گزارش فاز اول پروژه کامپایلر

گردآورندگان:

کیارش کوثری ، امیرحسین کنعانی ، آیلین نخستین

لینک پروژه

قواعد زبان

```
compilationUnit
: "MODULE" identifier ";" ( import )* block identifier "." ;
Import : ( "FROM" identifier )? "IMPORT" identList ";" ;
Block
: ( declaration )* ( "begin" statementSequence )? "end" ;
//////
declaration
: "int" *;
//////
variableDeclaration : identList ":" equalOps ;
equalOps : identifier ( "." identifier )* ;
identList : identifier ( "," identifier)* ;
/////
formalParameters
: "(" ( formalParameterList )? ")" ( ":" equalOps )? ;
formalParameterList
: formalParameter (";" formalParameter )* ;
formalParameter : ( "int" )? identList ":" equalOps ;
//////
statementSequence
: statement ( ";" statement )* ;
//////
statement
```

: equalOps ("+=” | “-=” | “/=” | “*= " expression | ("(" (expList)? ")")?)

| ifStatement | loopStatement ;

//////

ifStatement

: "if" expression "begin" statementSequence

(“elif”statementSequence * |"else" statementSequence)? "end" ;

//////

loopStatement

: "loop" expression "begin" statementSequence "end" ;

//////

expList

: expression ("," expression)* ;

expression

: simpleExpression (relation simpleExpression)? ;

relation

: "=" | "<" | "<=" | ">" | ">=" ;

simpleExpression

: ("+" | "-")? term (addOperator term)* ;

addOperator

: "+" | "-" | "or" ;

term

: power (mulOperator power)* ;

mulOperator

: "*" | "/" | "div" | "mod" | "and" ;

: factor (powerOperator factor)* ;

powerOperator

:”^”;

factor

: integer_literal | "(" expression ")" | "not" factor

| equalOps ("(" (expList)? ")")? ;

تحلیل لغوی

ابتدا در فایل $TokenKinds.def$ انواع $token$ها را مشخص می‌کنیم و در فایل $Token.h$ در کلاس $Token$ توابع زیر را پیاده‌سازی می‌کنیم:

- $getKind$: نوع $token$ را برمی‌گرداند.
- $getName$: خود $token$ را برمی‌گرداند.
- is: تست می‌کند آیا یک $token$ از جنس مشخص است یا نه.
- $isOneOf$: مانند تابع بالا، اما چندتا $token$ را چک می‌کند.

کلاس $Lexer$ توابع $next$ و $formToken$ را دارد که $next$ توکن بعدی را می‌دهد و $formToken$ توکن مورد نظر را می‌سازد.

در کلاس $Lexer.cpp$ متد $next$ را پیاده‌سازی می‌کنیم، به این صورت که $white\ space$ها را در نظر نمی‌گیریم و چک می‌کنیم اگر به $char$ رسیده باشیم چک می‌کنیم کلمات کلیدی هستند یا نه و توکن‌های مناسب را می‌سازیم و همچنین اعداد و عملگرها را نیز چک می‌کنیم و در آخر با $formToken$ توکن‌های مناسب را تولید می‌کنیم.

تولید پارسر و تحلیل نحوی

این بخش از کامپایلر در فایل‌های $Parser.h$ و $Parser.cpp$ پیاده‌سازی شده است. ابتدا ساختار درخت AST را توضیح می‌دهیم و در ادامه چگونگی ساختن درخت توسط پارسر بیان می‌شود.

- **parsePower** : این تابع یک عملیات توان عاملی را تجزیه می کند. این نحو را برای توان وعبارت حاصل را به اشاره گر Expr اختصاص می دهد.

- **parseBlock** : این تابع وظیفه تجزیه یک بلوک کد را بر عهده دارد. به عنوان ورودی لیست اعلامیه ها را می گیرد (DeclList) و عبارات (StmtList) و آنها را بر اساس آن تجزیه می کند.

- **parseDeclaration** : این تابع تجزیه یک اعلان کلی را انجام می دهد. نحو اعلان را تجزیه می کند و اعلان حاصل را به DeclList اضافه می کند.

