PL_Compiler

پروژه طراحی کامپایلر

استاد درس: دکتر آرش شفیعی

دانشجویان: محمدامین صابری_۹۹۳۶۲۳۰۲۶

ابوالفضل شیشه گر_۹۹۳۶۲۳۰۲۵

دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

بهار ۱۴۰۳

فهرست عناوین

فاز اول – Lexer

مقدمه

این برنامه یک تحلیل‌کننده توکن ساده برای زبان برنامه‌نویسی مجازی به نام PL است. این برنامه قادر است کدهای PL را خوانده، تحلیل کند و توکن‌های مختلف آن‌ها را شناسایی کند.

عملکرد

برنامه از ورودی یک فایل با پسوند pl. استفاده می‌کند و کلمات، نمادها، عملگرها و ساختارهای مختلف کد را مورد بررسی قرار می‌دهد. سپس توکن‌ها را شناسایی کرده و اطلاعات مربوط به هر توکن را چاپ می‌کند.

کدهای اصلی

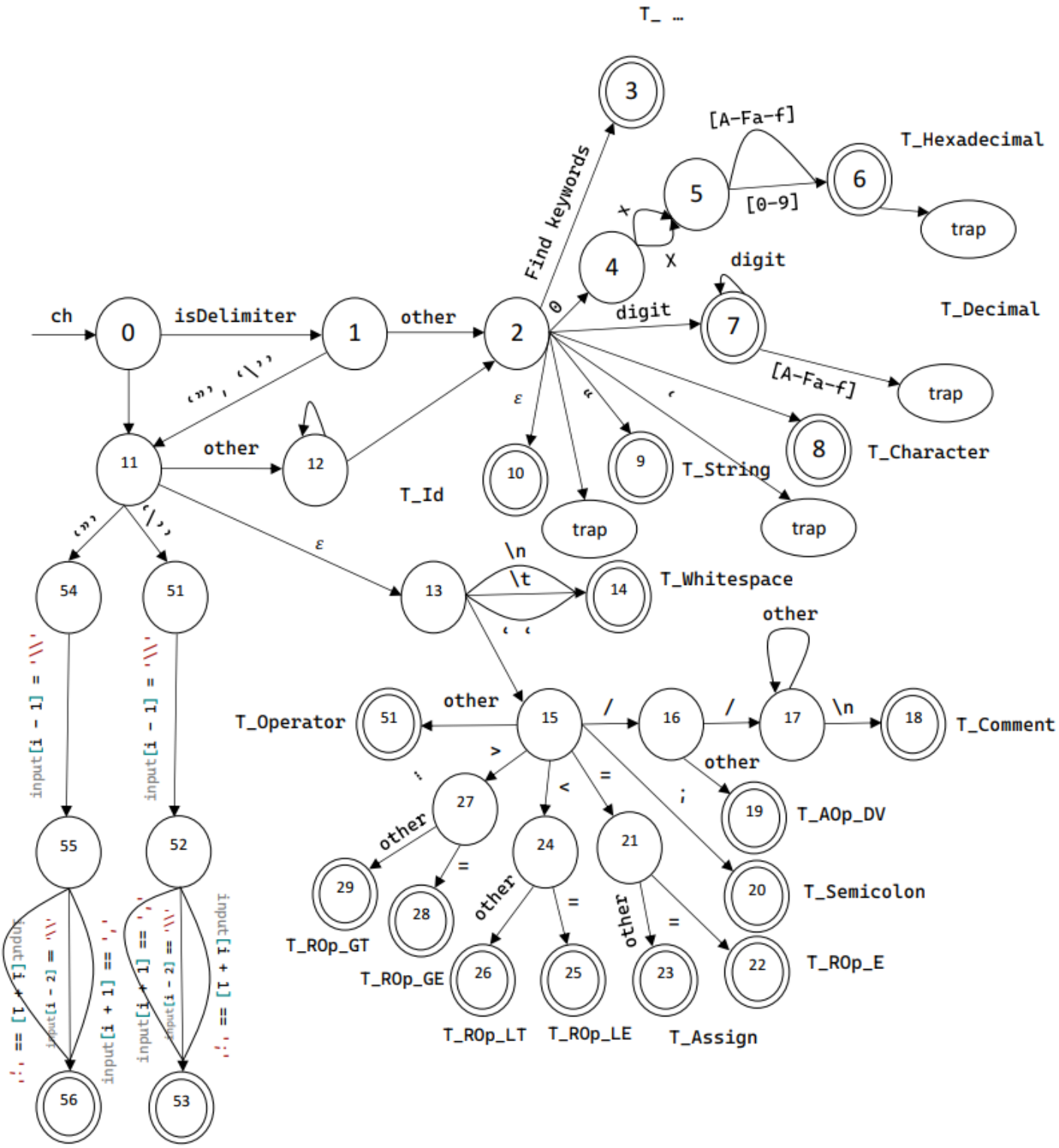
۱. **اعلانات و نمایش توکن‌ها** :ابتدا اعلاناتی مانند اعلان توکن‌ها و لیست کلمات کلیدی اعمال می‌شوند. سپس تابع **main** تعریف شده و از کاربر مسیر فایل pl. را درخواست می‌کند.

۲. **پردازش فایل** :فایل مورد نظر خوانده می‌شود و محتوای آن در یک رشته ذخیره می‌شود.

