**120747**

**Wanjugu Annsonia Njoki**

**Compiler Construction Assignment**

1. **How do you solve the problem of ambiguous grammar? Are there any well-established techniques/methods of solving this problem?**

A grammar is said to be ambiguous if for a given string more than one left or right derivation or the parse tree can be developed, implying that a given string can be derived using two or more parse trees.

There are a couple of ways to solve the ambiguity problem and they are;
**Implementing the precedence constraints**: In this method, the following rules are applied to remove ambiguity. First, the level where the production rule in the grammar is present is used to define the priority of the operator used between the symbols in the rule, second, the higher level of production rule means the priority of the operator used between the symbols is low, and third, the lower level of production rule means the priority of the operator used between the symbols is high.
**Associativity Constraints**: In this method, the following rules are applied to remove ambiguity. Use the left recursion in a production rule if the operator used is left-associative and the right recursion in a production rule if the operator used is right-associative.
In summation; associativity constraint is implemented using the rules below;

- If the operator is left associative, induce left recursion in its production.
- If the operator is right associative, induce right recursion in its production.

For example, consider the following ambiguous grammar:

A → A + A / A * / A. A / a / b

Remove the ambiguity by applying the steps below:
The operators used in the grammar are +, and *, and the operands used are a and b.
The precedence order can be defined as (a, b) having higher precedence than “*”, “*” having higher precedence than “.”, and “.” having higher precedence than “+”.
The grammar can be converted to unambiguous grammar as shown below:

A → A + B / B

B → B. C / C

C → C*/ D

D → a / b

2. **Research and write about bottom-up parsing (BUP). After highlighting the general concept of BUP, investigate the following parsers: LR, SLR and LALR**

A syntax analyser, also known as a parser, determines whether a given source program follows the rules implied by a context free grammar. If it is satisfied, the parser generates the programs parse tree, otherwise it generates an error message. There has been an increase in the number of parser generators such as Bison, Elkhound and SDF that support general context-free grammars, compared to LL, SLR and LALR parsers that handle ambiguities by dynamically merging the resulting parse trees, during parsing.

Whereas top-down parsers begin with the first symbol, bottom-up parsers begin with the first token of the input. Their mode of operation may appear to be far more intuitive than top-down parsing. They repeatedly match symbols from the input with the strings on the right-hand sides of production rules, replacing the matched strings with the corresponding left-hand sides. This process is repeated until only the first symbol remains. The rightmost derivation defines the correct matching order for the bottom-up parser. Bottom-up parsers must reverse the rightmost derivation.

Bottom-up parsing includes LR parsing that reads input text from left to right without pausing and produces a reverse rightmost derivation (LR Parser, 2021). This LR parser works by performing a series of shift and reduce steps. A Shift step moves one symbol forward in the input stream. That shifted symbol is transformed into a new single-node parse tree. A Reduce step applies a finished grammar rule to some of the most recent parse trees, merging them into a single tree with a new root symbol (LR parser - Javatpoint, 2021).

LR Parser is a shift-reduce parser that creates use of deterministic finite automata, identifying the set of all viable prefixes by reading the stack from bottom to top. It decides what handle, if any, is available.

A viable prefix of a right sequential form is that prefix that includes a handle, but no symbol to the right of the handle. Thus, if a finite state machine that identifies viable prefixes of the right sentential form is generated, it can guide the handle selection in the shift-reduce parser.

There are three types of LR Parsers which are as follows −

- SimpleLR Parser (SLR): It is very easy to implement but it fails to produce a table for some classes of grammars.

- Canonical LR Parser (CLR): It is the most powerful and works on large classes of grammars.
- Look Ahead LR Parser (LALR): It is intermediate in power between SLR and CLR

The deterministic nature of the LR parser contributes significantly to its efficiency. To avoid guesswork, the LR parser frequently considers the next scanned symbol before deciding what to do with previously scanned symbols. The lexical scanner processes one or more symbols before the parser. The lookahead symbols serve as the parsing decision's 'right-hand context' (Tutorialspoint , 2022).

