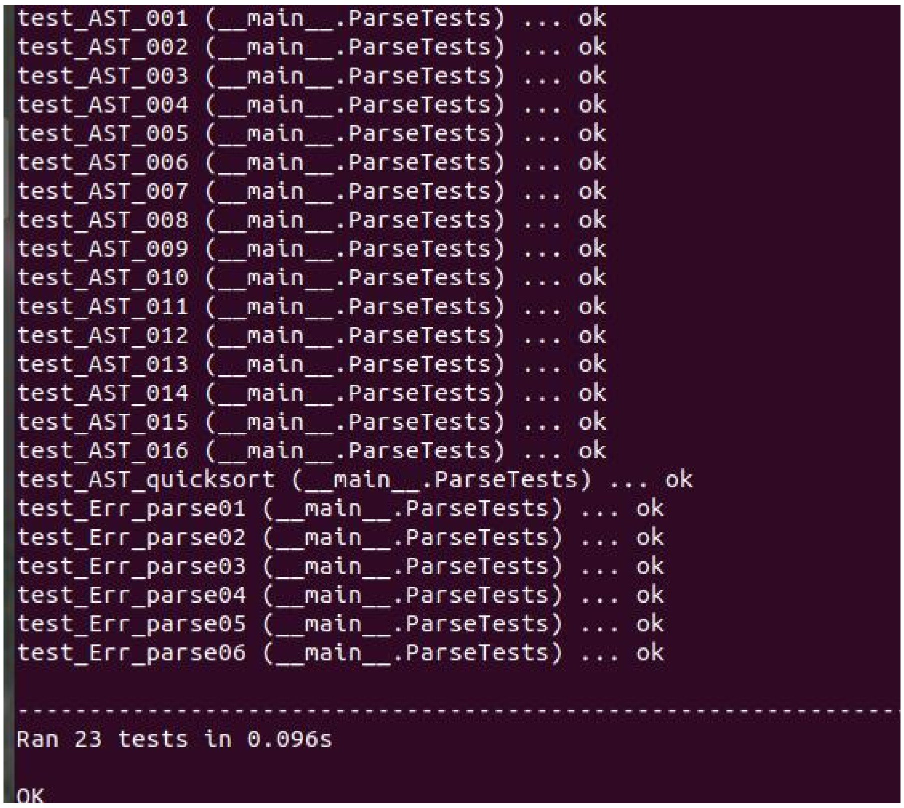
CSE 504 Compiler Design PA-01 Report
Name: Enbo Yu
ID: 113094714
Date: 17 SEP

I. Test Result



```
test_AST_001 (__main__.ParseTests) ... ok
test_AST_002 (__main__.ParseTests) ... ok
test_AST_003 (__main__.ParseTests) ... ok
test_AST_004 (__main__.ParseTests) ... ok
test_AST_005 (__main__.ParseTests) ... ok
test_AST_006 (__main__.ParseTests) ... ok
test_AST_007 (__main__.ParseTests) ... ok
test_AST_008 (__main__.ParseTests) ... ok
test_AST_009 (__main__.ParseTests) ... ok
test_AST_010 (__main__.ParseTests) ... ok
test_AST_011 (__main__.ParseTests) ... ok
test_AST_012 (__main__.ParseTests) ... ok
test_AST_013 (__main__.ParseTests) ... ok
test_AST_014 (__main__.ParseTests) ... ok
test_AST_015 (__main__.ParseTests) ... ok
test_AST_016 (__main__.ParseTests) ... ok
test_AST_quicksort (__main__.ParseTests) ... ok
test_Err_parse01 (__main__.ParseTests) ... ok
test_Err_parse02 (__main__.ParseTests) ... ok
test_Err_parse03 (__main__.ParseTests) ... ok
test_Err_parse04 (__main__.ParseTests) ... ok
test_Err_parse05 (__main__.ParseTests) ... ok
test_Err_parse06 (__main__.ParseTests) ... ok

----------------------------------------------------------------------
Ran 23 tests in 0.096s

OK
```

