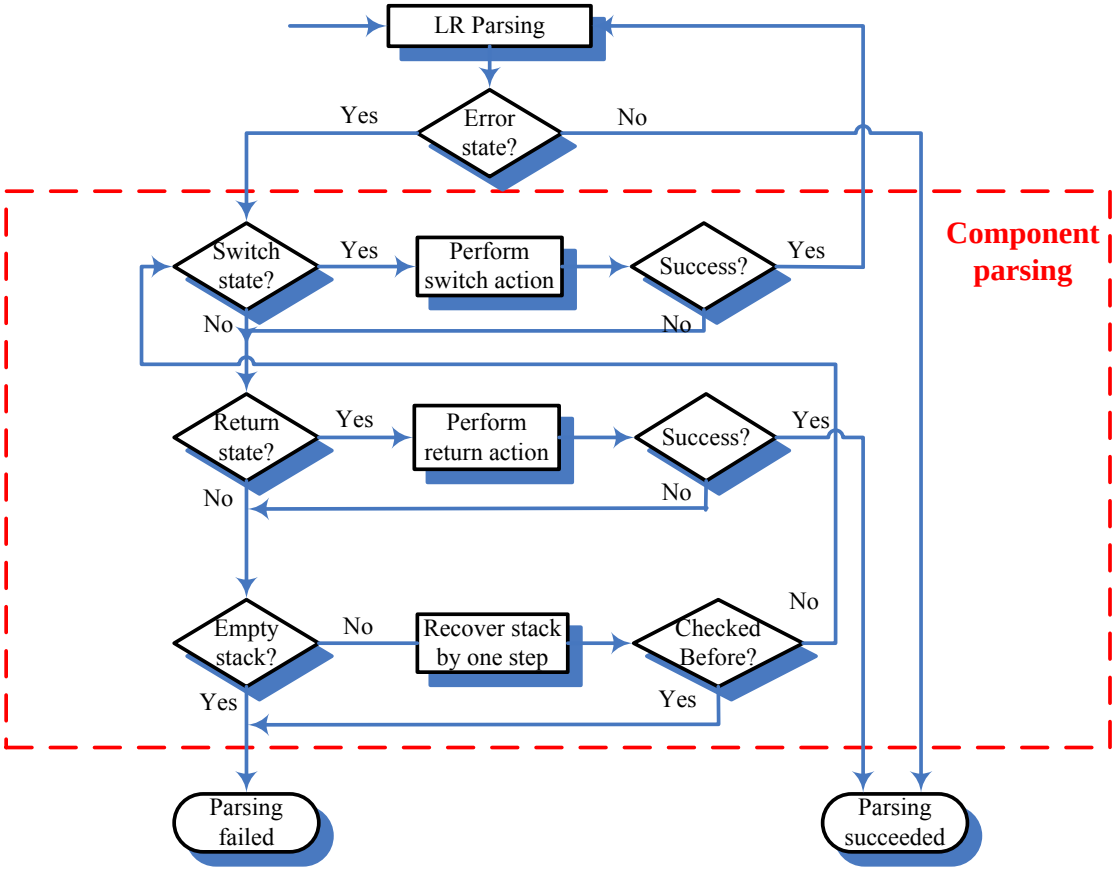


Figure 2: Flowchart of component-based LR parsing

Description: Once the component parsing is invoked, the switch condition will be checked first. If the available components returned by the switch map are not empty, the switchable components will be invoked one after another as child parsers to take over the parsing task, until one of them successfully parses the next segment of the stream and returns. The component parsing in this case will return a continue signal to resume the LR parsing of the current parser. Otherwise, if no child parsers can parse the remaining stream, or if there was no switch condition available, the return condition will be examined. Consequently, if the parser is in a return state, the parsing task will be returned to this

parser's parent parser. If the return action is successful, it can be claimed that this component has finished its parsing job and should be terminated. Otherwise, as all external actions fail, the current parsing stack will be recovered to the previous stage and external conditions will be re-checked. The same procedure will be repeated until a stage is reached where the external actions have already been examined before, or the parsing stack simply becomes empty, in which case the algorithm claims that this component's parsing fails by throwing an error flag. The pseudo code presented in Figure 3 provides a more formal description of this algorithm.

```
Pseudo code:
flag component_parse(program, stack):
  repeat
    state := stack.top()
    if switch_map (state) ≠ ∅
      ∀ component ∈ switch_map (state)
      if component.lr_parse(program) == true
        record stack configuration
        return continue_flag
      end if
    end if
    if return_map (state) == true
      if return action success
        return termination_flag
      end if
    end if
    if stack ≠ ∅
      recover stack by one step
    else
        return error_flag
    end if
  until reach the stack configuration when
      last switch action happened
  return error_flag
```

Figure 3: Pseudo code of component parsing

In the remainder of this section, a simple language illustrates how the algorithm works. Suppose there is a language X that only takes three strings: “aa”, “ab”, and “ac”. By using CCFG, this language is built on top of two language components Y and Z that each contains one string “b” and “ac”, respectively. The composition of language X is illustrated by the CCFG in Figure 4.

Suppose there are four programs to parse, namely, “aa”, “ab”, “ac” and “ad”. The parser X (together with sub-parsers Y and Z) should be able to parse the first three programs successfully and reject the last one. Figure 5 illustrates the action sequences (internal and external) and the stack evolution for all of the four input instances, with each parsing procedure detailed as follows:

Case 1: For program “aa”, parser X is able to parse the program by two shift actions and one reduce action, without the help of other parser components. In this case, it is simply LR parsing.

Case 2: For program “ab”, parser X will first shift the input character ‘a’ onto the stack. However, when parser X encounters input ‘b’, there are no internal actions available in its parsing table because ‘b’ is not in X’s terminal set. Therefore, parser X enters an error stage. At this time, component parsing is invoked to check if there are any external actions available. The switch map returns component Y for this state and parser Y will be tried. After parser Y shifts the second character ‘b’ onto the stack, an error state is reached again. Since parser Y is a stand-alone parser, there is no child parser Y can switch to. However, from the return action map, it is seen that Y is currently in a valid return stage. Therefore, parser Y will reduce its token into S_y and return it to parser X as G_y. The parser X then reduces aG_y into S_x and indicates a successful parse.

G_x = ({S_x}, {**a**}, {G_y, G_z}, {$S_x \rightarrow$ **a a** | **a** G_y | G_z }, S_x)
G_y = ({S_y}, {**b**}, ∅, {$S_y \rightarrow$ **b**}, S_y)
G_z = ({S_z}, {**a**, **c**}, ∅, {$S_z \rightarrow$ **a c**}, S_z)