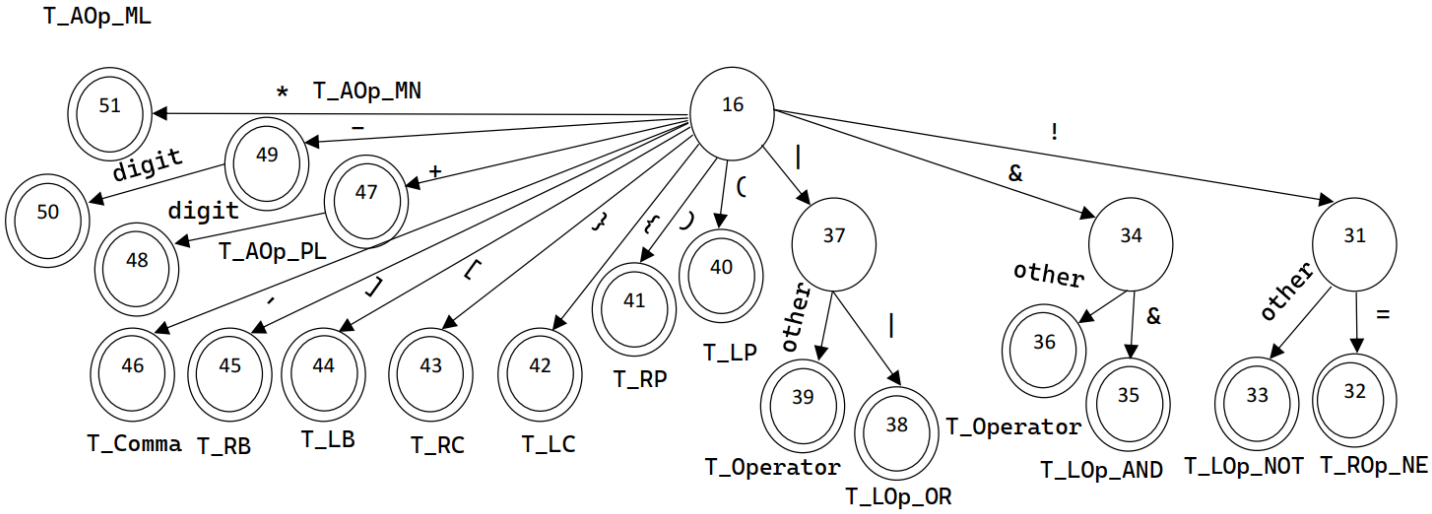
۳. **تحلیل توکن‌ها** :تابع **tokenize**برای تجزیه و تحلیل توکن‌ها استفاده می‌شود. این تابع به ترتیب هر کاراکتر از محتوای فایل را مورد بررسی قرار می‌دهد و توکن‌ها را تشخیص می‌دهد.

۴. **تشخیص توکن‌ها** :در هنگام تجزیه توکن‌ها، توکن‌های مختلفی از جمله کلمات کلیدی، اعداد، رشته‌ها، نمادها و عملگرها شناسایی می‌شوند. هر توکن با توجه به نوع خود چاپ می‌شود.

دیاگرام‌های گذار:



ادامه state شماره ۱۶



عبارات منظم

۱. اعداد هگزادسیمال :

- 0[xX][0-9a-fA-F]+

❖ یک 0 و یک x یا X حتما باید در ابتدای آن باشد و بعد از آن به بعد هر یک کاراکترهای A تا F ، a تا f و اعداد ۰ تا ۹ می‌تواند از یک بار تا بیشتر باشد.

۲. اعداد دسیمال:

- -?[0-9]+

❖ علامت منفی در پشت اعداد دسیمال می‌تواند وجود داشته باشد یا نداشته باشد و بعد از آن اعداد از ۰ تا ۹ می‌توانند با یک تکرار یا بیشتر وجود داشته باشند.

۳. کاراکترها:

- ‘[0-9A-Fa-f operators delimiters]*’

❖ برای تشخیص توکن‌های کاراکتر، هر آنچه میان دو تک کوتیشن قرار بگیرد را به عنوان توکن کاراکتر شناسایی می‌کنیم.

۴. رشته‌ها:

- “[0-9A-Fa-f operators delimiters]*”

❖ برای تشخیص توکن‌های رشته نیز مانند کاراکتر عمل می‌کنیم، با این تفاوت که هرچه را که میان دو double quotation بود را به عنوان توکن رشته شناسایی می‌کنیم.

۵. Id ها:

- _?[a-fA-F]*[0-9a-fA-F]*

❖ برای تشخیص توکن‌های id هر آن چیزی که جزو ۴ دسته قبلی نباشد و شامل اعداد و حروف باشد، به طوری که اول آن عدد نباشد، حروف یا زیر خط _ باشد، به عنوان توکن id شناخته می‌شود.

۶. اوپراتورها: برای اوپراتور ها، هر اوپراتوری که دیده بشود، همان موقع تشخیص داده می‌شود. برای بعضی از اوپراتور ها که دو کاراکتر پشت سر هم دارند، باید تا دیدن کاراکتر بعدی صبر کرد، مانند &&، ||، ==، => و ... .

۷. کامنت‌ها:

- $(//)^{+}$[0-9A-Fa-f]*other*\n

❖ اگر دو / پشت سر هم ببینیم، تا موقع رسیدن به یک اینتر '\n' که موجب رفتن به خط جدید شود، همه چیز به عنوان توکن کامنت شناسایی می‌شود. تعداد // ها باید حداقل یکی باشد تا کامنت تشخیص داده شود، بنابراین از + استفاده کردیم.

تحلیل کلی پروژه

۱. **اعلانات و ایجاد فضای نام:** در این بخش، کتابخانه‌های مورد نیاز به کد اضافه می‌شوند و فضای نام std اعلان می‌شود.

