

Compiler Design - Error Recovery

https://www.tutorialspoint.com/compiler_design/compiler_design_error_recovery.htm

Copyright © tutorialspoint.com

A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:

- **Lexical** : name of some identifier typed incorrectly
- **Syntactical** : missing semicolon or unbalanced parenthesis
- **Semantical** : incompatible value assignment
- **Logical** : code not reachable, infinite loop

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

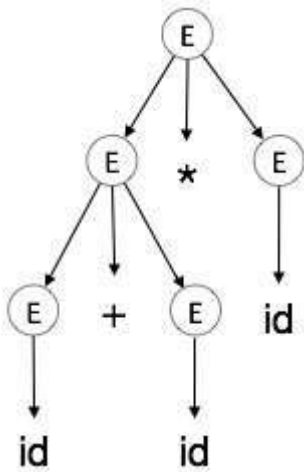
Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

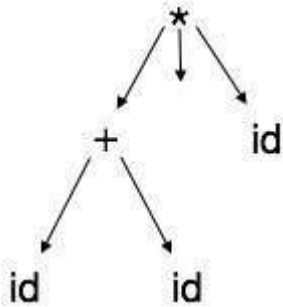
The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input *statement* X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity *timeandspace* of this strategy, it has not been implemented in practice yet.

Abstract Syntax Trees

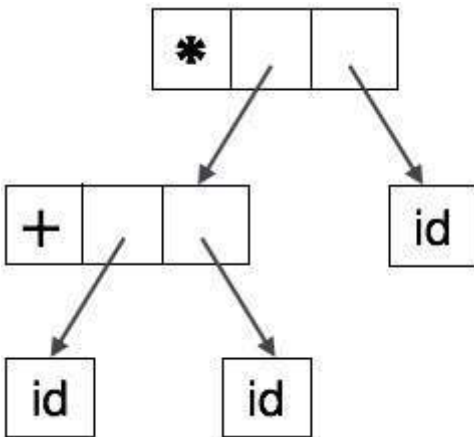
Parse tree representations are not easy to be parsed by the compiler, as they contain more details than actually needed. Take the following parse tree as an example:



If watched closely, we find most of the leaf nodes are single child to their parent nodes. This information can be eliminated before feeding it to the next phase. By hiding extra information, we can obtain a tree as shown below:



Abstract tree can be represented as:



ASTs are important data structures in a compiler with least unnecessary information. ASTs are more compact than a parse tree and can be easily used by a compiler.