Synthia’s Parser Documentation

Introduction

This documentation offers a comprehensive insight into the parsing algorithm employed by the parser in the provided codebase. The primary role of the parser is to meticulously scrutinize the syntax of source code and subsequently construct an abstract syntax tree (AST) to represent the program's structural elements.

Recursive Descent Parsing

The parser leverages a recursive descent parsing technique, a top-down approach that breaks the parsing process into smaller, more manageable components. Each of these components corresponds to a specific grammar rule or production, resulting in a modular and highly organized parsing structure. Recursive calls to these parsing functions are utilized to dissect and validate the input source code.

Parser Architecture

At the core of the parser is the `Parser` struct, which encapsulates several fundamental components:

- *Integration with Lexer*: The parser heavily relies on a lexer to tokenize the source code. These tokens are the fundamental building blocks that facilitate parsing.

- *Token Management*: The `current_token` and `peek_token` fields play pivotal roles in tracking the tokens currently under scrutiny. The `next_token` method efficiently advances the current token and peeks at the subsequent token provided by the lexer.

- *Error Handling*: An integral part of the parser's functionality is error handling. The `errors` vector is designated for storing parsing errors. Whenever an error is encountered, the parser formulates a corresponding error message and appends it to this vector for later reference and reporting.

Parsing Workflow

The parsing workflow can be succinctly summarized as follows:

1. *Initialization*: The parser is initialized by providing it with the lexer. Additionally, initial tokenization is performed, initializing the `current_token` and `peek_token` fields. The `errors` vector is established to manage error reporting.

2. *Parsing the Program :* The `parse_program` function serves as the entry point for the parsing process. This function commences a loop that persists until the conclusion of the program, denoted by encountering the `Token::Eof`.

- ***Statement Parsing***: Within this loop, the `parse_statement` function is invoked to meticulously parse individual statements.

- ***Statement Accumulation***: Successfully parsed statements are accumulated within the `statements` vector.

- ***Token Advancement***: The parser systematically advances to the subsequent token via the `next_token` method.

3. ***Parsing Statements***: The `parse_statement` function exhibits the capability to discern and delegate the parsing of distinct statement types, encompassing assignments, returns, includes, and expression statements.

4. ***Parsing Expressions***: Expressions are dissected by the `parse_expr` function. This versatile function adeptly handles a myriad of expression types, including identifiers, literals (e.g., numbers, booleans, strings), prefix and infix expressions (e.g., negation, addition), function calls, and indexed expressions. Operator precedence is diligently observed to ensure the correct sequencing of parsing operations.

5. ***Error Management***: Error handling remains integral throughout the parsing endeavor. In instances where unexpected tokens or grammatical anomalies are encountered, the parser responds by generating error messages. These informative error messages are meticulously compiled and deposited within the `errors` vector for subsequent examination and debugging.

Conclusion

The recursive descent parsing algorithm, proficiently implemented in this parser, endows it with the capability to scrutinize the intricacies of programming language syntax. By decomposing the parsing task into discrete, specialized functions, it effectively navigates and dissects intricate language constructs. Moreover, it empowers developers by providing detailed error messages, facilitating the debugging process, and fostering the creation of robust code.

Synthia’s Lexer Documentation

Introduction

This documentation provides a detailed explanation of the Lexer component in the provided codebase. The Lexer plays a crucial role in the early stages of the compilation process, responsible for tokenizing the source code. It scans the input code character by character and converts it into meaningful tokens that the Parser can then utilize for syntax analysis.

Lazy Static Initialization

Before delving into the Lexer's functionality, it's essential to highlight the `lazy_static` block at the beginning of the code. This block initializes a `HashMap` called `KEYWORDS` that associates keywords with their corresponding tokens. This mapping aids in recognizing reserved words in the source code and categorizing them accordingly.

Tokenization Process

The Lexer tokenization process is driven by the following key components and functions:

- ***Character Management*** : The Lexer tracks the current character (`ch`), the current position in the input (`position`), and the next position to read (`read_position`). It starts by setting the initial character and positions.