Figure 4: CCFG specification for language X

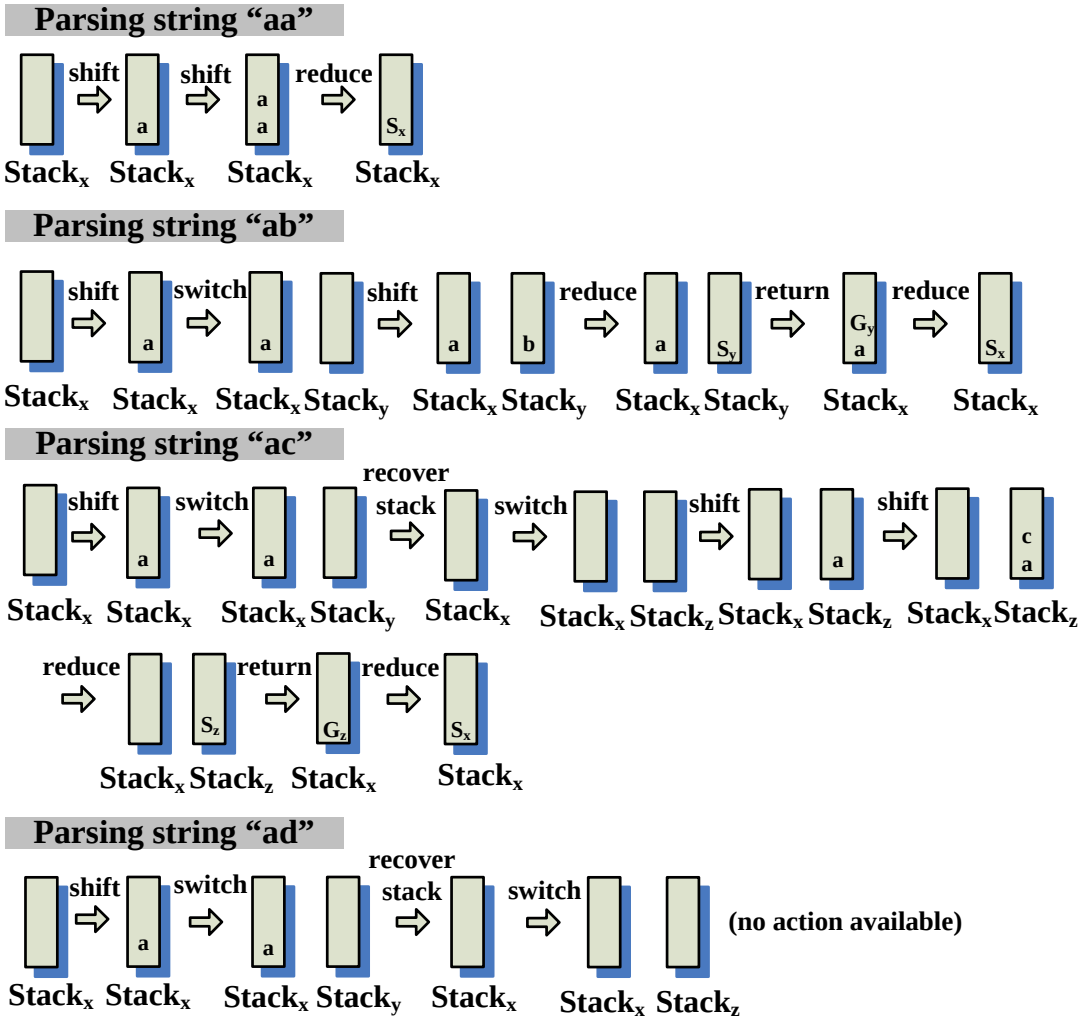


Figure 5: Sample action sequence and stack evolution in LR parsing

Case 3: For string "ac", similarly, after parser X shifts 'a', it enters an error state and attempts to switch to parser Y. However, this time parser Y also cannot parse the next input token 'c'. Because there is no switch or return state available and its parsing stack is empty,

parser Y immediately fails and is discarded. As parser X is the root parser, there is no return action for it. Therefore, now the only choice for parser X is to recover the stack by one step (i.e., pop the 'a' character from the stack) and recheck the external actions. The switch map returns component Z, so parsing is switched to parser Z. Similar to parser Y recognizing 'b' in case 2, parser Z recognizes "ac" and returns to parser X before parser X reduces G_Z into S_X and claims the parsing successful.

Case 4: For program "ad", similar to case 2, parser Y fails in parsing the 'd' character and parser X's stack is recovered. Then, parser Z is able to parse the first character 'a', but it fails at the 'd' character as well. As parser Z has no component symbols and it has not reached a return point yet, there is no external action that can be used, so parser Z fails. At this point, because there are no more external actions available and the stack is empty, parser X totally fails the parsing. Because it is the root parser, a syntax error is declared at this point.

3 Implementation of CLR Parsing

The CLR parsing algorithm and its associated parser generator have been fully implemented in Java by extending and modifying the parser and lexer generators CUP and JLex [17], respectively. The external actions have been added and the priority rules for resolving conflicts (i.e., internal - external action conflicts, and conflicts among external actions themselves) are properly enforced. For each CCFG component, a parser and lexer are generated as a Java package.

Figure 6 provides a sample CLR specification for the example languages mentioned in Section 2.2. It can be seen that a general CLR component typically contains four parts: component name, import declaration, production definitions and terminal definitions, which

```
Component X
// Component name.
language x;
// Imported components.
import y as y_language;
import z as z_language;
// Product definitions.
x ::= A A | A y_language
| z_language;
// Terminal definitions.
A   'a';
```

```
Component Y
language y;
y ::= B;
B    'b';
Component Z
language z;
// Exported symbols.
export z1;
z ::= A z1;
z1 ::= C;
A     'a';
C     'c';
```

Figure 6: The CLR specification for languages X, Y and Z

correspond to symbols S, C, N & P, T in the CCFG definition, respectively. The component name serves as both the default start symbol of the component and the language identifier used for composition. The identifier following the keyword "as" (e.g., `y_language`) provides an alias of the imported component, which can be directly used inside the current component. The order of the import declarations indicates the trial sequence when there are switch-switch conflicts, in which case the most frequently occurring sub-language should have higher priority. In order to make each specification module partially reusable, additional start symbols can be declared using an `export` declaration. Each exported symbol will generate a separate parser that uses it as the start symbol. In this case, the component name and a start symbol together represent the unique identification of a sub-language of the component, which can be imported by other components as child parsers. For example, if a component only wants to reuse the 'c' character of language `z`, it can achieve that by importing id `z.z1` (the grammar of the `Z` component is tailored for

demonstration purposes). Consequently, from the specification in Figure 6, a total of four parsers and three lexers are generated in three Java packages (parser `z` and `z.z1` share the same lexer and package), among which parser `x` will use parser `y` and `z` to conduct its parsing functionality.

In the following two sub-sections, the difficulties in implementing this system will be first discussed, followed by the corresponding solutions.

3.1 Implementation Challenges

To use the CLR parsing algorithm correctly, there are several questions to consider. First, *how are the external action maps constructed*? As described in Section 2.2, to transfer the parsing task from one parser to another once an error state occurs, the switch map and return map have to be consulted. The question is how to determine if a particular state is a switch or return state and where to store such information. Particularly, it is difficult to validate a return state because an LR parser normally identifies the termination of the parsing by shifting the final token EOF, which is less likely reached by a component parser.

Second, *how is it determined if a component's parsing is truly successful*? In LR (1) parsing, a valid shift or reduce action is secured by the next lookup. However, in CLR parsing, since each parser has its own parsing table, it is not easy to conduct lookups for the external actions. Moreover, because LR (1) grammars are not closed under composition, the overall "composed grammar" in CLR might not be a valid LR (1) grammar, which means a single lookup cannot always solve the problem. It is possible that a segment of the input stream can be parsed by more than one component, but only one of them is the ultimate correct choice. Therefore, a return action of a particular parser can only be treated

as a temporary success of its parsing. Further processing is needed to guarantee it is indeed the correct choice.

Third, *how is the parsing stream synchronized*? In CLR, each parser has its own lexer and definition of tokens, so synchronization is needed to enable multiple lexers to work on the same source stream sequentially. Because each LR (1) parser requires its lexer to look ahead one token before an action is executed, the stream's state has to be reset after the parsing is switched. Furthermore, there may be incorrect actions that are executed. To retry other actions, the lexical stream should be reset to the previous stage along with the parser's stack.

Fourth, *how are semantic actions incorporated with syntax specification*? Normally, an LR parser generator is able to embed semantic actions with each production, which will be executed once the production is reduced. However, because CLR parsing involves backtracking, which means there could be false reductions, the side-effects caused by executing related semantic actions must be revoked in a certain way.

3.2 Implementation Strategy

To solve the obstacles mentioned in the previous section, the following techniques have been employed. First, external actions are implemented by transforming them into internal actions. In the implementation of CLR parsing, instead of providing separate data structures to host the external action maps, switch and return information is integrated into the original shift-reduce parsing table. For the switch action, this is achieved by introducing a *dummy terminal symbol* (i.e., a token that does not exist in the input parse stream) to represent a child component[3]. Whenever it is needed to consult the *switch map*, the

[3] This idea is borrowed from the error-recovery mechanism utilized in most LR parsers, where a special nonterminal "error" is utilized as the dummy token.

algorithm returns those dummy symbols under which there are shift actions available (i.e., assuming the next input token is a dummy symbol). If it is not empty, the component symbols that are able to be switched are recorded. For the return condition, we simply return whether the current state has shift or reduce actions available under the EOF token. In summary, to check an external action, we examine the current stack state against predefined dummy tokens instead of the next real input token. For example, in the implementation of language X, Y is represented as a dummy terminal symbol. When an 'a' character is shifted onto the stack of X and external actions are checked, it will be shown that language Y can be switched to regardless of the next input symbol. After language Y recognizes a 'b' character, it will reach a return state even if the next input is not an EOF token.

