COMPILER CONSTRUCTION

CORE FUNCTIONS

The two core functions of a mini compiler typically include **Lexical Analysis (Scanning)** and **Syntax Analysis (Parsing)**. These functions are essential in converting source code into an intermediate representation for further processing.

**Lexical Analysis (Scanning)**

using System;

using System.Text.RegularExpressions;

public class Lexer

{

private string \_input;

private int \_position;

public Lexer(string input)

{

\_input = input;

\_position = 0;

}

public Token GetNextToken()

{

if (\_position >= \_input.Length)

return new Token("EOF", "End of File");

char current = \_input[\_position];

// Skip whitespaces

while (char.IsWhiteSpace(current))

{

\_position++;

if (\_position >= \_input.Length)

return new Token("EOF", "End of File");

current = \_input[\_position];

}

// Match keywords and identifiers

if (char.IsLetter(current))

{

string identifier = ReadWhile(char.IsLetterOrDigit);

if (IsKeyword(identifier))

return new Token("Keyword", identifier);

return new Token("Identifier", identifier);

}

// Match numbers

if (char.IsDigit(current))

{

string number = ReadWhile(char.IsDigit);

return new Token("Number", number);

}

// Match operators

if ("+-\*/=;".Contains(current))

{

\_position++;

return new Token("Operator", current.ToString());

}

throw new Exception($"Unexpected character: {current}");

}

private string ReadWhile(Func<char, bool> condition)

{

int start = \_position;

while (\_position < \_input.Length && condition(\_input[\_position]))

\_position++;

return \_input.Substring(start, \_position - start);

}

private bool IsKeyword(string word)

{

return word == "if" || word == "else" || word == "while" || word == "return";

}

}

public class Token

{

public string Type { get; }

public string Value { get; }

public Token(string type, string value)

{

Type = type;

Value = value;

}

public override string ToString()

{

return $"Type: {Type}, Value: {Value}";

}

}

// Example usage

class Program

{

static void Main(string[] args)

{

string code = "if x == 10 return x;";

Lexer lexer = new Lexer(code);

Token token;

do

{

token = lexer.GetNextToken();

Console.WriteLine(token);

} while (token.Type != "EOF");

}

}

**Explanation**:

* This lexer reads the input string and generates tokens based on rules.
* It skips whitespace, identifies keywords, reads identifiers, and parses numbers or operators.

**Syntax Analysis (Parsing)**

**Purpose**: The parser takes the tokens produced by the lexical analyzer and constructs a syntax tree (or abstract syntax tree, AST) based on grammar rules.

using System;

using System.Collections.Generic;

public class Parser

{

private List<Token> \_tokens;

private int \_position;

public Parser(List<Token> tokens)

{

\_tokens = tokens;

\_position = 0;

}

public Node Parse()

{

return ParseExpression();

}

private Node ParseExpression()

{

Node left = ParsePrimary();

while (Match("Operator"))

{

Token operatorToken = Previous();

Node right = ParsePrimary();

left = new BinaryExpressionNode(left, operatorToken, right);

}

return left;

}

private Node ParsePrimary()

{

if (Match("Number"))

return new NumberNode(Previous());

if (Match("Identifier"))

return new IdentifierNode(Previous());

throw new Exception("Unexpected token.");

}

private bool Match(string type)

{

if (Check(type))

{

Advance();

return true;

}

return false;

}

private bool Check(string type)

{

if (IsAtEnd()) return false;

return \_tokens[\_position].Type == type;

}

private Token Advance()

{

if (!IsAtEnd()) \_position++;

return Previous();

}

private bool IsAtEnd()

{

return \_position >= \_tokens.Count;

}

private Token Previous()

{

return \_tokens[\_position - 1];

}

}

public abstract class Node { }

public class BinaryExpressionNode : Node

{

public Node Left { get; }

public Token Operator { get; }

public Node Right { get; }

public BinaryExpressionNode(Node left, Token operatorToken, Node right)

{

Left = left;

Operator = operatorToken;

Right = right;

}

}

public class NumberNode : Node

{

public Token Token { get; }

public NumberNode(Token token)

{

Token = token;

}

}

public class IdentifierNode : Node

{

public Token Token { get; }

public IdentifierNode(Token token)

{

Token = token;

}

}

// Example usage

class Program

{

static void Main(string[] args)

{

var tokens = new List<Token>

{

new Token("Identifier", "x"),

new Token("Operator", "+"),

new Token("Number", "10"),

new Token("EOF", "End of File")

};

Parser parser = new Parser(tokens);

Node syntaxTree = parser.Parse();

Console.WriteLine("Parsing completed successfully.");

}

}

**Explanation**:

* The parser generates a binary expression tree from tokens.
* It checks tokens for specific types (Number, Operator, etc.) and combines them into nodes.
* The Binary\_Expression\_Node represents operations like addition.