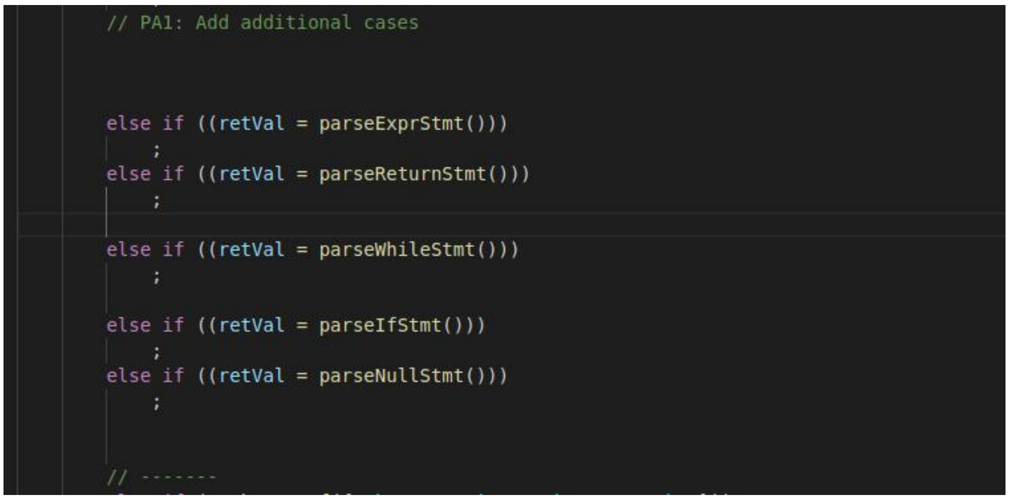
Ran 23 tests
OK

II. Class Details

<u>ParseStmt.cpp</u>

i. parseStmt

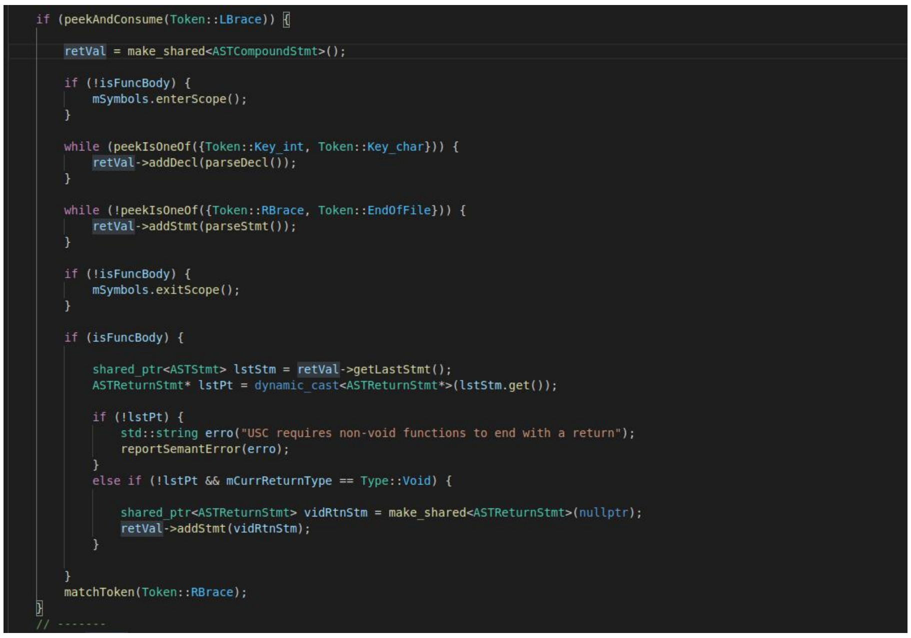
Some additional cases are added here, like parseExprStmt(), pareseReturnStmt(), parseWhileStmt(), etc, to make sure the following function can implement normally. And there is one issues that the part of 'parseNullStmt' must be put on the tail of the 'else if' otherwise the compiler will show two failure in 'test_AST_016' and 'test_AST_quicksort'.



```
// PA1: Add additional cases


else if ((retVal = parseExprStmt()))
    ;
else if ((retVal = parseReturnStmt()))
    ;

else if ((retVal = parseWhileStmt()))
    ;

else if ((retVal = parseIfStmt()))
    ;
else if ((retVal = parseNullStmt()))
    ;


// -------
```