Second, in order to make sure an incorrect switch or return action can be corrected eventually, backtracking is employed in CLR. For switch actions, as illustrated in the left part of Figure 7, the corresponding child parsers are created one after another to try parsing the remaining stream until one of them succeeds. Each time a switch fails, the parsing stack will be recovered to make sure the next child parser begins its attempted match from the same point. If none of the child parsers can successfully parse the stream, the whole switch parsing fails. For return actions, two types of components are defined: *perfect components* and *regular components*. A perfect component is one that has no ambiguity with its surroundings (i.e., using this component is the only way to interpret any sentence in that language). This type of component usually comes with special guards, such as statement (i.e., ending with '`;`' or '`}`'), or class definition (i.e., beginning with … `class` and ending with '`}`') in the Java language. For these components, a valid return action can guarantee

the previous parsing with this component parser was the correct choice. In the current CLR specification, a perfect component is indicated by prefacing the keyword `perfect` to the language id or the start symbol of an export declaration, as in `perfect language foo` or `export perfect foo`. Regular components are those components whose sentences may fully or partially overlap with other component sentences, which are usually the small components that do not have clear guards. Parsing with such components hinders determining if the return action is correct.

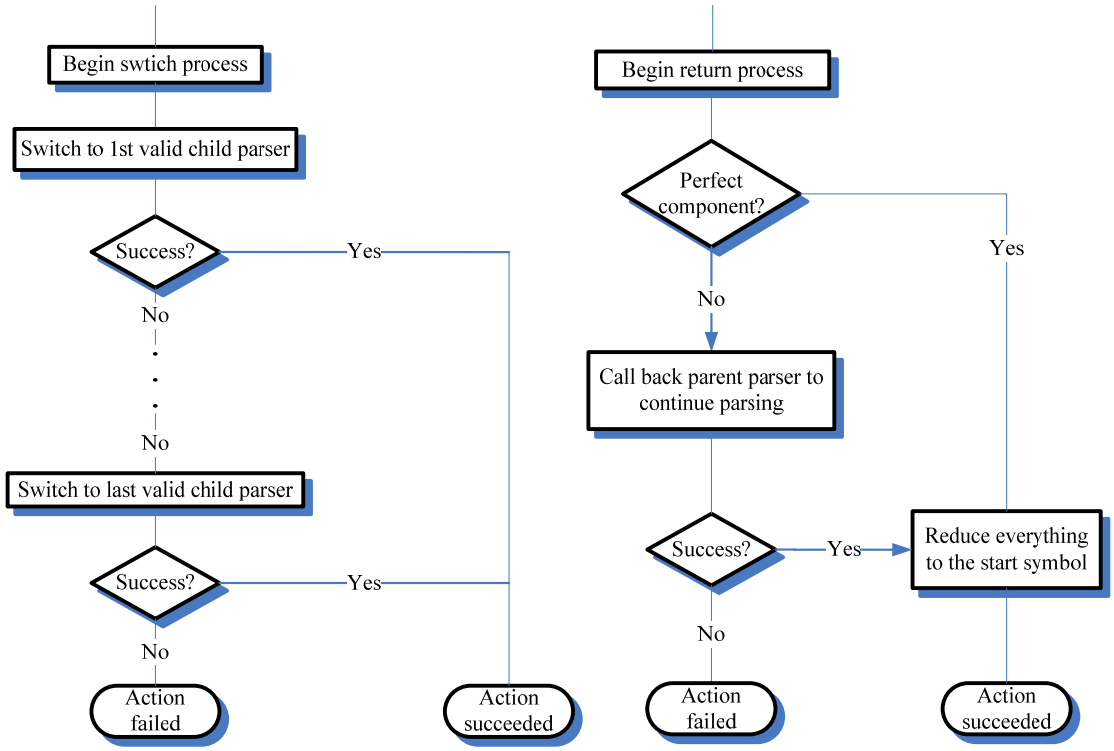


Figure 7: The implementation detail of the switch and return actions

As described in the right side of Figure 7, for a perfect component, the return actions will simply iteratively reduce all the symbols in the stack to the start symbol and then return the control to the parent parser to continue its parsing. The parser object is

immediately thrown away after the return. However, for a regular component, instead of returning to the parent parser directly, the child parser will *call-back* the parent's parsing method to resume the parse. In this case, the success of this return action is actually determined by the success of the parent parser's subsequent parsing result. If the parent parser fails at a certain place, this parser's stack has to be backtracked to a place where another external action is available. Meanwhile, the parent parser's stack will be recovered to the stage where the return action has not been executed. For example, in a programming language such as Java, if the grammars of the cast expression and the parenthesized expression are specified in two separate components, the prefix `(MyType)` of a cast expression (e.g., `(MyType) var`) might be successfully parsed first by an expression parser as a parenthesized expression. However, after it returns, its parent parser will not be able to continue its parsing because an expression cannot be followed by a variable (e.g., `var`). Therefore, the return action will be recovered such that a different parsing path can be tried.

Third, to solve the synchronization problem, a solo stream reader equipped with mark and reset functions is shared by all the lexers to harmonize and recover the input stream processing. Each time the parser is switched or returned, the static reader will be reset so that it always points to the unrecognized input stream. Supplying separate lexers for each parser is a significant contribution of CLR parsing (detailed in Section 5.2). However, in the case that multiple parsers do want to share the same lexical definition, we simply replace the keyword `import` with `importNL` (i.e., meaning no lexer importation) and then the parser's own lexer will be used directly by its child parsers.

Fourth, to address the problem related to semantic actions, the CLR specification is designed to be purely declarative. Given a CLR grammar component, a strongly-typed parse tree structure is automatically generated along with the parser. The tree construction actions are embedded within the generated parser. After one child parser returns or calls back its parent parser, its parse tree is silently plugged into the parent parser's parse tree. Because all the tree nodes are stored in the parsing stack, any tree node obtained from a false return or reduce will be popped eventually during the parse stack backtracking. Consequently, the tree structure will no longer hold a reference to this node as its parent node is already popped (i.e., last in first out). Eventually, all the parse tree components will result in a single parse tree. Further semantic analysis will be made through the generated tree classes instead of embedding them directly inside the parser. This avoids the need to undo semantic actions once a false reduction is detected.

4 CLR Parsing Case Study

To validate the benefits of using CLR, we have utilized CCFG to rewrite the Java language grammar in a component-based manner. The overall grammar follows the JLS introduced in [15] with extension to JDK 1.4, which contains 156 nonterminals and 359 productions. Depending on the scale of each component, eleven versions of the Java grammar have been developed for experimentation, with each version composed of one to eleven grammar components, where the one-component grammar is equivalent to the traditional non-modularized grammar. Development experience shows that the version that has four components (i.e., `Java language`, `Class`, `Statement` and `Expression`) offers high-quality compositionality. Other versions that contain more than

four components are provided in this section for exploration purposes, with the eleven-component implementation pushing the language modularity concept to an extreme.