- ***Reading Characters*** : The `read_char` function is pivotal in advancing through the input characters. It updates the current character and position while ensuring bounds checking to handle the end of input gracefully.

- ***Identifier Recognition*** : The `read_identifier` function identifies and reads identifiers. It scans characters while they conform to the rules of valid identifiers (ASCII alphabetic characters or underscores).

- ***Numeric Value Parsing*** : Numeric values are parsed by the `read_number` function. This function identifies and reads numbers, including floating-point numbers. It scans characters until a non-numeric character or decimal point is encountered.

- ***String Literal Extraction*** : The `read_string` function extracts string literals enclosed within double quotes. It captures characters between the opening and closing double quotes.

- ***Comment Handling*** : Comments beginning with `//` are recognized and skipped by the `read_comment` function. It scans until a newline character or the end of input is reached.

- ***Whitespace Skipping*** : The `skip_whitespace` function efficiently skips whitespace characters, including spaces, tabs, newlines, and carriage returns.

- ***Peeking at the Next Character*** : The `peek_char` function allows the Lexer to look ahead at the next character without consuming it. This is useful for recognizing multi-character tokens, such as equality (`==`) and inequality (`!=`).

- ***Token Generation*** : The `next_token` function is central to the tokenization process. It determines the type of token based on the current character and, if necessary, the next character (for multi-character tokens). It returns the appropriate token and advances the Lexer's position.

Token Types

The Lexer recognizes a variety of token types, including:

- ***Keywords*** : Reserved words like `make`, `func`, `if`, `else`, and others are recognized as keywords and mapped to their respective tokens.

- ***Punctuation***: Characters like `;`, `:`, `,`, `(`, `)`, `{`, `}`, `[`, `]`, `*`, `/`, `+`, `-`, `&`, `|`, `^`, `%`, `~`, and `!` represent various punctuation tokens.

- ***Comparison and Assignment Operators***: Tokens such as `=`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `<`, `<<`, `>>` are identified as comparison and assignment operators.

- ***String Literals*** : Double-quoted strings are extracted as string literal tokens.

- ***Single Quote 's' Token***: The Lexer recognizes a single-quote `'s'` token.

- ***Numeric Literals***: Numbers, including integers and floating-point values, are recognized and tokenized.

- ***Identifiers***: User-defined identifiers are captured as identifier tokens.

- ***Comments***: Comments are skipped and treated as comment tokens.

- ***Illegal Characters***: Any unrecognized or invalid characters result in an illegal token.

- ***End of Input*** : The end of the input is marked by the `Eof` token.

Conclusion

The Lexer component in the codebase is responsible for transforming raw source code into a stream of meaningful tokens. This initial step in the compilation process ensures that the subsequent parsing phase can effectively analyze the syntax of the programming language. The Lexer's ability to recognize and categorize various token types, keywords, and literals is fundamental to the accurate interpretation of the source code.

Synthia’s Token’s Documentation

This documentation provides an overview of the Token enumeration in the codebase. Tokens are fundamental units of a programming language, representing individual lexical elements in the source code. They serve as building blocks for the parsing and interpretation of programming languages.

Token Enumeration

The `Token` enumeration encompasses a wide range of tokens, including literals, operators, delimiters, keywords, and special tokens. Each variant of the `Token` enum represents a specific category of lexical element.

Special Tokens

1. ***Eof***: This token marks the end of the input source code.

2. ***Illegal***: The `Illegal` token represents any characters or sequences that are not recognized as valid tokens.

3. ***Comment***: The `Comment` token is used for comments in the source code. It is typically skipped during tokenization and does not impact the program's execution.

Literals

4. ***Ident(String)*** : The `Ident` token represents user-defined identifiers, such as variable names or function names, as strings.

5. ***Number(f64)*** : Numeric literals, including both integers and floating-point values, are represented by the `Number` token.

6. ***String(String)***: The `String` token captures string literals enclosed within double quotes.

7. ***Boolean(bool)***: Boolean literals, `true` and `false`, are represented by the `Boolean` token.

Operators

8. ***Assign***: The `Assign` token signifies the assignment operator (`=`).