ii. parseCompoundStmt

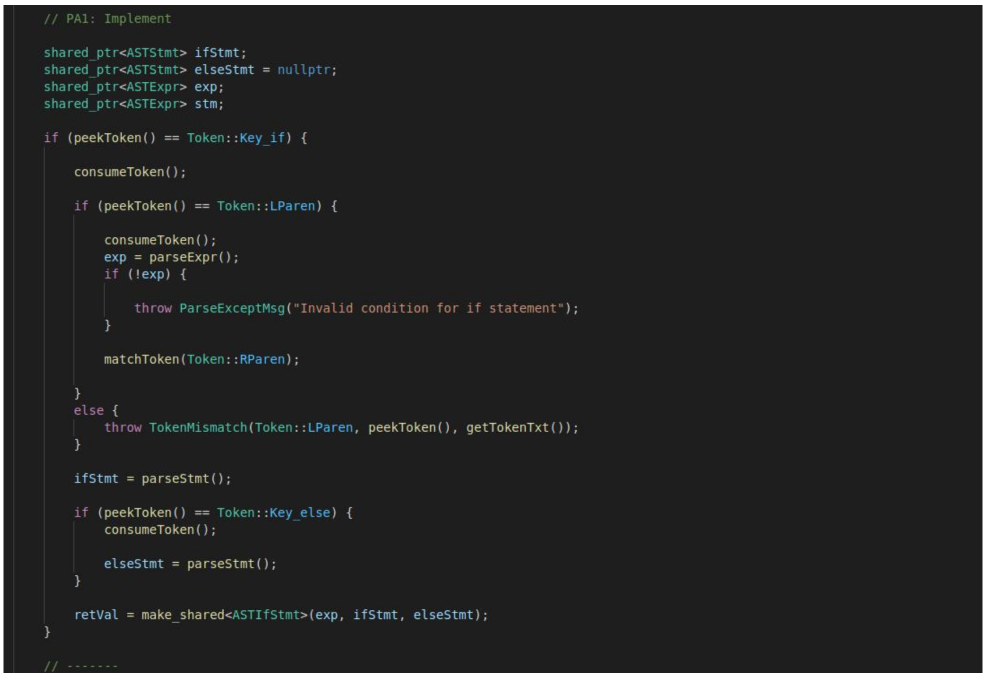
This part is used to parse and search other declarations and statements of sentences which will be compiled.



```
if (peekAndConsume(Token::LBrace)) {

    retVal = make_shared<ASTCompoundStmt>();

    if (!isFuncBody) {
        mSymbols.enterScope();
    }

    while (peekIsOneOf({Token::Key_int, Token::Key_char})) {
        retVal->addDecl(parseDecl());
    }

    while (!peekIsOneOf({Token::RBrace, Token::EndOfFile})) {
        retVal->addStmt(parseStmt());
    }

    if (!isFuncBody) {
        mSymbols.exitScope();
    }

    if (isFuncBody) {

        shared_ptr<ASTStmt> lstStm = retVal->getLastStmt();
        ASTReturnStmt* lstPt = dynamic_cast<ASTReturnStmt*>(lstStm.get());

        if (!lstPt) {
            std::string erro("USC requires non-void functions to end with a return");
            reportSemantError(erro);
        }
        else if (!lstPt && mCurrReturnType == Type::Void) {

            shared_ptr<ASTReturnStmt> vidRtnStm = make_shared<ASTReturnStmt>(nullptr);
            retVal->addStmt(vidRtnStm);
        }

    }
    matchToken(Token::RBrace);
}
// -------
```