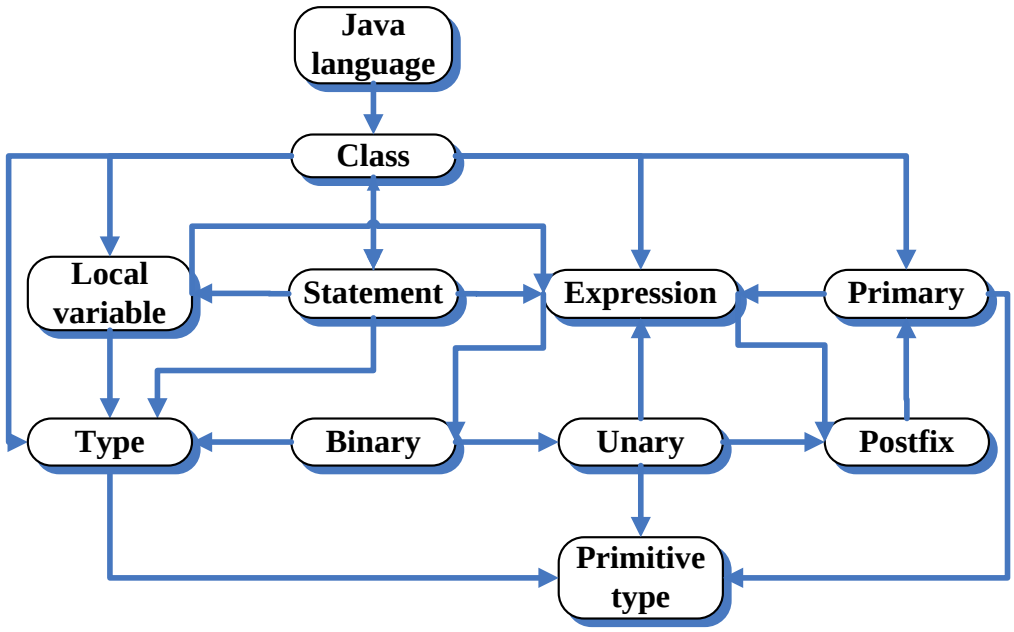


Figure 8: CLR components of JLS (11 component version)

Figure 8 demonstrates the version that contains 11 language components, where an arrow denotes the end component is a child component of the start component. Initially, a `Java language` component is created to describe an instance of the Java program (i.e., a compilation unit). A compilation unit is composed of a sequence of package, import and class declarations. The definition of package and import declarations is short enough to be specified in the `Java` component directly. However, production rules related to a class declaration tend to be lengthy and closely coupled, so a class declaration is extracted as an individual language for compositional development. Applying the same strategy recursively, productions related to variable declarations, statements and expressions are factored out as grammar components, among which the `Expression` component is further decomposed

into `Binary`, `Unary`, `Primary` and `Postfix` components. Type-related productions are also encapsulated as `Type` and `Primitive type` components, where the latter is a child of the former component.

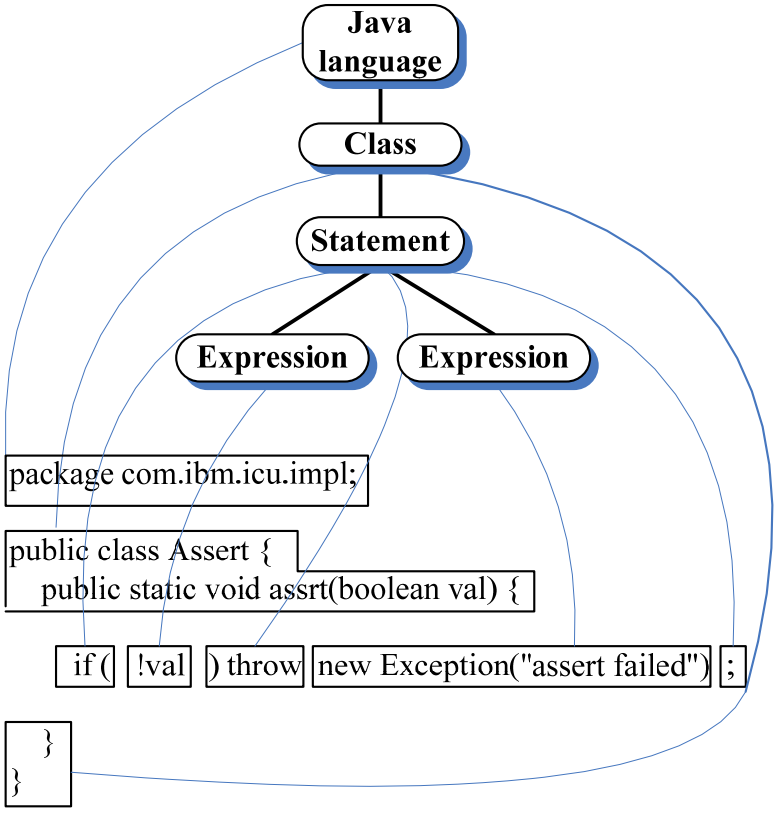


Figure 9: Parsing illustration of a sample Java program using the CLR JLS implementation (4 component version).

Figure 9 provides an illustration on how the four component implementation parses the text of a sample Java program[4]. A total of five parser instances are created sequentially to parse the input stream, which is segmented into eight corresponding sections. Compared to the single parser implementation generated from 800 lines of grammar specification,

[4] This is adapted from a Java program of Eclipse SDK-3.2 (available at http://bugs.icu-project.org/trac/browser/icu4j/trunk/src/com/ibm/icu/impl/Assert.java).

each parser component focuses on a relative small scope, which is more modular. For example, the statement parser is only used to parse the keywords and separators, whereas the parsing of concrete expressions is left to the expression parsers.

5 Evaluation of CLR Parsing

Based on our experience with applying CLR to the JLS, together with various other CLR practices described below, we have observed positive results in using CLR parsing technology. This section evaluates the implementation of the CLR parsing algorithm against regular LR parsing using three criteria: compatibility with software engineering principles, language description ability and performance. A discussion of CLR's drawbacks and limitations is also provided.

5.1 Software Engineering Benefits

CLR parsing provides benefits toward changeability, reusability and independent development of a language implementation. Moreover, the CLR specification is more comprehensible than conventional LR grammars. These benefits are outlined as follows:

- **Changeability**. CLR parsing is naturally equipped with information hiding. A change to a component is isolated inside the component to avoid propagation. Furthermore, since each component generates a separate parser, any change inside a grammar component will only require the recompilation of itself. This is a major difference with other modular grammar development approaches such as PPG and SDF, where any change inside a component will require recompilation of all the related modules. This is discussed further in Section 6.

- **Reusability**. As each component is an independent entity, it is free to compose and reuse different components to build new languages. For example, to extend the Java language to SQLJ [18], we simply need to add an `SQL` parser as the child parser of the `Statement` parser to integrate SQL statements as legal statements in Java. Moreover, using CLR parsing, after a parser for a language is developed, a set of sub-language parsers are also ready to reuse. Specifically, they might be reusable to build other Domain-Specific Languages (DSLs), especially those designed using the piggyback pattern (i.e., building a new language by partially reusing existing languages) [19]. For example, the `Binary` component in the Java specification, which has more than 50 lines of CFG productions, can be reused to build the binary expression part of C or C++ with little change. The Java statement parser can be reused to implement compiler-compilers such as CUP, where Java statements are embedded in the BNF specifications as semantic actions. The sub-components are also reusable in development of Integrated Development Environments (IDEs), where various entities such as the outline view and debugger require parsers for class definitions and expressions.
- **Independent development.** In CLR, each grammar component has its own namespace and start symbol. Dependencies among components are handled at the code level instead of the grammar level. Left recursion is the only thing that needs to be concerned with globally. In the current implementation, the generated Java class of each parser maintains a set of symbols that indicate the "first set" of the corresponding grammar component, and a left recursion check is made when a new parser is loaded. Notice that this analysis is conducted dynamically at parsing time.

Therefore, the global analysis is achieved beyond the grammar level. As a result, the loose coupling in CLR enables each language component to be individually developed, debugged and tested against different portions of a source program. Particularly, to debug a component whose child components are not developed yet, an empty component can be used to represent the child components. Therefore, multiple developers can work on the grammar of the same language concurrently and produce separate parser components. This reduces the development cycle of syntax analysis and benefits other compiler phases that follow.

- **Comprehensibility**. By decomposing the large grammar into small grammar components, the number of intertwined symbols and productions inside a single component are reduced, resulting in a specification that is easy to understand and maintain. In the original JLS specification, explicit long names have to be created to capture the necessary information for improved comprehensibility. In CLR, because the role of each symbol is defined by the component name and nonterminal name pair, the name of each symbol tends to be short and precise. For example, in the `statement` component, the nonterminal `local_variable_declaration_statement`, can be shortened as `local_variable_declaration`. Moreover, inside a grammar component all the symbols and productions are related to certain language entities (e.g., expressions, statements), which makes the definition more cohesive.

