Notes for CSC 2/454, Sept. 2, 9, and 14, 2020

In a real compiler, the input may have errors, including lexical errors. In most compilers, the parser drives the front end. A famous example was the if-then-else statements of Algol-60 and Pascal. It is necessary but not sufficient to make a grammar LL(1).

Note that \$LT(4,21)) \$ can be greater than 25, as the input length is not taken into account.

The concept of LL(1) is best seen in the context of a bottom-up parser. LL(1) means that when a parser is trying to match a token, it can only look at the stack and the input, without worrying about the context of the tokens.

As a corollary, it follows that a grammar is not LL(1) if there are two distinct derivations for the same input, where the next input token is different. In such a case, we can't make a decision without looking at the context of the tokens.

LL(1) grammars can be translated into a bottom-up parser by constructing a parse table, which is also called the reduction table.

A complete system for parsing includes the lexical analyzer, the parser proper (which is the compiler's brain), and the error recovery system. The S-R model divides the lexical analyzer, parser, and error recovery system into separate modules.

In a top-down parser, a rule is chosen to generate a subtree of the parse tree. The top-down parser then proceeds to fill in the leaves of the tree. For example, in the expression 3 + 4 * 5, the rule 3 + 4 * 5 is chosen, and then the rules for the operands 3 and 4 * 5 are applied.

The parse tree represents the structure of the expression. In this case, the parse tree is a binary tree, with the top node being the expression 3 + 4 * 5. The bottom nodes are the operands 3 and 4 * 5, and the leaves are the numbers 3, 4, and 5.

In a bottom-up parser, the rules are applied in reverse order. The parser starts with the root node and applies the rules in reverse order. For example, in the expression 3 + 4 * 5, the rule 3 + 4 * 5 is applied, and then the rules for the operands 3 and 4 * 5 are applied.

The parse tree represents the structure of the expression. In this case, the parse tree is a binary tree, with the root node being the expression 3 + 4 * 5. The leaves are the numbers 3 and 4 * 5, and the internal nodes are the operators + and *.

The parse tree allows for easy error recovery. If there is an error in the input, the parser can backtrack and try a different rule. For example, in the expression 3 + 4 * 5, if the operator * is recognized as a + instead, the parser can backtrack and try the rule 3 * 4 instead.

The parse tree also allows for easy error recovery. If there is an error in the input, the parser can backtrack and try a different rule. For example, in the expression 3 + 4 * 5, if the operator * is recognized as a + instead, the parser can backtrack and try the rule 3 + 4 instead.

The parse tree allows for easy error recovery. If there is an error in the input, the parser can backtrack and try a different rule. For example, in the expression 3 + 4 * 5, if the operator * is recognized as a + instead, the parser can backtrack and try the rule 3 * 4 instead.

The parse tree allows for easy error recovery. If there is an error in the input, the parser can backtrack and try a different rule. For example, in the expression 3 + 4 * 5, if the operator * is recognized as a + instead, the parser can backtrack and try the rule 3 + 4 instead.

The parse tree allows for easy error recovery. If there is an error in the input, the parser can backtrack and try a different rule. For example, in the expression 3 + 4 * 5, if the operator * is recognized as a + instead, the parser can backtrack and try the rule 3 * 4 instead.

The parse tree allows for easy error recovery. If there is an error in the input, the parser can backtrack and try a different rule. For example, in the expression 3 + 4 * 5, if the operator * is recognized as a + instead, the parser can backtrack and try the rule 3 + 4 instead.

The parse tree allows for easy error recovery. If there is an error in the input, the parser can backtrack and try a different rule. For example, in the expression 3 + 4 * 5, if the operator * is recognized as a + instead, the parser can backtrack and try the rule 3 * 4 instead.

The parse tree allows for easy error recovery. If there is an error in the input, the parser can backtrack and try a different rule. For example, in the expression 3 + 4 * 5, if the operator * is recognized as a + instead, the parser can backtrack and try the rule 3 + 4 instead.

The parse tree allows for easy error recovery. If there is an error in the input, the parser can backtrack and try a different rule. For example, in the expression 3 + 4 * 5, if the operator * is recognized as a + instead, the parser can backtrack and try the rule 3 * 4 instead.

The parse tree allows for easy error recovery. If there is an error in the input, the parser can backtrack and try a different rule. For example, in the expression 3 + 4 * 5, if the operator * is recognized as a + instead, the parser can backtrack and try the rule 3 + 4 instead.

The parse tree allows for easy error recovery. If there is an error in the input, the parser can backtrack and try a different rule. For example, in the expression 3 + 4 * 5, if the operator * is recognized as a + instead, the parser can backtrack and try the rule 3 * 4 instead.

The parse tree allows for easy error recovery. If there is an error in the input, the parser can backtrack and try a different rule. For example, in the expression 3 + 4 * 5, if the operator * is recognized as a + instead, the parser can backtrack and try the rule 3 + 4 instead.