iii. parseIfStmt

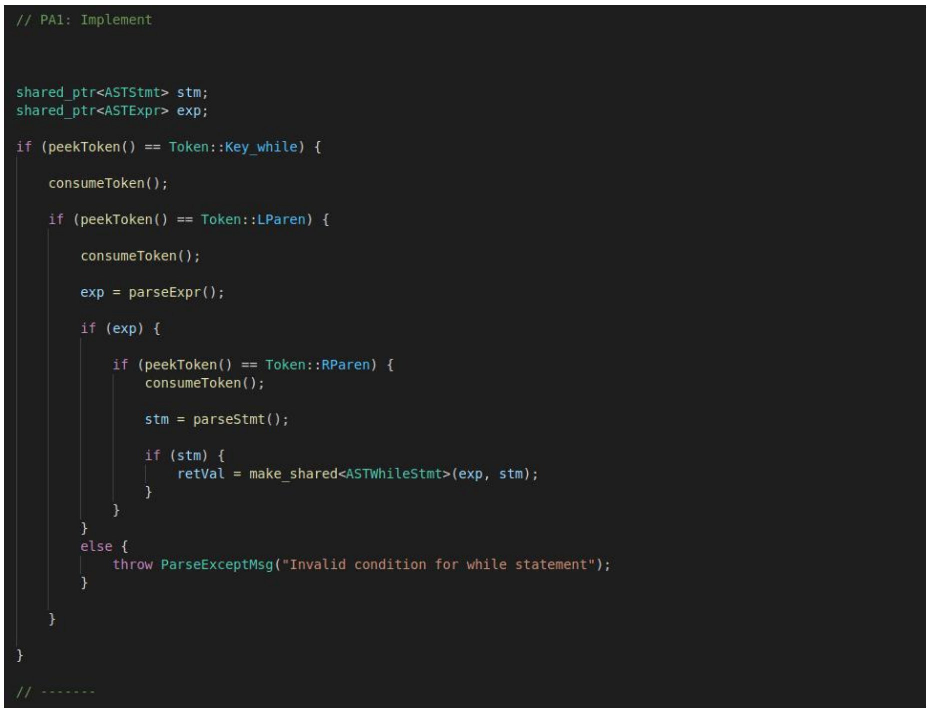
This part is used to parse ‘if then’ and ‘else’ statement.



```
// PA1: Implement

shared_ptr<ASTStmt> ifStmt;
shared_ptr<ASTStmt> elseStmt = nullptr;
shared_ptr<ASTExpr> exp;
shared_ptr<ASTExpr> stm;

if (peekToken() == Token::Key_if) {

    consumeToken();

    if (peekToken() == Token::LParen) {

        consumeToken();
        exp = parseExpr();
        if (!exp) {

            throw ParseExceptMsg("Invalid condition for if statement");
        }

        matchToken(Token::RParen);

    }
    else {
        throw TokenMismatch(Token::LParen, peekToken(), getTokenTxt());
    }

    ifStmt = parseStmt();

    if (peekToken() == Token::Key_else) {
        consumeToken();

        elseStmt = parseStmt();
    }

    retVal = make_shared<ASTIfStmt>(exp, ifStmt, elseStmt);
}

// -------
```