۲. **تعریف متغیرها و توابع:** در این قسمت، متغیرها و توابعی که برای پیاده‌سازی برنامه لازم هستند، تعریف می‌شوند. این شامل تعریف مجموعه کلمات کلیدی، توابع بررسی توکن‌ها و تابع اصلی است.

۳. **متغیرهای اصلی و ورودی کاربر:** در این بخش، متغیرهایی برای نگه‌داری مسیر فایل ورودی و باز کردن فایل تعریف می‌شوند. سپس کاربر از طریق ورودی استاندارد (cin) مسیر فایل مورد نظر را وارد می‌کند.

۴. **باز کردن فایل و بررسی موفقیت باز کردن:** در این بخش، فایل باز می‌شود و در صورت موفقیت یا عدم موفقیت در باز کردن، پیام مناسب چاپ می‌شود.

۵. **خواندن و تحلیل محتوای فایل:** در این قسمت، محتوای فایل به صورت خط به خط خوانده می‌شود و سپس تابع تحلیل توکن‌ها بر روی این محتوا فراخوانی می‌شود.

۶. **تحلیل توکن‌ها:** در این بخش، هر کاراکتر از محتوای فایل بررسی می‌شود و توکن‌ها تشخیص داده می‌شوند. توکن‌های مختلف از جمله کلمات کلیدی، اعداد، رشته‌ها، نمادها و عملگرها شناسایی و پردازش می‌شوند.

تحلیل تابع tokenize

۱. **تعریف متغیرها و آغاز حلقه:** در این بخش، متغیرهای مورد نیاز برای تحلیل توکن‌ها تعریف می‌شوند، از جمله یک متغیر رشته‌ای برای نگه‌داری توکن فعلی، و دو پرچم برای نشان دادن این که آیا درون یک رشته یا کاراکتر تکی هستیم یا نه.

۲. **حلقه اصلی:** در این بخش، یک حلقه بر روی تمام کاراکترهای محتوای فایل ایجاد می‌شود. هر کاراکتر به ترتیب بررسی می‌شود.

۳. **بررسی اولویت کاراکترها:** ابتدا بررسی می‌شود که آیا کاراکتر جاری جداکننده توکن است یا نه. اگر کاراکتر جداکننده بود و ما درون رشته یا کاراکتر تکی نبودیم، توکن فعلی را چاپ می‌کنیم و پاک می‌کنیم تا برای توکن بعدی آماده شود.

۴. **شناسایی نوع توکن:** اگر کاراکتر جداکننده نبود، توکن فعلی با این کاراکتر ادامه می‌یابد. همچنین بررسی می‌شود که آیا درون رشته یا کاراکتر تکی هستیم یا نه تا بتوانیم رشته‌ها و کاراکترهای تکی را تشخیص دهیم.

۵. **تمام شدن حلقه و پایان تابع:** حلقه تا زمانی ادامه پیدا می‌کند که تمام کاراکترهای محتوای فایل بررسی شوند. سپس تابع به پایان می‌رسد و عملکرد تمامی توکن‌ها و نشانه‌ها چاپ می‌شود.

فاز دوم – Parser

مقدمه

برنامه parser.cpp شامل یک تجزیه‌کننده (parser) برای یک زبان برنامه‌نویسی است. این تجزیه‌کننده کد منبع را به ساختاری به نام درخت نحو انتزاعی (AST - Abstract Syntax Tree) تجزیه می‌کند. این درخت نمایانگر ساختار نحوی کد است و به برنامه‌نویس یا ابزارهای دیگر اجازه می‌دهد تا معنای کد را تجزیه و تحلیل کنند.

اجزای اصلی برنامه

1. Header Files

```
#include <iostream>
#include <vector>
#include <string>
#include "Lexer.h"
```

این فایل‌ها کتابخانه‌های مورد نیاز و هدر فایل Lexer را شامل می‌شوند.

2. Enum `NodeType`

این enum انواع مختلف گره‌هایی که می‌توانند در AST ظاهر شوند را تعریف می‌کند. هر نوع گره نشان‌دهنده یک ساختار نحوی خاص در زبان برنامه‌نویسی است.

3. Struct Node

این struct یک گره در AST را نمایش می‌دهد و شامل:

- نوع گره (`NodeType type`)
- مقدار گره (`std::string value`)
- فرزندان گره (`std::vector<Node> children`)

4. Class `ASTPrinter`

این کلاس شامل توابعی برای چاپ AST است. این توابع به منظور نمایش درخت نحو به صورت خوانا و با فرورفتگی مناسب به کار می‌روند.

- `print(const Node &node, int indent = 0)`: چاپ یک گره و فرزندانش با فرورفتگی مناسب.
- `printIndent(int indent)`: تابع کمکی برای چاپ فاصله‌ها به منظور فرورفتگی.
- `nodeTypeToString(NodeType type)`: تبدیل یک `NodeType` به رشته‌ای برای چاپ.

۵. ساختار گرامری

گرامر زبان برنامه‌نویسی شامل قواعد نحوی است که ساختارهای مختلف کد را تعریف می‌کند. این ساختارها شامل برنامه، اعلان‌ها، عبارات، حلقه‌ها، شرط‌ها، و غیره می‌شود. این گرامر به parser اجازه می‌دهد تا کد ورودی را به درستی تجزیه و تحلیل کند.

نحوه کارکرد parser

۱. Lexer: ابتدا کد منبع را به توکن‌ها تجزیه می‌کند. هر توکن نمایانگر یک واحد نحوی از کد منبع است مانند کلمات کلیدی، شناسه‌ها، عملگرها و غیره.
۲. Parser: توکن‌های تولید شده توسط Lexer را می‌گیرد و آن‌ها را به ساختارهای نحوی مطابق با گرامر تبدیل می‌کند. این کار با استفاده از یک درخت نحو انتزاعی (AST) انجام می‌شود.
۳. AST: درخت نحوی که نمایانگر ساختار برنامه است و می‌تواند برای تجزیه و تحلیل بیشتر، بهینه‌سازی، و یا اجرای کد استفاده شود.