5.2 Language Description Ability

In terms of language description abilities, a CLR grammar is more expressive than a regular LR grammar due to its backtracking mechanism. Essentially, the expressiveness of

CLR is a superset of LR and a subset of arbitrary context-free grammars. It can describe any deterministic context-free language. Additionally, by enabling multiple lexers to tokenize the same stream, CLR parsing is able to solve the problems caused by ambiguous tokens, which normally occur in conventional LR parsing.

- **Expressive power.** Although the CLR approach generates LR parsers, the CLR grammar is richer than a pure LR grammar. In principle, the backtracking logic employed in CLR parsing is equivalent to infinite lookahead [20]. By extracting the shift symbol and its associated productions into a separate component, CLR's backtracking can resolve the traditional shift-reduce or reduce-reduce conflicts in LR parsers. Therefore, it is able to produce a deterministic result as long as the grammar is not inherently ambiguous. An example of using this advantage is the processing of *island grammars* [21, 22], where the structure of interesting parts of a language (i.e., the *islands*) is described in detail and the remaining part (i.e., the *water*) is simply mentioned as character streams. This type of language normally is difficult to handle by regular parsing methods due to the conflicts between the island and water. Figure 10 provides an island grammar example adapted from [22] that is used to extract email addresses from an arbitrary text document. The overall grammar is difficult to specify using LR grammars because it is ambiguous (e.g., terminal `@` can be either recognized as an email symbol or one of the water characters). However, in CLR, by extracting the water definition as an external component, the ambiguity is naturally resolved. As in Figure 10, the symbol `email` is defined by internal productions and `water` is defined as an imported component. This gives priority to the email concern. Consequently, the `island` parser will

always first parse the stream as email addresses until there is no internal action available, when the `water` parser will be used to consume a single token and return.

- **Ambiguous tokens.** Allowing an independent lexer for each parser permits the same token to be interpreted differently inside various component scopes. This is very helpful for processing embedded languages where the host language has different reserved words from the guest language. For example, SQLJ could be implemented by a Java parser that hosts an SQL parser, with each having its own lexer to tokenize input programs. Therefore, a token like `count` will be naturally recognized as a variable in the Java scope and as a keyword in the SQL scope. If we use a conventional communication mechanism between a single parser and single lexer, all the reserved keywords will be prohibited, regardless of the scope. Similarly, multiple-lexers can also benefit those languages that have no reserved keywords, such as PL/I, where a statement like `IF IF = THEN THEN IF = THEN;` is legal. In a proper CLR implementation where expressions and statements are described in two separate components, `IF` and `THEN` are only treated as keywords in the statement component, but not in the expression component. Consequently, only the first `IF` and second `THEN` will be parsed as statement keywords, but others are parsed as identifiers in the expression. Similar usage can also be applied to a stand alone language that does have reserved words. For instance, an annoying fact for C++ developers is that most C++ parsers surprisingly fail to parse a template reference like `vector<vector<int>>` although it visually appears to be legal syntax. The problem is caused by the fact that the two consecutive right angle brackets are falsely recognized by the C++ lexer as a right-

shift token. To avoid this problem, the above expression has to be rewritten as `vector<vector<int> >`, where one or more empty spaces are inserted between the two brackets. This problem also can be solved by the CLR easily if type-related productions have their own component, as the right-shift token ">>"is only preserved in the `Expression` component (or some of its child components) as operators. In the `Type` component, which should be used to parse the above clause, it will be recognized simply as two template reference guards.

Notice that CLR is designed to handle unambiguous grammars. If there is actually more than one correct choice, it means the grammar is ambiguous. In that case, CLR will only find the first correct pass and terminate.

```
language island;
import water as other;
// Syntax definition
island ::= unit | unit island;
unit ::= email | other;
email ::= SEGMENT AT hostname;
hostname ::= SEGMENT DOT SEGMENT | SEGMENT DOT hostname;
//Lexical definition
DOT ".";
AT "@"|"(at)"|"at";
SEGMENT [A-Za-z][A-Za-z0-9\-\_]*;

language water;
water ::= ANY;
ANY .; // '.' means any character
```

Figure 10: CLR specification for the island grammar example

5.3 Performance Measurement

In this paper, CLR's parsing performance is not directly compared with other parsing tools, due to the fact that the development language (e.g., C vs. Java) and the input

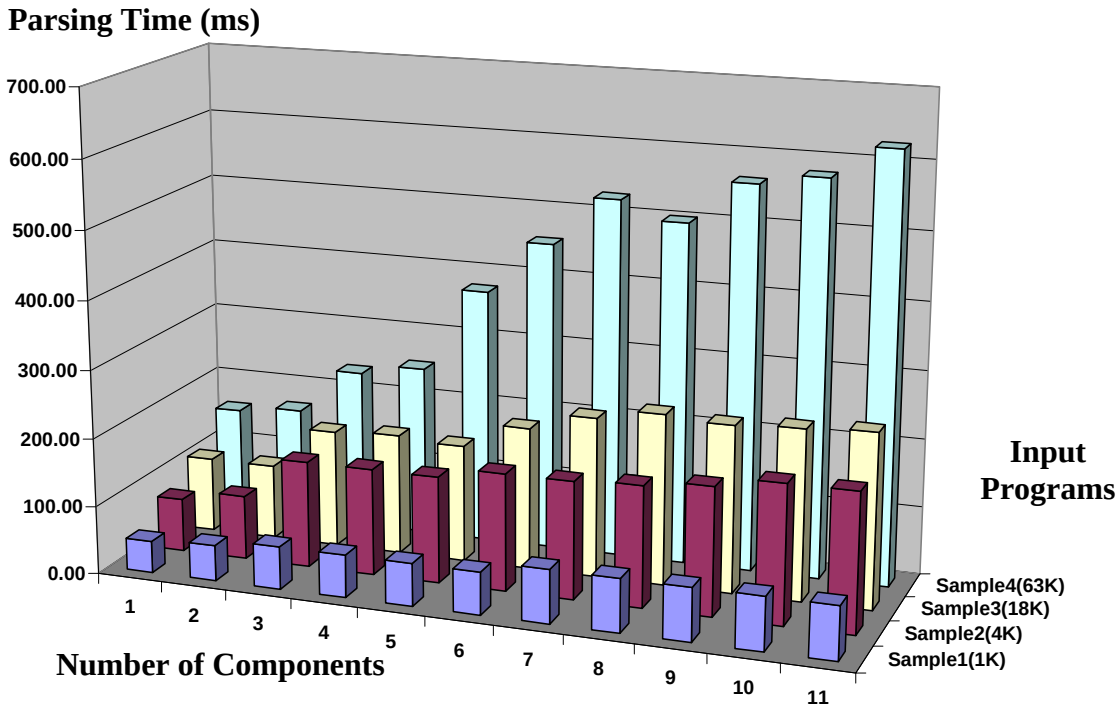
grammar can affect the parsing speed significantly. Instead, the performance of CLR is measured with its LALR counterpart and itself with different numbers of components. In general, CLR parsing is slower than LR parsing if there is more than one parser component (CLR is equivalent to LR parsing if there is only one component).

We have tested the parsing speed of the 11 versions of the JLS implementation (the single component version is the regular LALR parser) by using them to parse four randomly selected Java programs from the Eclipse SDK of size 1k, 4k, 18k and 63k[5]. Figure 11 illustrates the parsing performance. The table at the bottom of Figure 11 shows the parsing time and associated percentage increase when compared to the previous number of components. It is clear that as the number of components increase from 1 to 11, the parsing time increases in a sub-linear manner (156ms to 625ms in the case of Sample4). The table provides the actual value of the parsing time and the increasing ratio when a new component is extracted and added to the implementation. Notice that by adjusting the import order of child components in each component, a different set of testing results can be produced. However, in this case the difference is very minor because there are few switch-switch conflicts occurring in these implementations.