iv. parseWhileStmt

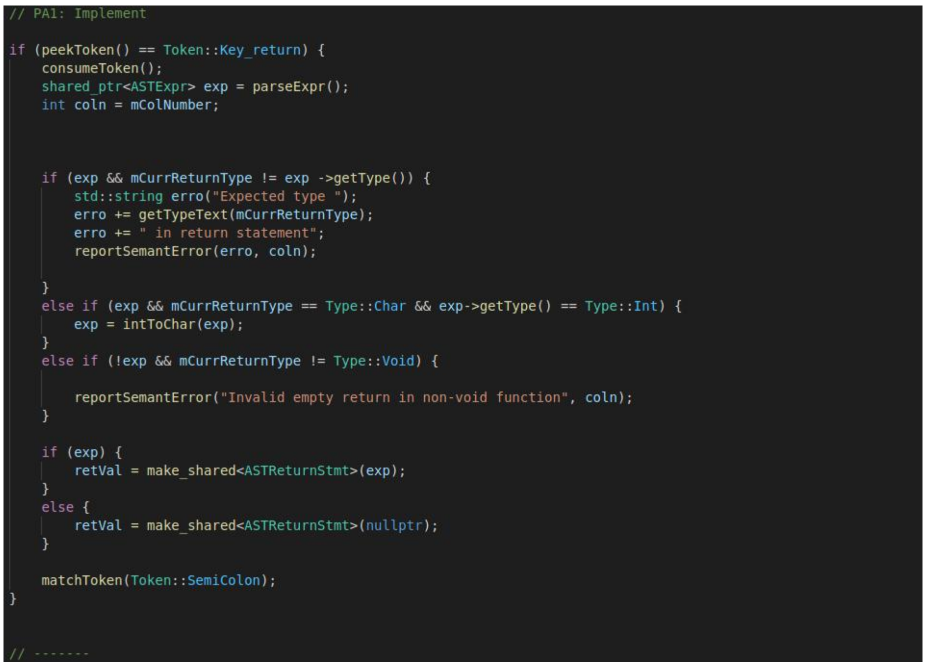
This part is used to compiler ‘while’ statement, which can pass test010. Implement parseWhileStmt and update parseStmt so that it also checks for while loops.



```
// PA1: Implement


shared_ptr<ASTStmt> stm;
shared_ptr<ASTExpr> exp;

if (peekToken() == Token::Key_while) {

    consumeToken();

    if (peekToken() == Token::LParen) {

        consumeToken();

        exp = parseExpr();

        if (exp) {

            if (peekToken() == Token::RParen) {
                consumeToken();

                stm = parseStmt();

                if (stm) {
                    retVal = make_shared<ASTWhileStmt>(exp, stm);
                }
            }
        }
        else {
            throw ParseExceptMsg("Invalid condition for while statement");
        }

    }

}

// -------
```

v. parseReturnStmt

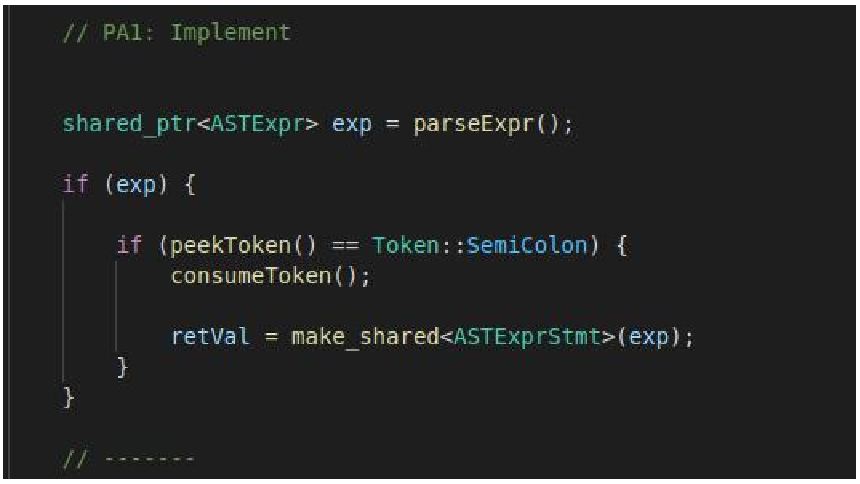
This part is used to parse ‘return’ statement.



```
// PA1: Implement

if (peekToken() == Token::Key_return) {
    consumeToken();
    shared_ptr<ASTExpr> exp = parseExpr();
    int coln = mColNumber;



    if (exp && mCurrReturnType != exp ->getType()) {
        std::string erro("Expected type ");
        erro += getTypeText(mCurrReturnType);
        erro += " in return statement";
        reportSemantError(erro, coln);

    }
    else if (exp && mCurrReturnType == Type::Char && exp->getType() == Type::Int) {
        exp = intToChar(exp);
    }
    else if (!exp && mCurrReturnType != Type::Void) {

        reportSemantError("Invalid empty return in non-void function", coln);
    }

    if (exp) {
        retVal = make_shared<ASTReturnStmt>(exp);
    }
    else {
        retVal = make_shared<ASTReturnStmt>(nullptr);
    }

    matchToken(Token::SemiColon);
}


// -------
```

