Programming Language Concepts

From compiler
-> Lexical analysis
-> Syntax Analysis
-> Semantics Analyser
-> Intermediate code

Lecture 14

Programming Language Concepts

Parser:

- **Definition:** A parser is a software component that analyzes the syntax of a language. It takes input in the form of a sequence of tokens and checks whether the input conforms to the syntax rules of the language. If the input is syntactically correct, the parser typically produces a parse tree or abstract syntax tree.

- **Role in Compilers:** Parsers are a critical component of compilers and interpreters, responsible for translating source code into executable instructions.

Programming Language Concepts

Parser: Importance

- **Syntax Checking:** Parsers ensure that the input code follows the syntax rules defined by the programming language, helping programmers catch errors early in the development process.

- **Semantic Analysis:** Parsers may also perform semantic analysis, checking whether the input code has meaningful constructs according to the language's semantics.

Programming Language Concepts

Parser: Types

Top-Down Parsers

- **Overview:** Top-down parsers start with the highest level of abstraction in the grammar and recursively expand non-terminals to match the input.
- **Example:** Recursive Descent Parser
- **Advantages:** Intuitive, easy to implement manually.
- **Limitations:** May suffer from backtracking and left recursion issues.

Programming Language Concepts

Parser: Types

Bottom-Up Parsers

- **Overview:** Bottom-up parsers start with the input tokens and construct parse trees by applying production rules in reverse.
- **Example:** Shift-Reduce Parser, LR Parser
- **Advantages:** Efficient for large grammars, handle left recursion better.
- **Limitations:** Complex to implement, require parsing tables.

Programming Language Concepts

Parser: Recursive Descent Parsing

Overview: Recursive descent parsing is a top-down parsing technique where each non-terminal in the grammar is associated with a recursive function.

Characteristics:

- ***Recursive nature:*** Each non-terminal is typically associated with a parsing function.
- ***Predictive:*** Decisions are made based on the current token and lookahead.
- ***Example:*** Parsing arithmetic expressions using recursive descent.

E -> E + T/T
T -> T *F / F
F -> (E) | id

Programming Language Concepts

Step 1
E -> T E' (replace E with E')
E' -> +T E' | epsilon
T -> F T'
T' -> * FT' | epsilon

Parser: LL Parsing

Overview: LL parsing is a class of top-down parsers that parse input from left to right, constructing a leftmost derivation.

Steps:
1. Elimination of left recursion
2. Elimination of left factoring
3. First & Follow
4. Parsing table
5. Stack implementation
6. Parse tree

Characteristics:

- Uses a parsing table or predictive parsing techniques.
- Commonly used for parsing programming languages.

Example: LL(1) parsing table construction.

Programming Language Concepts

Parser: Shift Reduce Parsing

Overview: Shift-reduce parsing is a type of bottom-up parsing that performs two actions: shifting tokens onto a stack and reducing them according to production rules.

Characteristics:

- Operates on a stack and input buffer.
- Uses a set of parsing actions: shift, reduce, accept, or error.

Example: LR(1) parsing algorithm.

Programming Language Concepts

Parser: LR Parsing

Overview: LR parsing is a class of bottom-up parsers that parse input from left to right, constructing a rightmost derivation.

Characteristics:

- Uses a parsing table for efficient parsing.
- More powerful than LL parsing in handling a wider range of grammars.

Example: LR(1) parsing table construction.

Programming Language Concepts

Parser: Comparison of Parsing Techniques

Comparison Factors	Recursive Descent Parsing	LL Parsing	Shift-Reduce Parsing	LR Parsing
Efficiency	Depends on grammar and implementation May suffer from backtracking	Efficient for LL(1) grammars Construction of parsing tables can be time-consuming	Efficient, handles wider range of grammars Depends on parsing table size and automaton	Efficient for a broad class of grammars Construction of parsing tables can be complex
Ease of Implementation	Relatively easy, intuitive mapping between grammar productions and parsing functions	Construction of parsing tables based on grammar	More complex than LL parsing Involves state transitions and stack operations	Construction of parsing tables using algorithms such as SLR, LALR, CLR
Handling of Grammars	Limited in handling left-recursive grammars efficiently May require modifications for ambiguity	Well-suited for LL(1) grammars Limited in handling left recursion	Can handle left-recursive grammars Requires resolution strategies for conflicts	Most powerful, handles left-recursive and ambiguous grammars Efficient in resolving conflicts
Error Handling	Error recovery may be challenging Limited to simple error reporting	Error recovery techniques are possible. Additional parsing table entries.	Shift Reduce and Reduce-Reduce conflicts may arise, error recovery mechanisms.	Efficient error recovery mechanisms Detailed error messages.

Have a Nice Day