There are two reasons attributed to the time increase. First, when a parser is switched or returned, the overhead of the function call is not ignorable, especially when the component is small and parser switching happens frequently. For example, in Figure 12 it is shown that there are 3885 switch actions and 2648 return actions executed when parsing Sample4 using the 11 component version. These thousands of extra calls are unavoidable

[5] Available at http://bugs.icu-project.org/trac/browser/icu4j/trunk/src/com/ibm/icu/impl. The four programs are named Assert.java, CalendarCache.java, Trie.java and CalendarAstronomer.java, respectively.

because they are needed by the backtracking mechanism, and hence are attributed to the extra overhead of CLR parsing over regular LR parsing.



	1	2	3	4	5	6	7	8	9	10	11
Sample1 (1k)	47	52	62	62	62	62	78	78	78	78	78
		11%	19%	0%	0%	0%	26%	0%	0%	0%	0%
Sample2 (4K)	78	94	156	156	156	172	172	177	187	203	203
		20%	67%	0%	0%	10%	0%	3%	6%	8%	0%
Sample3 (18K)	109	109	172	177	172	209	234	250	245	250	255
		0%	57%	3%	-3%	21%	12%	7%	-2%	2%	2%
Sample4 (63K)	156	166	234	250	375	453	526	500	563	578	625
		6%	41%	7%	50%	21%	16%	-5%	13%	3%	8%

Figure 11: Parsing speed comparison among 11 versions of CLR implementation of JLS

Another factor is that in LR parsing, the program is always parsed deterministically, with the development cost of making the grammar LR, whereas CLR employs backtracking

to resolve conflicts across parser components, which means the same program piece could be tried by multiple parsers until one succeeds. As seen in Figure 12, the number of switches is always greater or equal to the number of returns. Because a success switch is always followed by a corresponding return, the gap actually indicates the number of failed switch actions. As some internal actions may be executed after the parser is switched, failed switches consume a considerable amount of parsing overhead.

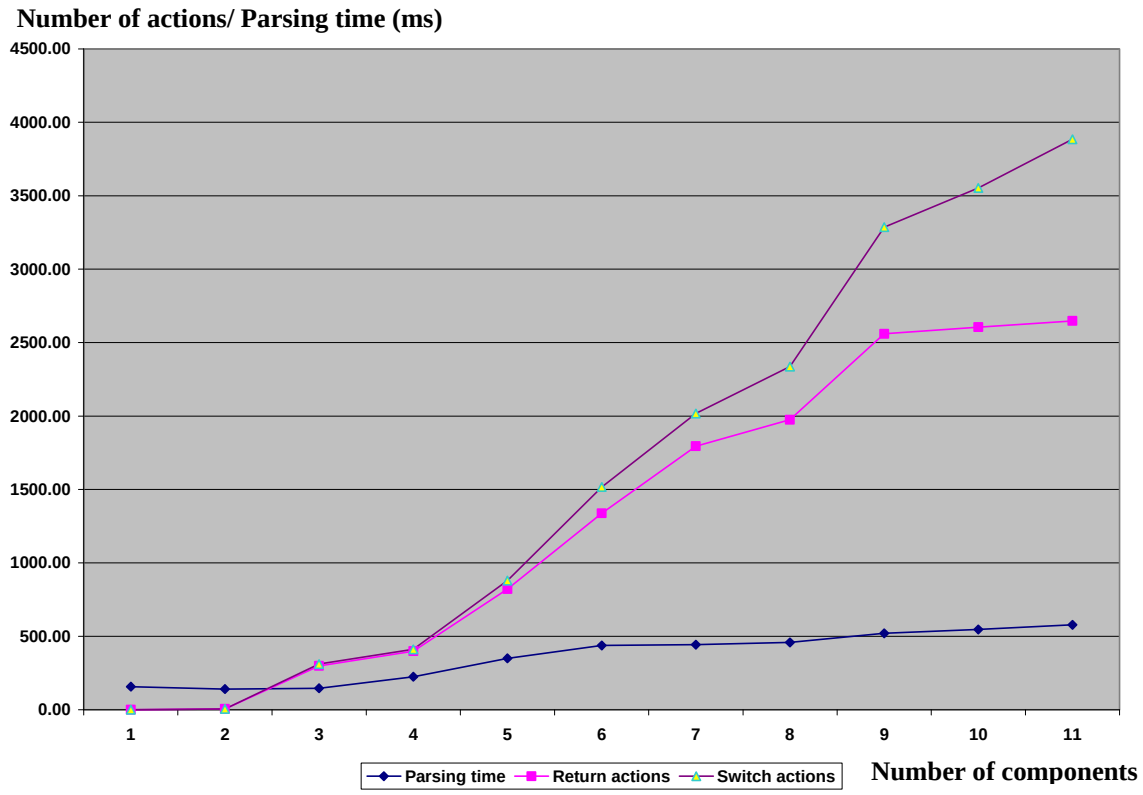


Figure 12: The number of external actions used to parse Sample4

However, because each component parser uses LR parsing internally, most parts of CLR parsing are still deterministic. If the number of components is small (e.g., ≤ 4), the dominant overhead of CLR parsing is still tokenization and internal actions. In Figure 12, the corresponding parsing time of each implementation is also provided to reflect the

relationship between parsing time and number of external actions. It shows that the increase in the number of external actions is not proportional to the increase in parsing time (e.g., from 1 component to 11 components, the number of external actions increases by several orders of magnitude, but parsing time only increases 2-4 times). Moreover, it is not always the case that the parsing time goes up as the number of components increases. For instance, for input program Sample4, the eight component parser is actually 5% faster than the seven component version. To explain this phenomenon, as the number of components increases, the size of the parsing table decreases, which optimizes the table lookup time. If the savings on the table lookup overcome the overhead on extra function calls and backtracking, the overall parsing speed will increase, otherwise it will decrease.

5.4 Discussion

The current CLR implementation has two limitations caused by the backtracking used in CLR. In general, backtracking has the risk of exponential parsing time [7], so it is possible to develop a set of CLR components that are configured in a poor manner such that it causes exponential behavior. However, such risk is much less than in general LR parsing:

- In CLR, backtracking only occurs when there is an external action available at the state and the action conflicts with an internal action or another external action. There is no backtracking for shift-reduce or reduce-reduce conflicts as found in regular LR backtracking. In practice, each component only imports a limited number of other components, each of which is only switchable under a limited number of states. It is rare that there are both internal and external actions available for a certain state. Particularly, parsing by *leaf parsers* (i.e., the components that

have no imported components) is guaranteed to be linear bounded. Even if they are tried multiple times, the complexity will still be polynomial, not exponential.

- Employing perfect components can significantly prune the paths during backtracking, because once a "perfect language" is recognized, the parsing with this parser will not need to be backtracked (i.e., a perfect component guarantees the interpretation was right).

Overall, CLR gives the developer control over the whole language grammar by resolving most conflicts manually inside a small scope, so that backtracking is seldom needed. In contrast, it is very easy to write a grammar that causes exponential parsing in regular backtracking LR parsers [23].

The other problem caused by backtracking is error-recovery. In YACC-like LR parsers, the error recovery process is invoked immediately after an error state occurs. However, in CLR, an error state does not mean the program actually contains invalid syntax until all external actions have been tried. This breaks the error-recovery mechanism of regular LR parsing. The same problem also exists in other indeterminate parsing technologies, such as Generalized LR (GLR) and backtrack LR. CLR does not currently support error-recovery. A possible solution will be discussed in Section 7.

6 Related work

Modularity in syntax analysis is not a new problem. There have been a number of attempts to provide grammars with modular constructs, which are supported by parser generators based on various parsing algorithms.