vi. parseExprStmt

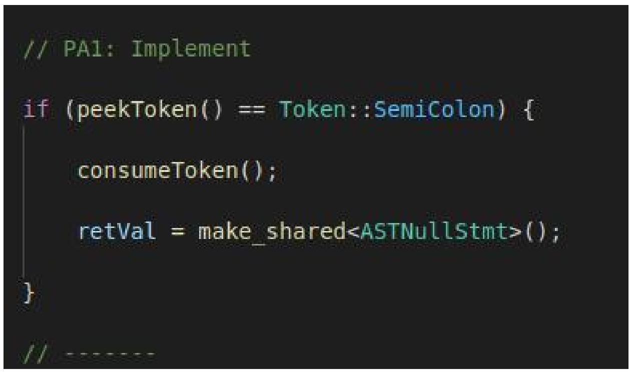
This part is used to parse Expression statement, hook it up to parseStmt.



```
    // PA1: Implement


    shared_ptr<ASTExpr> exp = parseExpr();

    if (exp) {

        if (peekToken() == Token::SemiColon) {
            consumeToken();

            retVal = make_shared<ASTExprStmt>(exp);
        }
    }

    // -------
```

vii. parseNullStmt

This part is used to parse Nulll statement, hook it up to parseStmt.



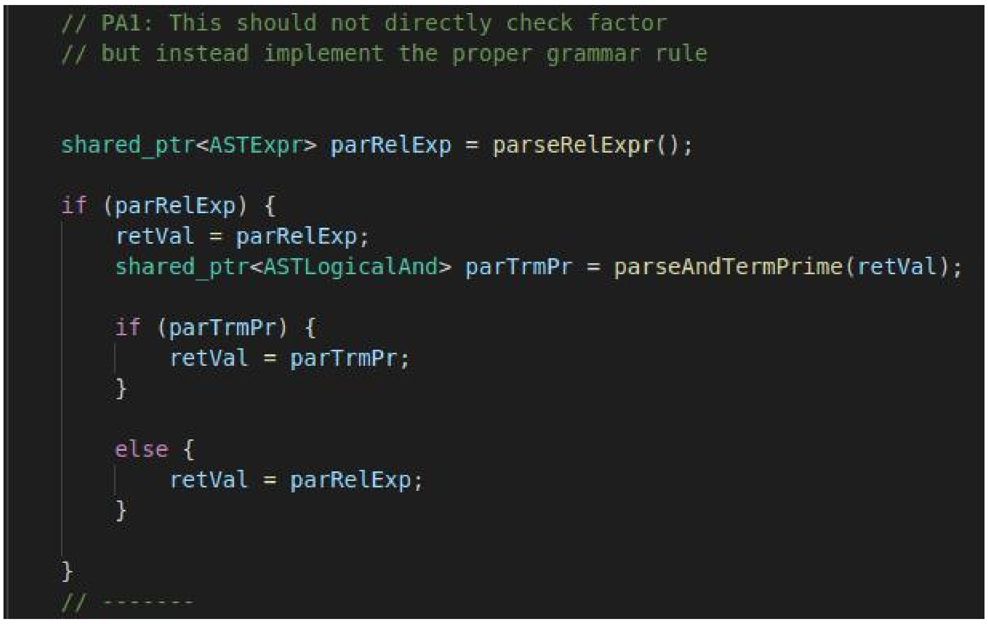
```
// PA1: Implement

if (peekToken() == Token::SemiColon) {

    consumeToken();

    retVal = make_shared<ASTNullStmt>();

}

// -------
```