- **parseVariableDeclaration** : این تابع وظیفه تجزیه اعلان های متغیر را بر عهده دارد. اعلان متغیر را تجزیه می کند نحو و اعلان حاصل را به DeclList اضافه می کند.

- **parseStatementSequence** : این تابع دنباله ای از عبارات را تجزیه می کند. فهرستی از عبارات (StmtList) و هر عبارت را در دنباله تجزیه می کند.

- **parseStatement** : این تابع تجزیه یک عبارت واحد را انجام می دهد. سینتکس یک عبارت و را تجزیه می کند و عبارت حاصل را به StmtList اضافه می کند.

- **parseIfStatement** : این تابع به طور خاص با تجزیه عبارات if سروکار دارد. این دستور دستور if را تجزیه می کند و عبارت حاصل را به StmtList اضافه می کند

- **parseLoopStatement** : این تابع وظیفه تجزیه عبارات while را بر عهده دارد. سینتکس مدتی را تجزیه می کند عبارت و عبارت حاصل را به StmtList اضافه می کند.
- **parseExpList** : این تابع لیستی از عبارات را تجزیه می کند. یک ExprList می گیرد و هر عبارت را در آن تجزیه می کند
- **parseExpression** : این تابع تجزیه یک عبارت را انجام می دهد. سینتکس یک عبارت و را تجزیه می کندعبارت حاصل را به اشاره گر Expr اختصاص می دهد.

.

پس از ساخت درخت باید آن را پیمایش کنیم.

تحلیل معنایی

این بخش از کامپایلر در فایل‌های $Sema.h$ و $Sema.cpp$ پیاده‌سازی شده‌است. پس از ساخت درخت می‌توانیم با پیمایش آن از درستی معنایی ورودی مطمئن شویم. برای این کار نودهایی که امکان دارند در آن‌ها اشتباهات منطقی رخ دهد را بررسی می‌کنیم. بقیه نودها نیز صرفا فرزندان خود را فراخوانی می‌کنند. اشتباهات منطقی محتمل در زبان ما عبارتند از:

- استفاده از متغیر قبل از تعریف آن
- تعریف متغیر با نام تکراری
- تقسیم بر صفر

در این مرحله این موارد را بررسی می‌کنیم و در صورت وجود چنین اشتباهاتی خطای $semantic\ error$ برمی‌گردانیم. برای بررسی متغیرها نیاز داریم که متغیرهای تعریف شده تا کنون را در حافظه ذخیره کنیم. به همین دلیل یک $StringSet$ برای این کار در نظر می‌گیریم. هر متغیری که در نود $Declaration$ تعریف می‌شود، در این مجموعه هم اضافه می‌شود. به این صورت در عبارات بعدی می‌توان وجود این اسم را بررسی کرد. در نود $Factor$ ، اگر تایپ برابر $ident$ باشد به معنی استفاده از یک متغیر است. پس باید متغیر نام برده شده در مجموعه ما وجود داشته باشد. اگر تابع $find$ مقدار $false$ برگرداند، به معنی عدم وجود نام متغیر است و باید خطای $semantic$ ایجاد شود. همین بررسی را در نود $Deceleration$ نیز انجام می‌دهیم. این بار در صورت وجود نام متغیر در مجموعه باید خطای $semantic$ ایجاد کنیم.

اشتباه تقسیم بر صفر ممکن در نود $BinaryOp$ اتفاق بیافتد. در این نود ابتدا بررسی می‌کنیم که آیا عملیات گفته شده از نوع تقسیم است یا خیر. سپس نیاز داریم که سمت راست عملیات را به نود $Factor$ تبدیل کنیم. در صورت موفق نبودن عملیات تبدیل، مقدار متغیر f برابر $NULL$ خواهد شد. سپس در صورت $NULL$ نبودن، نوع $Factor$ را بررسی می‌کنیم. اگر نوع $Factor$ برابر $Number$ بود به معنی تقسیم برا یک عدد ثابت است. سپس مقدار $Factor$ را استخراج می‌کنیم. اگر این مقدار برابر صفر باشد، یعنی یک خطای تقسیم بر صفر رخ داده است.