The first category of related LR parser generators is represented by LISA [12] and PPG [11]. Both tools allow a new grammar module to use different operators to override, inherit and extend productions from an existing module. However, despite their flexibility in incremental development, these grammar modules cannot be composed directly. The conflicts across modules have to be resolved manually either by modifying the grammar or assigning priority to tokens [7]. Another LR parser generator that supports modular development is BtYacc (Backtracking Yacc [24]). It uses backtracking to resolve shift-reduce or reduce-reduce conflicts. Once a conflict is encountered, BtYacc will try all possible paths until one of them succeeds. Therefore, BtYacc's modules are compositional. On the other hand, SDF (Syntax Definition Formalism) [25] and the DMS (Design Maintenance System) [26] use GLR parsing, where grammars are allowed to contain LR conflicts. When shift-reduce or reduce-reduce conflicts occur, a GLR parser forks the parse stack and tries all the possible paths in parallel. Consequently, their modules can be freely composed, provided that the same nonterminal does not coexist in multiple modules. To some extent, the backtracking technology can be treated as a Depth First Search (DFS) solution and GLR can be treated as Breadth First Search (BFS). DFS finds the first successful path and returns, but BFS will retrieve all the successful ones. Therefore, GLR is well-known for handling ambiguous grammars, but backtracking does not serve that purpose. Due to its BFS implementation nature and the overhead of maintaining a graph-structured stack, SDF parsers are typically slower by a factor of ten or more than their LALR counterparts on non-ambiguous input [20].

Beyond the LR grammar family, there are also other frameworks that support modular language development. Koskimies' lazy recursive descent parsing [27, 28], which

takes each nonterminal and its RHS productions as a module, was among the first ones to introduce modular parsing techniques. Particularly, its "syntax-directed" scanning technique allows interpreting the same token with multiple meanings in different contexts, which enjoys the same benefits as the independent lexer described in this paper. Recent parser generators and transformation systems that support modular development include ANTLR (ANother Tool for Language Recognition) [29] and TXL (Turing eXtender Language) [30], which are based on LL (k) parsing, and Rats! [31], which is built on recent research on Parsing Expression Grammars (PEGs) [32]. ANTLR has to manually resolve the possible ambiguities generated from composition using predicates, but TXL and Rats! do not need to resolve any ambiguity because their production rules are inherently ordered.

However, merely enabling textual modularity at the grammar level is not strong enough to address the problems mentioned in Section 1. The nature of module inclusion of these tools is pure text copying and the main goal is to support incremental language development [12], rather than decompose the development complexity. For each language, the parser is still implemented as a singleton that forces the remaining phases to follow the same track. In comparison to these parsing technologies, the significance of CLR parsing comes from the fact that the composition occurs among parsers instead of grammars. Figure 13 demonstrates the difference between modular parser generation and compositional parser generation. Parser-level composition is superior in several aspects:

- The coupling between components resides at the parser level instead of the grammar level. Any update to a particular component does not require a recomposition of all the components in the package and regeneration of a large parse table as in PPG, BtYacc or SDF. For an implementation of a complex language with a number of

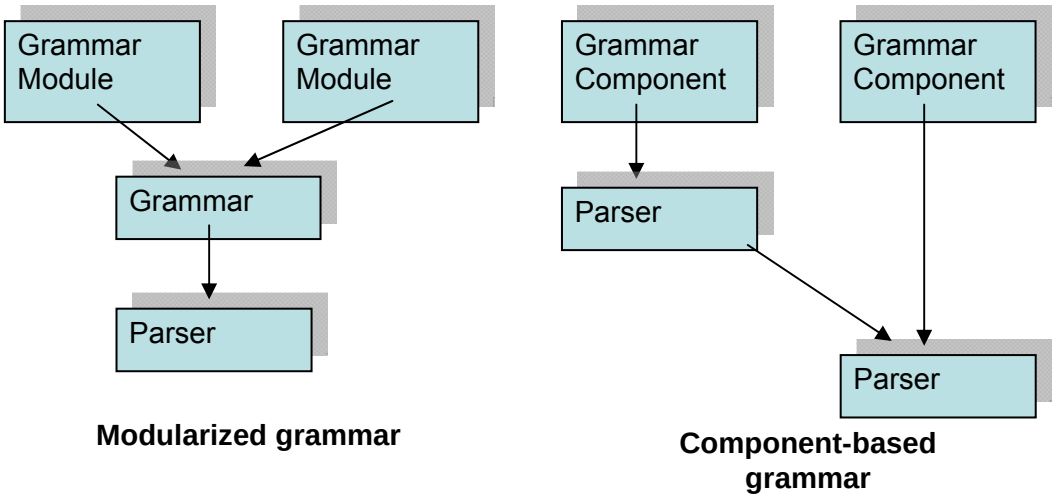


Figure 13: Difference between modular parser generation and compositional parser generation

modules, composition and generation overhead is not negligible. For example, for SDF modules of the Java language, module composition time alone took 8.2 seconds[6].

- The decomposition of the single parser leads to the decomposition of its parsing table and parser code size. A smaller table sometimes speeds up the table lookup time, as illustrated in Section 5.3. A smaller parser can avoid certain problems caused by large code size. For example, our attempt to build a one component version of the JLS parser with a concrete syntax tree construction cannot be compiled by javac 1.5 due to a "code too large" problem as the CUP generated parser contains a 7000-line switch statement.
- As we have discussed in Section 3.2, each parser component can maintain its own lexer, which provides support for ambiguous tokens. To solve the same problem,

[6] The tests were conducted on a machine with the following specifications: AMD Athlon(tm) XP 2800+ (cache size: 512 KB), 1 GB RAM, Linux (Fedora Core 2). 8.2 is the average time of 10 rounds of tests.

SDF uses scannerless parsing, which essentially takes each character as a token and leaves the GLR parser to recognize the terminal symbols, at which level ambiguity is allowed. However, this implementation essentially assigns a task originally handled by a Deterministic Finite Automata (DFA) to a more expensive Pushdown Automata (PDA), which definitely hurts performance due to the overhead on stack operations. Some lexer generators such as Lex [2] and JLex [17] allow the same lexeme to represent different tokens in different states. However, to switch lexer states, the developer is required to manually trigger it as lexer actions, which sometimes is difficult to use as the grammar context is not known [29]. Notably, ANTLR allows the same parser to work with multiple lexers. The programmer still needs to invoke the action manually, but the action resides at the grammar level. The lexer switching process is controlled by a token stream multiplexor, which has the same role as the static reader employed in the CLR implementation.

- Parser composition actually happens at the Java bytecode level, which means within the CLR family, a component parser is still reusable by a third party if the source grammar is not released, even if the Java source code is not available.

Delegating Compiler Objects (DCOs) [33] is among the few approaches that are targeted to build a large language implementation with smaller compiler components. Although DCO does not provide sophisticated handling of parsing conflicts and it is unclear what type of grammar it can process, the framework's philosophy that a language implementation should not only have functional decomposition (e.g., lexer, parser, code generator) but also allow structural decomposition is totally in-line with the fundamental ideas described in this paper. Another previous effort on code-level parser composition is

from the functional language community. Notably, parser combinators allow runtime manipulation of parsers [34]. Specifically, a parser is coded directly using a functional language such as Haskell [35] and it can be passed around as a parameter. In [36], implementing parser combinators in Java is introduced. The essence of this strategy is to handcraft a parser instead of using declarative grammar specifications, which has its obvious drawbacks. Mapping it manually to an imperative language makes it more verbose and inefficient [34]. As a result, its performance is not comparable to table-driven LR parsers [36].

For parser generators in which the lexer is accessible in parser actions (e.g., ANTLR), it is possible to invoke a sub-parser by manually inserting semantic actions in the syntax definition, but this comes with a limitation and a development cost. The language module must have clear guards (e.g., JavaDoc has "/**" and "*/"). The grammar must be divided in such a way that the open guard is specified in the parent parser to indicate the switch and the closing guard is used as the end token of the child parser to indicate the return. Moreover, besides the parser switching actions, the parsing result from the sub-parser also has to be connected to the main parser manually.

7 Future Work

There is still room available for Component-based LR parsing to be improved. The problems caused by backtracking should be addressed in the future implementation. First, certain restrictions can be invented to constrain the backtracking behavior, such as setting a maximum backtracking length or limiting the number of non-perfect components in use. These restrictions should be designed in such a way that it will not significantly affect the

language description ability of CLR grammars. For error handling, a possible strategy is to try all possibilities and record the path that goes the furthest. The path in this case includes all the involved parser components and their corresponding parsing stack configurations at the time the furthest point is reached. When all possibilities fail, the error recovery methods of all parsers in the longest path will be invoked in a bottom-up manner until one of them handles the error, as in the Chain of Responsibility design pattern [8].