9. ***Plus*** : The `Plus` token represents the addition operator (`+`).

10. ***Minus***: The `Minus` token represents the subtraction operator (`-`).

11. ***Bang***: The `Bang` token represents the logical NOT operator (`!`) or the bitwise NOT operator (`~`).

12. ***Asterisk***: The `Asterisk` token represents the multiplication operator (`*`).

13. ***Slash***: The `Slash` token represents the division operator (`/`).

14. ***Percent***: The `Percent` token represents the modulo operator (`%`).

Comparison Operators

21.***Less***: The `Less` token represents the less-than operator (`<`).

22. ***Greater***: The `Greater` token represents the greater-than operator (`>`).

23. ***LessEqual***: The `LessEqual` token represents the less-than-or-equal-to operator (`<=`).

24. ***GreaterEqual***: The `GreaterEqual` token represents the greater-than-or-equal-to operator (`>=`).

25. ***Equals***: The `Equals` token represents the equality operator (`==`).

26. ***NotEquals***: The `NotEquals` token represents the inequality operator (`!=`).

Delimiters

27. ***Comma***: The `Comma` token represents a comma (`,`), often used to separate elements in lists.

28. ***Colon***: The `Colon` token represents a colon (`:`), often used in various contexts such as defining types.

29. ***Semicolon***: The `Semicolon` token represents a semicolon (`;`), often used to terminate statements.

30. ***LeftParen***: The `LeftParen` token represents a left parenthesis (`(`).

31. ***RightParen***: The `RightParen` token represents a right parenthesis (`)`).

32. ***LeftBrace***: The `LeftBrace` token represents a left curly brace (`{`), often used to denote code blocks.

33. ***RightBrace***: The `RightBrace` token represents a right curly brace (`}`).

34. ***LeftBracket***: The `LeftBracket` token represents a left square bracket (`[`).

35. ***RightBracket***: The `RightBracket` token represents a right square bracket (`]`).

Keywords

37. ***Set***: The `Set` token represents the keyword `set`.

38. ***Func***: The `Func` token represents the keyword `func`.

39. ***If***: The `If` token represents the keyword `if`.

40. ***Else***: The `Else` token represents the keyword `else`.

41. ***Return***: The `Return` token represents the keyword `return`.

42. ***Include***: The `Include` token represents the keyword `add`.

43. ***Typeof***: The `Typeof` token represents the keyword `typeof`.

44. ***Loop***: The `Loop` token represents the keyword `loop`.

45. ***Break***: The `Break` token represents the keyword `break`.

46. ***Continue***: The `Continue` token represents the keyword `continue`.

Display Implementation

The `Token` enum includes a `Display` implementation, which allows tokens to be formatted as strings when needed.

Conclusion

Tokens are the foundational elements of the lexer, representing the smallest units of a programming language. The `Token` enum in the codebase comprehensively covers a wide range of token types, ensuring that the lexer can effectively identify and categorize lexical elements in the source code. These tokens are subsequently used in the parsing and interpretation phases of the compilation process.

Synthia's Scanner Documentation

This documentation provides an overview of the `Scanner` struct and its associated methods, which are responsible for tokenizing source code. Tokenization involves breaking down the input source code into individual tokens, which are fundamental units used for subsequent parsing and interpretation.

Scanner Struct

The `Scanner` struct is responsible for processing source code and generating a list of tokens. It maintains the state of the scanning process, including the current position in the source code, line number, and a collection of generated tokens.

Fields

- `**source**`: A `String` that holds the source code to be tokenized.

- `**tokens**`: A `Vec<Token>` that stores the generated tokens.

- `**start**`: An index that marks the start of the current token being scanned.

- `**current**`: An index that represents the current position in the source code.

- `**line**`: An integer that tracks the current line number in the source code.

- `**keywords**`: A `HashMap<&'static str, TokenType>` that maps keyword strings to their corresponding token types.

Methods

`fn new(source: &str) -> Self`

- *Parameters*: `source` - A reference to the source code as a string.
- *Returns*: A new `Scanner` instance initialized with the provided source code.

