Formal Languages & Finite Automata

Laboratory work 6:

Parser & Building an Abstract Syntax Tree

Student: Grigoraș Dumitru
Verified: prof., Cojuhari Irina
univ. assist., Crețu Dumitru

Chișinău, 2024

Theory

A parser is a fundamental component of compilers and interpreters responsible for analyzing the syntactic structure of source code or input data according to a specified grammar. It operates after lexical analysis (performed by a lexer) and before semantic analysis.

1. Parsing Techniques:
    - Top-Down Parsing: This approach starts with the grammar's start symbol and attempts to derive the input through a sequence of grammar rule applications.
    - Bottom-Up Parsing: Contrary to top-down parsing, bottom-up parsing begins with the input tokens and works upwards to deduce the grammar rules applied.

Abstract Syntax Tree (AST):

2. An Abstract Syntax Tree is a hierarchical representation of the syntactic structure of source code or input data, capturing the relationships between various elements like expressions, statements, and declarations. Key aspects of ASTs include:
    - Nodes: Each node in the AST corresponds to a syntactic construct from the input. For example, an expression node might represent an arithmetic operation.
    - Edges: Edges between nodes depict the relationships between syntactic constructs. For instance, an edge from an assignment statement node to an identifier node signifies assignment to that identifier.
    - Traversal: ASTs can be traversed in various ways (pre-order, post-order, etc.) to perform operations such as code generation, optimization, or analysis.
3. Building an AST:
    - Parsing Process: During parsing, as the parser recognizes grammar rules and constructs, it constructs the AST concurrently.
    - Node Types: Different node types are used in ASTs to represent distinct syntactic constructs. For example, there may be nodes for binary expressions, if statements, loops, function declarations, etc.
    - Semantic Information: In addition to syntax, AST nodes often store semantic information such as types, variable names, constant values, etc.
4. Applications:
    - Compilation: ASTs are pivotal in compilers for generating intermediate code or target code from source code.
    - Interpretation: In interpreters, ASTs are directly used for executing program logic based on the parsed input.
    - Analysis: ASTs facilitate static analysis tasks like identifying code patterns, detecting errors, performing optimizations, and extracting information for tools like IDEs

Objectives:

1) Get familiar with parsing, what it is and how it can be programmed [1].
2) Get familiar with the concept of AST [2].
3) Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.
4) Implement a simple parser program that could extract the syntactic information from the input text

Implementation

ASTNode interface

The ASTNode interface in the Lab6.AST package provides a blueprint for nodes in an Abstract Syntax Tree (AST).

```
package Lab6.AST;

import java.util.List;

public interface ASTNode {

   String getText();

}
```

AgentNode class:

The AgentNode class in the Lab6.AST package represents a specific type of node in an Abstract Syntax Tree (AST) related to agents. Here's an overview of the class:

1. Fields:
    - agentName: A string field that stores the name of the agent associated with this node.
    - children: A list of ASTNode objects representing the child nodes of this agent node.
2. Constructor:
    - AgentNode(String agentName): Initializes an AgentNode object with the given agent name and initializes the list of children nodes.
3. Methods:
    - getAgentName(): Returns the name of the agent associated with this node.
    - addChild(ASTNode child): Adds a child node to the list of children nodes.
    - getChildren(): Returns the list of child nodes.
    - getText(): Implements the getText() method from the ASTNode interface, returning a formatted string representing the agent node.
4. Usage:
    - This class is used to represent agent nodes in an AST, where each agent node may have child nodes representing various aspects or actions related to the agent.
    - Developers can create instances of AgentNode to build up an AST structure for modeling agent-based systems or related concepts.

DestinationNode class:
The DestinationNode class in the Lab6.AST package is designed to represent a node in an Abstract Syntax Tree (AST) that corresponds to a destination agent. Below is an overview of the class:

1. Fields:
    - destinationAgent: A string field that stores the name of the destination agent associated with this node.
    - children: A list of ASTNode objects representing the child nodes of this destination node.
2. Constructor:
    - DestinationNode(String destinationAgent): Initializes a DestinationNode object with the given destination agent name.
3. Methods:
    - getDestinationAgent(): Returns the name of the destination agent associated with this node.
    - addChild(ASTNode child): Adds a child node to the list of children nodes.
    - getChildren(): Returns the list of child nodes.
    - getText(): Implements the getText() method from the ASTNode interface, returning a formatted string representing the destination node.
4. Usage:
    - Developers can use the DestinationNode class to represent nodes in an AST that denote destination agents within a system or context.
    - The addChild() method allows for adding child nodes to the destination node, enabling the construction of hierarchical AST structures.
    - getText() provides a textual representation of the destination node, which can be useful for displaying or processing ASTs in a human-readable format.