مزایا و کاربردها

- ❖ تحلیل کد: AST به ابزارهای تحلیل کد اجازه می‌دهد تا ساختار و معنای کد را بررسی کنند.
- ❖ تولید کد: از AST می‌توان برای تولید کد اجرایی یا ترجمه کد به زبان‌های دیگر استفاده کرد.
- ❖ بهینه‌سازی: AST به کامپایلرها اجازه می‌دهد تا بهینه‌سازی‌های مختلفی روی کد انجام دهند.

گرامر

Program → { statement };

statement → declaration | assignment | ifStatement | forLoop | printStatement | block | returnStatement | comment | jumpStatement;

declaration → type identifier [arraySize] [assignmentRHS] { "," identifier [arraySize] [assignmentRHS] };";"

assignment → identifier [arrayIndex] "=" expression;";"

ifStatement → "if" "(" expression ")" statement ["else" statement];

forLoop → "for" "(" (declaration | assignment | ";"), expression, ";", (assignment | expression) ")" statement;

printStatement → "print" "(" printArguments ");" "

block → "{" { statement } "};"

returnStatement → "return" expression;";"

comment → "comment;"

jumpStatement → "break" | "continue;"

expression → logicalOrExpression;

logicalOrExpression → logicalAndExpression { "||" logicalAndExpression };

logicalAndExpression → equalityExpression { "&&" equalityExpression };

equalityExpression → relationalExpression { ("==" | "!=") relationalExpression };

relationalExpression → additiveExpression { ("<" | ">" | "<=" | ">=") additiveExpression };

additiveExpression → multiplicativeExpression { ("+" | "-") multiplicativeExpression };

multiplicativeExpression → unaryExpression { ("*" | "/" | "%") unaryExpression };

unaryExpression → ["!"] primaryExpression;

primaryExpression → identifier | literal | "true" | "false" | "character" | "string" | "(" expression ");"

identifier → "identifier;"

arraySize → "[" literal "];"

arrayIndex → "[" expression "];"

assignmentRHS → "=" expression;

printArguments → expression { "," expression };

۱. برنامه (Program)

یک برنامه شامل یک دنباله‌ای از عبارات یا اعلان‌ها است.

۲. اعلان (Declaration)

اعلان‌ها شامل اعلان متغیر یا تابع هستند.

۳. عبارت (Statement)

عبارات شامل تخصیص‌ها، عبارات شرطی، حلقه‌ها، عبارات چاپ و عبارات کنترلی مانند break و continue هستند.

۴. تخصیص (Assignment)

تخصیص‌ها مقادیر را به متغیرها اختصاص می‌دهند.

۵. عبارت شرطی (IfStatement)

عبارات شرطی ساختارهای شرطی با شاخه‌ها را تعریف می‌کنند.

۶. حلقه (ForLoop)

حلقه‌ها ساختارهای تکراری را تعریف می‌کنند.

۷. چاپ (PrintStatement)

عبارات چاپ مقادیر را چاپ می‌کنند.

۸. بلاک (Block)

یک بلاک شامل یک دنباله‌ای از عبارات است که در یک بلاک (مثلاً { ... }) قرار گرفته‌اند.

۹. عبارت (Expression)

عبارات شامل عبارات ریاضی یا منطقی هستند.

۱۰. لیتریال (Literal)

لیتریال‌ها مقادیر ثابت یا لیتریال هستند.

۱۱. شناسه (Identifier)

شناسه‌ها نام متغیرها یا توابع هستند.

۱۲. عملگرهای یونری و باینری (Unary and Binary Operations)

عملگرهای یونری و باینری عملیات‌هایی با یک یا دو عملوند هستند.

۱۳. نظرها (Comments)

نظرها درون کد هستند.

۱۴. عبارات کنترلی (Jump Statements)

عبارات کنترلی شامل دستورات کنترلی مانند break یا continue هستند.

جدول تجزیه

رسم جداول تجزیه و بازیابی خطا برای این گرامر نیاز به تحلیل دقیقی از قواعد گرامری دارد. جداول تجزیه (Parsing Tables) معمولاً برای تجزیه‌کننده‌های LL(1) یا LR(1) رسم می‌شوند. برای این منظور، ابتدا باید قواعد گرامری را به فرم استاندارد بنویسیم و سپس جداول را رسم کنیم. در اینجا ما از تجزیه LL(1) استفاده کرده‌ایم، پس جداول تجزیه این نوع تجزیه را رسم می‌کنیم.

برای شروع، قواعد گرامری را به صورت دقیق و فرم استاندارد نوشتیم. در قدم بعد برای رسم جدول پیش‌بینی، ابتدا مجموعه‌های FIRST و FOLLOW را برای هر غیرترمینال محاسبه می‌کنیم.