This method creates a new `Scanner` instance, initializing it with the source code.

`fn scan_tokens(self: &mut Self) -> Result<Vec<Token>, String>`

- ***Returns***: A `Result` containing either a vector of `Token` instances if the scanning is successful or an error message as a string if there are scanning errors.

This method initiates the tokenization process, scanning the entire source code and generating tokens. It returns the list of tokens if successful or an error message if there are scanning errors. This method also adds an EOF (end-of-file) token to mark the end of the token stream.

`fn is_at_end(self: &Self) -> bool`

- ***Returns***: `true` if the scanner has reached the end of the source code; otherwise, `false`.

This method checks whether the scanner has reached the end of the source code.

`fn scan_token(self: &mut Self) -> Result<(), String>`

- ***Returns***: A `Result` indicating success or an error message if there are scanning errors.

This method scans the next token in the source code and adds it to the list of tokens. It handles different types of tokens, including single-character tokens, two-character tokens, literals, and identifiers. If an error occurs during scanning, it returns an error message.

`fn identifier(&mut self)`

This method handles the scanning of identifiers. It continues scanning while encountering alphanumeric characters and checks if the scanned identifier matches any keywords in the `keywords` hashmap. It then adds the corresponding token to the list of tokens.

`fn number(self: &mut Self) -> Result<(), String>`

- ***Returns***: A `Result` indicating success or an error message if there are parsing errors.

This method handles the scanning of numeric literals. It scans both integer and floating-point literals and adds the corresponding `Number` token to the list of tokens.

`fn peek_next(self: &Self) -> char`

- ***Returns***: The next character in the source code, or `'\0'` if the scanner has reached the end.

This method returns the character immediately following the current position in the source code.

`fn string(self: &mut Self) -> Result<(), String>`

- ***Returns***: A `Result` indicating success or an error message if there are parsing errors.

This method handles the scanning of string literals enclosed in double quotes. It continues scanning until it encounters the closing double quote, adding the `StringLit` token to the list of tokens. If the string literal is unterminated, it returns an error.

`fn peek(self: &Self) -> char`

- ***Returns***: The current character in the source code, or `'\0'` if the scanner has reached the end.

This method returns the character at the current position in the source code without advancing the scanner.

`fn char_match(self: &mut Self, ch: char) -> bool`

- ***Returns***: `true` if the current character matches the provided character `ch`; otherwise, `false`.

This method checks whether the current character in the source code matches the provided character `ch`. If a match is found, it advances the scanner position.

`fn advance(self: &mut Self) -> char`

- ***Returns***: The current character at the scanner's position, and advances the scanner to the next character.

This method returns the current character in the source code and advances the scanner to the next character.

`fn add_token(self: &mut Self, token_type: TokenType)`

- ***Parameters***: `token_type` - The type of token to add to the list of tokens.

This method adds a token with the specified `token_type` to the list of tokens, using the current scanning position to determine the token's lexeme.

`fn add_token_lit(self: &mut Self, token_type: TokenType, literal: Option<LiteralValue>)`

- ***Parameters***:
  - *token_type* - The type of token to add to the list of tokens.
  - *literal* - An optional literal value associated with the token.

This method adds a token with the specified `token_type` to the list of tokens. It also accepts an optional literal value associated with the token.

TokenType Enumeration

The `TokenType` enumeration defines all the possible token types that can be generated by the `Scanner`. Each variant corresponds to a specific type of token, such as keywords, operators, literals, and delimiters.

LiteralValue Enumeration

The `LiteralValue` enumeration represents possible literal values associated with tokens. It includes variants for floating-point numbers (`FValue`) and string values (`StringValue`).

Token Struct

The `Token` struct represents an individual token generated by the scanner. It includes information about the token's type, lexeme (textual representation), optional literal value, and the line number where the token was found.

Conclusion

The `Scanner` struct plays a crucial role in the compilation process by breaking down source code into discrete tokens. These tokens are essential for subsequent parsing and interpretation steps. The provided methods and enumerations help handle different types of tokens and enable robust error handling during the scanning process.