TextMessageNode class:
The TextMessageNode class in the Lab6.AST package is designed to represent a node in an Abstract Syntax Tree (AST) that contains text message information. Below is an overview of the class:

1. Fields:
    - messageText: A string field that stores the text message associated with this node.
    - children: A list of ASTNode objects representing the child nodes of this text message node.
2. Constructor:
    - TextMessageNode(String messageText): Initializes a TextMessageNode object with the given message text.
3. Methods:
    - getMessageText(): Returns the text message associated with this node.
    - addChild(ASTNode child): Adds a child node to the list of children nodes.
    - getChildren(): Returns the list of child nodes.
    - getText(): Implements the getText() method from the ASTNode interface, returning a formatted string representing the text message node.
4. Usage:
    - Developers can use the TextMessageNode class to represent nodes in an AST that contain text messages or textual content within a larger context.
    - The addChild() method allows for adding child nodes to the text message node, enabling the construction of hierarchical AST structures.
    - getText() provides a textual representation of the text message node, which can be useful for displaying or processing ASTs in a human-readable format.

Parser class:

The Parser class provided above is an implementation of a simple parser for a specific grammar that parses input sentences and builds an Abstract Syntax Tree (AST) based on the parsed structure. Below is an overview of the class and its functionality:

1. Class Structure:
    - The Parser class has fields for storing tokens (List<Token> tokens) and tracking the current token index (int currentTokenIndex).
    - It contains methods for parsing different components of the input sentence and constructing the AST nodes accordingly.
2. Constructor:
    - Parser(List<Token> tokens): Initializes the parser with a list of tokens generated by a lexer.
3. Parsing and AST Construction:
    - parse(): This method is the entry point for parsing. It checks the syntax of the input sentence and constructs the AST nodes accordingly. The method follows a specific grammar defined by the match() method and the expected token types.
    - agent(), textMessage(), and destination(): These methods handle parsing of specific parts of the input sentence corresponding to agent, text message, and destination components, respectively.
    - match(TokenType expectedType): Matches the current token type with the expected type and advances the token index if the match is successful.
    - peek(): Retrieves the current token without advancing the index.
    - advance(): Moves to the next token in the list.
4. Main Method:
    - The main method demonstrates the usage of the parser by tokenizing an input sentence, creating a parser instance, parsing the input, and printing the resulting AST nodes.
5. AST Node Printing:
    - printAST(ASTNode node): Recursively prints the AST nodes in a hierarchical manner, distinguishing between different types of AST nodes (AgentNode, TextMessageNode, DestinationNode).
6. Usage:
    - Developers can use this parser to parse input sentences based on the grammar defined in the Parser class and build an AST representing the syntactic structure of the input.
    - The AST can then be further processed or used for various purposes such as semantic analysis, code generation, or interpretation.

Conclusion

The lab work involved creating a parser for a specific grammar and building an Abstract Syntax Tree (AST) based on the parsed input sentences. Here is a conclusion for this lab work:
In this lab, we implemented a parser that demonstrates the process of syntactic analysis and AST construction for a simple language. The parser takes input sentences, tokenizes them using a lexer, checks their syntax based on predefined grammar rules, and constructs an Abstract Syntax Tree (AST) representing the hierarchical structure of the input.

The key components of our parser include methods for parsing different elements of the language, such as agents, text messages, and destinations. These methods use a set of predefined rules to match tokens and build corresponding AST nodes. The parser ensures that input sentences adhere to the specified grammar, handling syntax errors gracefully and providing informative error messages.

The main method of the parser showcases its functionality by tokenizing an input sentence, parsing it using the defined grammar rules, and then printing the resulting AST nodes in a hierarchical manner. This process helps in understanding how the parser interprets the syntactic structure of the input and organizes it into a meaningful AST representation.

Overall, this lab work has provided valuable insights into the process of parsing, grammar-based syntax analysis, and AST construction, which are fundamental concepts in language processing and compiler design. By implementing this parser, we have gained a deeper understanding of how programming languages are parsed and represented internally, laying the groundwork for more advanced language processing tasks in the future.

Bibliography

[1] Formal Languages and Compiler Design Accessed February 14, 2024. https://else.fcim.utm.md/pluginfile.php/110457/mod_resource/content/0/Theme_1.pdf.

[2] Regular Language. Finite Automata. Accessed February 15, 2024. https://drive.google.com/file/d/1rBGyzDN5eWMXTNeUxLxmKsf7tyhHt9Jk/view