۱. محاسبه مجموعه‌های FIRST

- **FIRST(Program) = { { }**
- **FIRST(statement) = { type, identifier, if, for, print, {, return, comment, break, continue }**
- **FIRST(declaration) = { type }**
- **FIRST(assignment) = { identifier }**
- **FIRST(ifStatement) = { if }**
- **FIRST(forLoop) = { for }**
- **FIRST(printStatement) = { print }**
- **FIRST(block) = { { }**
- **FIRST(returnStatement) = { return }**
- **FIRST(comment) = { comment }**
- **FIRST(jumpStatement) = { break, continue }**
- **FIRST(expression) = { identifier, literal, true, false, character, string, (, ! }**
- **FIRST(logicalOrExpression) = FIRST(expression)**

- **FIRST(logicalAndExpression) = FIRST(expression)**
- **FIRST(equalityExpression) = FIRST(expression)**
- **FIRST(relationalExpression) = FIRST(expression)**
- **FIRST(additiveExpression) = FIRST(expression)**
- **FIRST(multiplicativeExpression) = FIRST(expression)**
- **FIRST(unaryExpression) = { !, identifier, literal, true, false, character, string, (}**
- **FIRST(primaryExpression) = { identifier, literal, true, false, character, string, (}**
- **FIRST(identifier) = { identifier }**
- **FIRST(arraySize) = { [}**
- **FIRST(arrayIndex) = { [}**
- **FIRST(assignmentRHS) = { = }**
- **FIRST(printArguments) = { identifier, literal, true, false, character, string, (}**

۲. محاسبه مجموعه‌های **FOLLOW**

- **FOLLOW(Program) = { $ }**
- **FOLLOW(statement) = { type, identifier, if, for, print, {, return, comment, break, continue, }, $ }**
- **FOLLOW(declaration) = { type, identifier, if, for, print, {, return, comment, break, continue, }, $ }**
- **FOLLOW(assignment) = { type, identifier, if, for, print, {, return, comment, break, continue, }, $ }**
- **FOLLOW(ifStatement) = { type, identifier, if, for, print, {, return, comment, break, continue, }, $ }**
- **FOLLOW(forLoop) = { type, identifier, if, for, print, {, return, comment, break, continue, }, $ }**
- **FOLLOW(printStatement) = { type, identifier, if, for, print, {, return, comment, break, continue, }, $ }**
- **FOLLOW(block) = { type, identifier, if, for, print, {, return, comment, break, continue, }, $ }**
- **FOLLOW(returnStatement) = { type, identifier, if, for, print, {, return, comment, break, continue, }, $ }**
- **FOLLOW(comment) = { type, identifier, if, for, print, {, return, comment, break, continue, }, $ }**
- **FOLLOW(jumpStatement) = { type, identifier, if, for, print, {, return, comment, break, continue, }, $ }**
- **FOLLOW(expression) = {), ;, ,,] }**

- **FOLLOW(logicalOrExpression) = FOLLOW(expression)**
- **FOLLOW(logicalAndExpression) = { ||,), ;, ,,] }**
- **FOLLOW(equalityExpression) = { &&, ||,), ;, ,,] }**
- **FOLLOW(relationalExpression) = { ==, !=, &&, ||,), ;, ,,] }**
- **FOLLOW(additiveExpression) = { <, >, <=, >=, ==, !=, &&, ||,), ;, ,,] }**
- **FOLLOW(multiplicativeExpression) = { +, -, <, >, <=, >=, ==, !=, &&, ||,), ;, ,,] }**
- **FOLLOW(unaryExpression) = { *, /, %, +, -, <, >, <=, >=, ==, !=, &&, ||,), ;, ,,] }**
- **FOLLOW(primaryExpression) = FOLLOW(unaryExpression)**
- **FOLLOW(identifier) = { [, =, *, /, %, +, -, <, >, <=, >=, ==, !=, &&, ||,), ;, ,,] }**
- **FOLLOW(arraySize) = { ,, ; }**
- **FOLLOW(arrayIndex) = { = }**
- **FOLLOW(assignmentRHS) = { ,, ; }**
- **FOLLOW(printArguments) = {), ,,] }**

جدول تجزیه LL(۱)

به دلیل بزرگ بودن این جدول، ترسیم آن در این فایل مقدور نبود، لذا در فایلی جدا با نام LLdecompositionTable(1) که در این پوشه ضمیمه شده است، آورده شده.

جداول بازیابی خطا

برای بازیابی خطا، می‌توانیم از قواعد گرامری استفاده کنیم تا خطاهای متداول را شناسایی کنیم. برای مثال:

- **خطای تخصیص** :اگر بعد از **Identifier**، **=**نیامد، می‌توان پیام خطا **.Missing '=' in assignment**را نمایش داد.
- **خطای بلوک** :اگر بعد از **}**، بلوک به درستی بسته نشد، می‌توان پیام خطا **.Unclosed block**را نمایش داد.

جدول بازیابی خطا نیز مانند جدول تجزیه به دلیل بزرگی، در همان فایل الحاق شده است.

برای توضیح دقیق‌تر در مورد تمام قسمت‌ها و توابع برنامه parser.cpp، باید کد کامل را بررسی کنیم. به همین دلیل، ابتدا بخش‌های مختلف کد را معرفی کرده و سپس توضیح می‌دهیم. بخش‌هایی که در توضیحات قبلی نیامده‌اند را هم پوشش می‌دهیم.

توضیح هر تابع

1. Enum NodeType

   enum NodeType

```
{
  NODE_PROGRAM,
  NODE_STATEMENT,
  NODE_DECLARATION,
  NODE_ASSIGNMENT,
  NODE_IF_STATEMENT,
  NODE_FOR_LOOP,
  NODE_FUNCTION_DEFINITION,
  NODE_BLOCK,
  NODE_EXPRESSION,
  NODE_LITERAL,
  NODE_IDENTIFIER,
  NODE_PRINT_STATEMENT,
  NODE_UNARY_OP,
  NODE_BINARY_OP,
  NODE_COMMENT,
  NODE_JUMP_STATEMENT
};
```

این enum انواع گره‌هایی که می‌توانند در AST ظاهر شوند را تعریف می‌کند.