In component-based language development, it is often desirable to reuse only a small number of nonterminal definitions from other modules. As the scale of such reuse is quite small, it would be too costly to produce a stand-alone parser for that purpose. Moreover, a module may need to be tailored before being reused, which is difficult to achieve by direct parser composition. Therefore, the syntax specification could be improved by borrowing some strategies from incremental development approaches to copy and modify production rules from other grammar components. Besides using the keyword `import` to declare the component reuse at the code level, another keyword `include` can be used to indicate module reuse at the textual level. These two types of declarations can be seen as language constructs borrowed from general-purpose programming languages, namely, `import` from Java and `include` from C++. New types of operators also should be supported to inherit, extend, overwrite or modify the included modules. Notice that module inclusion is still different from pure module-based development. Once a change is made inside a component, although multiple included modules might need to be read, there is just one sub-parser regenerated, with other component parsers remaining intact. In other parser generators that support modular development, the change would require regeneration of the whole parser.

Although the focus of this paper is CLR, the component-based parsing schema is not limited to LR parsers. The independent nature of each component allows the idea to be extended to any other parsing algorithm, such as LL, GLR, combination of LL and LR parsers (e.g., [37]), and even handcrafted parsers. Specifically, the requirements for a parser to become a valid component are the following:

- It must have the return action defined to be an eligible child component;
- It must have the switch action defined to be an eligible parent component;
- If a component is neither a perfect component nor a leaf component, or it contains internal-external conflicts, the parser must support backtracking logic as return or switch actions could be wrong.

These components can coordinate together by sharing a global interface. This enables each language construct to be implemented by the most suitable approach. For example, if one part of the language is not LR but other parts are, we could build a hand-coded parser for that particular component and compose it with the main parser built by a LR parser generator.

8 Conclusion

In this paper, CLR parsing is presented as a novel parsing technology that is designed to support component-based development in language implementations. It adds the regular LR parsing switch and return actions to dispatch parsing tasks from one parser to another. CLR parsing decreases the complexity of building a large language by constructing a set of smaller language parsers from grammar components, which at the same time strengthens software engineering principles (e.g., changeability, independent

development and reusability). CLR is more expressive than regular LR as it employs backtracking and multiple lexers to resolve the conflicts at the syntax and lexical levels. Compared to other parser generation tools that support modular grammars, the CLR parser generator produces loosely coupled parser components with more manageable code size and code-level reusability. Notably, each parser component can be individually developed and tested, which helps to reduce the development cycle of the overall language implementation. CLR is an ideal platform to develop programming languages and DSLs with complex and hybrid language constructs. Its usage can be extended further after combining with other parsing technologies, such as LL and handcrafted parsers.

Therefore, CLR successfully solves the modularity problem in language implementation from a structure perspective. Besides the Java language, several DSLs such as Google Query Language and Robot language [38] have also been developed using the CLR parsing algorithm. These sample applications further support CLR's benefits in practical usage. The source code of the CLR parser generator and some of its sample specifications are available at the project web page http://www.cis.uab.edu/softcom/cde.

Reference

[1] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607-639, 1965.

[2] J. R. Levine. *lex & yacc*. O'Reilly, 1992.

[3] *CUP: Parser Generator for Java*.

http://www.cs.princeton.edu/~appel/modern/java/CUP/

[4] D. Cook. *Gold parser builder*. Available at: www.devincook.com/goldparser

[5] A. Johnstone, E. Scott, and G. Economopoulos. The grammar tool box: a case study comparing GLR parsing algorithms. *Electronic Notes in Theoretical Computer Science.* 110: 97-113, 2004.

[6] R. Lämmel and C. Verhoef. Cracking the 500-Language Problem. *IEEE Software,* 18(6): 78 - 88, November 2001.

[7] M. v. d. Brand, M. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings of the International Workshop on Program Comprehension*, pp. 108-117, June 1998.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] *JavaCC: Java Compiler Compiler*, Sun Microsystems, Inc. https://javacc.dev.java.net/

[10] *Introduction to JJTree.* http://www.j-paine.org/jjtree.html

[11] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, pp. 138-152, April 2003.

[12] M. Mernik and V. Žumer. Incremental programming language development. *Computer Languages, Systems and Structures*, 31:1–16, April 2005.

[13] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*, 2nd edition. ACM Press, 2002.

[14] U. Aßmann. *Invasive Software Composition*. Springer-Verlag, 2003.

[15] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*, Addison-Wesley, 1996.

[16] C. Y. Baldwin and K. B. Clark. *Design Rules: the Power of Modularity Volume 1*. MIT Press, 1999.

[17] *JLex: Java Lexical Analyzer Generator*. http://www.cs.princeton.edu/~appel/modern/java/JLex/

[18] J. Melton and A. Eisenberg. *Understanding SQL and Java Together: A Guide to SQLJ, JDBC, and Related Technologies*. Morgan-Kaufmann, 2000.

[19] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4), 316–344, December 2005.

[20] S. McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. In *Proceedings of the 13th International Conference on Compiler Construction*, pp. 73–88, March 2004.

[21] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the IEEE 8th Working Conference on Reverse Engineering*, pp. 13–22, October 2001.

[22] J. Visser and J. Scheerder, *A Quick Introduction to SDF*, CWI, April 2000. ftp://ftp.stratego-language.org/pub/stratego/docs/sdfintro.pdf

[23] A. Johnstone, E. Scott, and G. Economopoulos. Generalised parsing: some costs. In *Proceedings of the 13th International Conference on Compiler Construction*, pp. 89–103, March 2004.

[24] *BtYacc: BackTracking Yacc* http://www.siber.com/btyacc/

[25] E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, September 1997.

[26] I. Baxter, P. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *Proceedings of the International Conference on Software Engineering*, pp. 625-634, May 2004.

[27] K. Koskimies. Techniques for modular language implementation. *Acta Cybernetica*, 9(3): 193-209, November 1990.

[28] K. Koskimies. Lazy recursive descent parsing for modular language implementation. *Software—Practice and Experience*, 20(9): 749-772, August 1990.

[29] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(*k*) parser generator. *Software—Practice and Experience*, 25(7):789–810, July 1995.

[30] R. Cordy. TXL - A Language for programming language tools and applications. *Electronic Notes in Theoretical Computer Science*. 110:3–31, 2004.

[31] R. Grimm. Better extensibility through modular syntax. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 38-51, June 2006.

[32] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, pp. 111–122, January 2004.

[33] J. Bosch. Delegating compiler objects: modularity and reusability in language engineering. *Nordic Journal of Computing*, 4(1): 66-92, March 1997.

[34] G. Hutton and E. Meijer. Monadic parsing in Haskell. *Journal of Functional Programming* 8(4):437-444, July 1998

[35] S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.

[36] A. Dijkstra and D. S. Swierstra. Lazy functional parser combinators in Java. In *Proceedings of the 1st Workshop on Multiparadigm Programming with Object-Oriented Languages*, pp. 11–42, May 2001.

[37] B. Slivnik and B. Vilfan. Producing the left parse during bottom-up parsing. *Information Processing Letters*, 96:220-224, December 2005.

[38] X. Wu. *Conquer Compiler Complexity: Component-Based Language Implementation with Object-Oriented Syntax and Aspect-Oriented Semantics.* VDM Verlag, 2008.

Biography

Xiaoqing Wu is a Quantitative Analyst at Bank of America Corporation. He received his Ph. D. degree in computer science from the University of Alabama at Birmingham in 2007. His research interests include compiler design, programming languages, aspect-oriented programming and software engineering. He is a member of ACM.

Barrett R. Bryant is Professor and Associate Chair of Computer and Information Sciences at the University of Alabama at Birmingham. His research interests include theory and implementation of programming languages, formal specification, and component-based software engineering. He is a member of ACM and EAPLS and a Senior Member of IEEE.

Jeff Gray is an Associate Professor of Computer and Information Sciences at the University of Alabama at Birmingham. His research interests include model-driven engineering,

aspect-oriented software development, and generative programming. He is a member of ACM and a Senior Member of IEEE.

Marjan Mernik is an Associate Professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. His research interests include programming languages, compilers, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.