SLR stands for simple LR. It is the smallest grammar class, with the fewest number of states. SLR is very simple to implement and is similar to LR parsing. The only difference between SLR and LR (0) parsers is that there is a possibility of 'shift reduced' conflict in the LR (0) parsing table because we are entering'reduce' corresponding to all terminal states. We can solve this problem by entering'reduce' in the terminating state, which corresponds to FOLLOW of the LHS of production. This is known as SLR (1) item collection (GeeksforGeeeks, 2022).

SLR parsing is LR (0) parsing, but with a different reduce rule:

```
For each edge (X: (I, J))
if X is terminal, put shift J at (I, X)
        if I contain Aà -> α. where Aà ->α. has rule number n
                for each terminal x in Follow(A), put reduce n at (I, x)
```

SLR generators compute the lookahead using a simple approximation method based on the grammar, ignoring the specifics of individual parser states and transitions (Javatpoint., 2022). This disregards the context of the current parser state. If a nonterminal symbol S is used in multiple places in the grammar, SLR treats those places as a group rather than individually. The SLR generator computes Follow(S), the set of all terminal symbols that can immediately follow any occurrence of S. Each reduction to S in the parse table has Follow(S) as its LR (1) lookahead set. Such follow sets are also used by generators for LL top-down parsers. When using follow sets, an SLR grammar has no shift/reduce or reduce/reduce conflicts (Tutorialspoint, 2022).

LALR parser was invented as a variant of the LR(k), as a space-efficient parser as its underlying automaton is the LR (0) machine, regardless of the value of k. By reducing the number of states, the LALR(k) helps reduce the space requirement of an LR(k) parser while maintaining the speed advantage: this is accomplished by merging states. Most commercially available parser generators only handle the case of k=1 (Looking ahead 1 symbol).

A LALR (1) parser can be built by first creating an LR (0) parser and then merging states with the same LR (0) items. The merge, however, has no effect on the lookahead values or symbols that determine the reduction value's position in the parsing table. This improves the parser's ability to detect errors earlier, making it more powerful and efficient.

The LALR(k) set of languages is a proper subset of the LR(k) set of languages. In practice, however, LALR(k) grammars are used because they are powerful enough to accommodate most programming languages. Based on an input grammar, the parser-generator yacc (and bison) generates an LALR parser. Furthermore, yacc (and bison) provide facilities for resolving shift-reduce conflicts based on operator declared precedence.

## References

Aaby, A. A. (2004). Compiler Construction using Flex and Bison. Walla Walla College.

Brabrand, Claus & Giegerich, Robert & Møller, Anders. (2007). Analyzing Ambiguity of Context-Free Grammars. Science of Computer Programming. 75. 176-191. 10.1016/j.scico.2009.11.002.

Bruce S. N. Cheung. 1995. Ambiguity in context-free grammars. In Proceedings of the 1995 ACM symposium on Applied computing (SAC '95). Association for Computing Machinery, New York, NY, USA, 272–276. https://doi.org/10.1145/315891.315991

CHRISTIANSEN, H. (2005). CHR grammars. Theory and Practice of Logic Programming, 5(4-5), 467-501. doi:10.1017/S1471068405002395

GeeksforGeeks. (2021, June 15). From SLR Parser (with Examples): https://www.geeksforgeeks.org/slr-parser-with-examples/

Javatpoint. (2022, Retrieved October 27, 2022). SLR 1 Parsing. From Javatpoint.: https://www.javatpoint.com/slr-1-parsing

*LALR parsers and yacc/bison - Bottom-up parsing*. (n.d.). Retrieved October 29, 2022, from https://1library.net/article/lalr-parsers-and-yacc-bison-bottom-up-parsing.y93309dy.

Mark van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In Proc. 11th International Conference on Compiler Construction, CC ’02, 2002.

Watson, D. (2017). *A Practical Approach to Compiler Construction (Undergraduate Topics in Computer Science*) (1st ed. 2017). Springer.

*What is Bottom-up Parsing?* (n.d.). Retrieved October 29, 2022, from https://www.tutorialspoint.com/what-is-bottom-up-parsing

William A. Barrett, Rodney M. Bates, David A. Gustafson, and John D. Couch. 1986. Compiler construction: theory and practice (2nd ed.). SRA School Group, USA.