2. Struct Node

```
struct Node
{
  NodeType type;
  std::string value;
  std::vector<Node> children;
};
```

این struct یک گره در AST را نمایش می‌دهد. هر گره شامل نوع گره، مقدار گره و لیستی از گره‌های فرزند است.

3. Class ASTPrinter

```
class ASTPrinter
{
Public:
  void print(const Node &node, int indent = 0) const
  {
    printIndent(indent);
    std::cout << nodeTypeToString(node.type) << ": " << node.value << "\n";
    for (const Node &child : node.children)
    {
      print(child, indent + 2);
    }
  }

private:
  void printIndent(int indent) const
  {
    for (int i = 0; i < indent; ++i)
    {
      std::cout << ' ';
    }
  }

  std::string nodeTypeToString(NodeType type) const
  {
    switch (type)
    {
    case NODE_PROGRAM:
      return "Program;"
    case NODE_DECLARATION:
```

```
    return "Declaration;"
  case NODE_STATEMENT:
    return "Statement;"
  case NODE_EXPRESSION:
    return "Expression;"
  case NODE_IDENTIFIER:
    return "Identifier;"
  case NODE_LITERAL:
    return "Literal;"
  case NODE_PRINT_STATEMENT:
    return "PrintStatement;"
  case NODE_IF_STATEMENT:
    return "IfStatement;"
  case NODE_FOR_LOOP:
    return "ForLoop;"
  case NODE_BLOCK:
    return "Block;"
  case NODE_ASSIGNMENT:
    return "Assignment;"
  case NODE_BINARY_OP:
    return "BinaryOp;"
  case NODE_UNARY_OP:
    return "UnaryOp;"
  case NODE_COMMENT:
    return "Comment;"
  case NODE_JUMP_STATEMENT:
    return "Jump;"
  default:
    return "Unknown;"
  }
}
```

```
};
```

توابع کلاس ASTPrinter

- print(const Node &node, int indent = 0) : این تابع یک گره و فرزندان آن را با فرورفتگی مناسب چاپ می‌کند.
- printIndent(int indent) : این تابع به تعداد indent فاصله چاپ می‌کند تا فرورفتگی مناسب را ایجاد کند.
- nodeTypeToString(NodeType type) : این تابع نوع گره را به رشته‌ای تبدیل می‌کند که می‌تواند چاپ شود.

1. Constructor and Destructor

```
class Parser {
public:
  Parser(Lexer &lexer) : lexer(lexer) {}

private:
  Lexer &lexer;
};
```

2. Parse Functions

```
Node parseProgram();
Node parseStatement();
Node parseDeclaration();
Node parseAssignment();
Node parseIfStatement();
Node parseForLoop();
Node parsePrintStatement();
Node parseBlock();
Node parseExpression();
Node parseLiteral();
Node parseIdentifier();
Node parseUnaryOp();
Node parseBinaryOp();
```

```
Node parseComment();
Node parseJumpStatement();
```

هر یک از این توابع بخشی از کد را مطابق با قواعد گرامری تجزیه می‌کند و یک گره AST تولید می‌کند.

توضیحات توابع :

2. parseProgram

این تابع برنامه را تجزیه کرده و یک گره Program ایجاد می‌کند.

```
Node parseProgram(){
  Node programNode;
  programNode.type = NODE_PROGRAM;
  while (lexer.hasNext()) {
    if (isDeclaration()) {
      programNode.children.push_back(parseDeclaration());
    } else }
      programNode.children.push_back(parseStatement());
    }
  }
  return programNode;
}
```

3. parseStatement :

این تابع یک عبارت را تجزیه می‌کند و یک گره Statement ایجاد می‌کند.

```
Node parseStatement(){
    //بررسی نوع عبارت و فراخوانی تابع مناسب برای تجزیه
}
```

4. parseDeclaration :

این تابع یک اعلان را تجزیه می‌کند و یک گره Declaration ایجاد می‌کند.

```
Node parseDeclaration(){
    //تجزیه اعلان متغیر یا تابع
}
```

5. parseAssignment :

این تابع یک تخصیص را تجزیه می‌کند و یک گره Assignment ایجاد می‌کند.

```
Node parseAssignment(){
```

```
    //تجزیه تخصیص
}
```

6. parseIfStatement :

این تابع یک عبارت شرطی را تجزیه می‌کند و یک گره IfStatement ایجاد می‌کند.

```
Node parseIfStatement(){
    //تجزیه عبارت شرطی
}
```

7. parseForLoop :

این تابع یک حلقه for را تجزیه می‌کند و یک گره ForLoop ایجاد می‌کند.

```
Node parseForLoop(){
   //for تجزیه حلقه
}
```

8. parsePrintStatement :

این تابع یک عبارت چاپ را تجزیه می‌کند و یک گره PrintStatement ایجاد می‌کند.

```
Node parsePrintStatement(){
    //تجزیه عبارت چاپ
}
```

9. parseBlock :

این تابع یک بلوک کد را تجزیه می‌کند و یک گره Block ایجاد می‌کند.

```
Node parseBlock(){
    //تجزیه بلوک کد
}
```

10. parseExpression :

این تابع یک عبارت را تجزیه می‌کند و یک گره Expression ایجاد می‌کند.

```
Node parseExpression(){
```

```
    تجزیه عبارت//
}
```

11. parseLiteral :

این تابع یک مقدار لیتریال را تجزیه می‌کند و یک گره Literal ایجاد می‌کند.

```
Node parseLiteral(){
    تجزیه مقدار لیتریال//
}
```

12. parseIdentifier :

این تابع یک شناسه را تجزیه می‌کند و یک گره Identifier ایجاد می‌کند.

```
Node parseIdentifier(){
    تجزیه شناسه//
}
```

13. parseUnaryOp :

این تابع یک عملگر یوناری را تجزیه می‌کند و یک گره `UnaryOp` ایجاد می‌کند.

```
Node parseUnaryOp(){
    تجزیه عملگر یوناری//
}
```

14. parseBinaryOp :

این تابع یک عملگر باینری را تجزیه می‌کند و یک گره BinaryOp ایجاد می‌کند.

```
Node parseBinaryOp(){
    تجزیه عملگر باینری//
}
```

15. parseComment :

این تابع یک نظر را تجزیه می‌کند و یک گره Comment ایجاد می‌کند.

```
Node parseComment() {
    تجزیه نظر//
}
```

16. parseJumpStatement :

این تابع یک عبارت کنترلی را تجزیه می‌کند و یک گره JumpStatement ایجاد می‌کند.

```
Node parseJumpStatement() {
    تجزیه عبارت کنترلی//
}
```

نتیجه‌گیری

برنامه parser.cpp یک جزء حیاتی از یک کامپایلر یا مفسر است که وظیفه تجزیه و تحلیل نحوی کد منبع را بر عهده دارد. با استفاده از ساختارهای تعریف شده مانند Node و ASTPrinter، این برنامه کد ورودی را به یک ساختار درختی تجزیه می‌کند که می‌تواند برای مراحل بعدی مانند تحلیل معنایی، بهینه‌سازی، و تولید کد استفاده شود.

فاز سوم – Semantic

مقدمه

در فرآیند کامپایل کردن یک برنامه، پس از تحلیل نحوی (Syntax Analysis) که ساختار گرامری کد منبع را بررسی می‌کند، مرحله‌ی تحلیل معنایی (Semantic Analysis) قرار دارد. تحلیل معنایی مسئول بررسی سازگاری و معنای کد برنامه است و اطمینان حاصل می‌کند که کد با قوانین معنایی زبان برنامه‌نویسی مطابقت دارد. این مرحله شامل بررسی اعلان متغیرها، تخصیص مقادیر، بررسی نوع داده‌ها، و اطمینان از صحت استفاده از توابع و سایر ساختارهای برنامه است.

کلاس SymbolTable

این کلاس یک جدول نماد را برای نگهداری نمادهای مختلف (متغیرها، توابع و غیره) در طول برنامه مدیریت می‌کند.

متغیرهای کلاس

- table: یک وکتور از unordered_map که هر نقشه نمادها را برای یک سطح محدوده خاص نگه می‌دارد.
- currentScope: سطح فعلی محدوده در حال پردازش.

توابع کلاس

- SymbolTable(): سازنده که سطح محدوده اولیه را ایجاد می‌کند.
- enterScope(): برای ورود به یک سطح محدوده جدید.
- exitScope(): برای خروج از یک سطح محدوده.
- insert(const std::string &name, const std::string &type): برای وارد کردن یک نماد جدید به جدول در سطح محدوده فعلی.
- lookup(const std::string &name): برای جستجوی نمادی با نام داده شده در تمامی سطوح محدوده از بالاترین سطح به پایین‌ترین سطح.

کلاس SemanticAnalyzer

این کلاس برای تحلیل معنایی درخت نحوی تولید شده توسط پارسر استفاده می‌شود. تحلیل معنایی شامل بررسی متغیرهای اعلام شده، اطمینان از وجود تابع main و بررسی تخصیص‌ها و دستورات شرطی است.

متغیرهای کلاس

- symbolTable: یک نمونه از کلاس SymbolTable برای نگهداری نمادها.
- Errors: یک وکتور از رشته‌ها برای ذخیره خطاهای معنایی.

توابع کلاس

`analyze(const Node &root):`

این تابع تحلیل معنایی را آغاز می‌کند. ابتدا وجود تابع main را بررسی می‌کند و سپس به طور بازگشتی تمام گره‌های درخت نحوی را برای تحلیل معنایی بررسی می‌کند.

`checkMainFunction(const Node &root):`

این تابع بررسی می‌کند که آیا تابع main در برنامه وجود دارد یا خیر. اگر وجود نداشته باشد، یک خطای معنایی گزارش می‌دهد و برنامه را متوقف می‌کند.

`checkNode(const Node &node):`

این تابع بسته به نوع گره (node.type)، عمل مربوطه را انجام می‌دهد. برای مثال، اگر گره از نوع NODE_DECLARATION باشد، تابع checkDeclaration را فراخوانی می‌کند. این تابع به صورت بازگشتی بر روی تمام گره‌های فرزند نیز اعمال می‌شود.

`checkDeclaration(const Node &node):`

این تابع بررسی می‌کند که آیا متغیر در همان سطح محدوده از قبل اعلام شده است یا خیر. اگر چنین باشد، خطا گزارش می‌شود. در غیر این صورت، متغیر به جدول نماد اضافه می‌شود.

`checkArrayDeclaration(const Node &node):`

این تابع اعلام آرایه را بررسی می‌کند. ابتدا اندازه آرایه را بررسی می‌کند و در صورتی که اندازه معتبر باشد، تمامی عناصر آرایه را به جدول نماد اضافه می‌کند. اگر اندازه آرایه نامعتبر باشد، خطا گزارش می‌شود.

`checkAssignment(const Node &node):`

این تابع تخصیص متغیر را بررسی می‌کند. ابتدا بررسی می‌کند که آیا متغیر اعلام شده است یا خیر. اگر اعلام نشده باشد، خطا گزارش می‌شود. در صورت تخصیص به آرایه، نوع و محدوده ایندکس نیز بررسی می‌شود.

`checkIfStatement(const Node &node):`

این تابع بررسی می‌کند که شرط دستور if یک مقدار منطقی (bool) است. سپس بدنه if و بلوک else (در صورت وجود) را بررسی می‌کند.

`checkForLoop(const Node &node):`

این تابع یک حلقه for را بررسی می‌کند. ابتدا یک سطح محدوده جدید ایجاد می‌کند، سپس قسمت‌های مختلف حلقه را بررسی می‌کند و در نهایت از سطح محدوده خارج می‌شود.

`checkIdentifier(const Node &node):`

این تابع بررسی می‌کند که آیا متغیر قبل از استفاده اعلام شده است یا خیر. اگر متغیر اعلام نشده باشد، خطا گزارش می‌شود.

`reportError(const std::string &error):`

این تابع خطاهای معنایی را ثبت می‌کند. در نهایت، اگر خطاها وجود داشته باشند، آن‌ها را نمایش می‌دهد و برنامه را متوقف می‌کند.

کلاس TypeChecker

این کلاس وظیفه بررسی نوع‌ها در برنامه را بر عهده دارد. شامل بررسی نوع متغیرها، آرایه‌ها، توابع و عملیات‌های مختلف است.

متغیرهای کلاس:

- symbolTable: یک unordered_map برای نگهداری نوع متغیرها.
- functionTable: یک unordered_map برای نگهداری اطلاعات توابع.

توابع کلاس:

`check(const Node &node):`

این تابع بررسی نوع را آغاز می‌کند و بسته به نوع گره (node.type) ، تابع مربوطه را فراخوانی می‌کند. به صورت بازگشتی بر روی تمام گره‌های فرزند نیز اعمال می‌شود.

`checkDeclaration(const Node &node):`

این تابع بررسی می‌کند که آیا نوع متغیری که اعلام می‌شود با نوع عبارت مقداردهی اولیه مطابقت دارد یا خیر. اگر تطابق نداشته باشد، خطا گزارش می‌شود.

`checkArrayDeclaration(const Node &node):`

این تابع اعلام آرایه را بررسی می‌کند. اندازه آرایه باید معتبر باشد و نوع آرایه باید با نوع عناصر مطابقت داشته باشد. در غیر این صورت، خطا گزارش می‌شود.

`checkAssignment(const Node &node):`

این تابع تخصیص متغیر را بررسی می‌کند. نوع متغیر و نوع عبارت تخصیص باید مطابقت داشته باشند. اگر مطابقت نداشته باشند، خطا گزارش می‌شود.

`checkFunctionDefinition(const Node &node):`

این تابع تعریف تابع را بررسی می‌کند. نام تابع، نوع بازگشتی و نوع پارامترها را ثبت می‌کند و سپس بدنه تابع را بررسی می‌کند تا تطابق نوع‌ها را بررسی کند.

`checkFunctionCall(const Node &node):`

این تابع فراخوانی تابع را بررسی می‌کند. نام تابع باید در جدول توابع ثبت شده باشد و تعداد و نوع پارامترها باید با تعریف تابع مطابقت داشته باشند. اگر مطابقت نداشته باشند، خطا گزارش می‌شود.

`checkPrintStatement(const Node &node):`

این تابع دستور چاپ را بررسی می‌کند. برای هر عبارت چاپی، نوع آن را بررسی می‌کند.

`checkIfStatement(const Node &node):`

این تابع بررسی می‌کند که شرط دستور if یک مقدار منطقی (bool) باشد. سپس بدنه if و بلوک else (در صورت وجود) را بررسی می‌کند.

`checkForLoop(const Node &node):`

این تابع یک حلقه for را بررسی می‌کند. شرط حلقه باید یک مقدار منطقی (bool) باشد و سپس قسمت‌های مختلف حلقه را بررسی می‌کند.

`checkBlock(const Node &node):`

این تابع یک بلوک کد را بررسی می‌کند. تمامی دستورات درون بلوک را به ترتیب بررسی می‌کند.

`checkReturnStatement(const Node &node):`

این تابع بررسی می‌کند که نوع عبارت بازگشتی با نوع بازگشتی تعریف شده برای تابع مطابقت داشته باشد.

`checkExpression(const Node &node):`

این تابع نوع یک عبارت را بررسی می‌کند. بسته به نوع گره (node.type)، تابع مربوطه را فراخوانی می‌کند و نوع عبارت را بازمی‌گرداند.

`checkBinaryOp(const Node &node):`

این تابع عملیات دودویی را بررسی می‌کند. نوع دو عملوند باید مطابقت داشته باشد و در غیر این صورت، خطا گزارش می‌شود. نوع عملیات را بازمی‌گرداند.

`checkUnaryOp(const Node &node):`

این تابع عملیات یک‌گانه را بررسی می‌کند. نوع عملوند را بررسی می‌کند و نوع عملیات را بازمی‌گرداند.

`getLiteralType(const std::string &value):`

این تابع نوع یک مقدار ثابت را تشخیص می‌دهد و نوع آن را برمی‌گرداند. برای مثال، اعداد به عنوان int، مقادیر منطقی به عنوان bool و رشته‌ها به عنوان string تشخیص داده می‌شوند.

`inferType(const Node &node):`

این تابع نوع یک گره را استنباط می‌کند. بسته به نوع گره (node.type)، تابع مربوطه را فراخوانی می‌کند و نوع گره را بازمی‌گرداند.

`error(const std::string &message):`

این تابع خطاهای نوع را گزارش می‌دهد. پیام خطا را چاپ می‌کند و برنامه را متوقف می‌کند.