Implement these two types of statements, and hook them up to parseStmt. These two classes can pass test011, 015 and 016. And the three test are also influenced by **parseCompoundStmt** class.

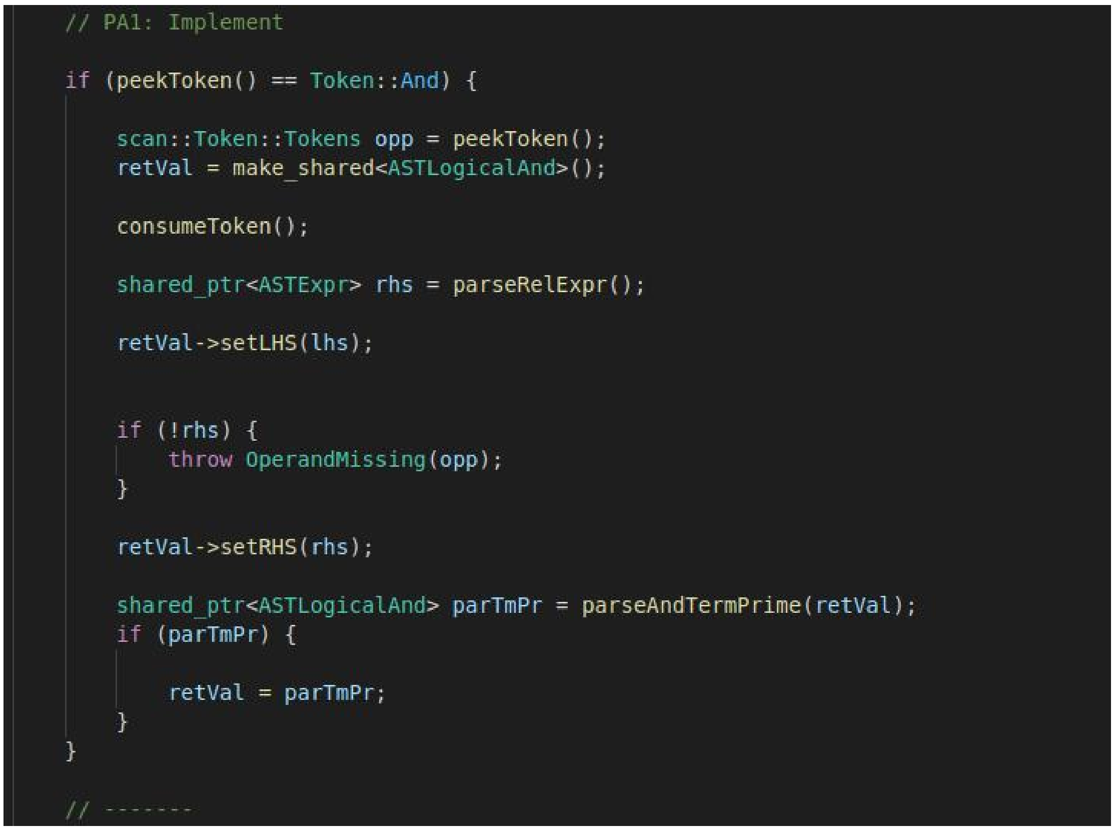
ParseExpr.cpp

i. parseAndTerm & parseAndTermPrime

Get the term and check it to parseAndTerm or parseExpr next. The parseExpr will go to parseTerm and parseExprPrime will go to parseTermPrime. All other binary operators should apply, which can pass the test009.



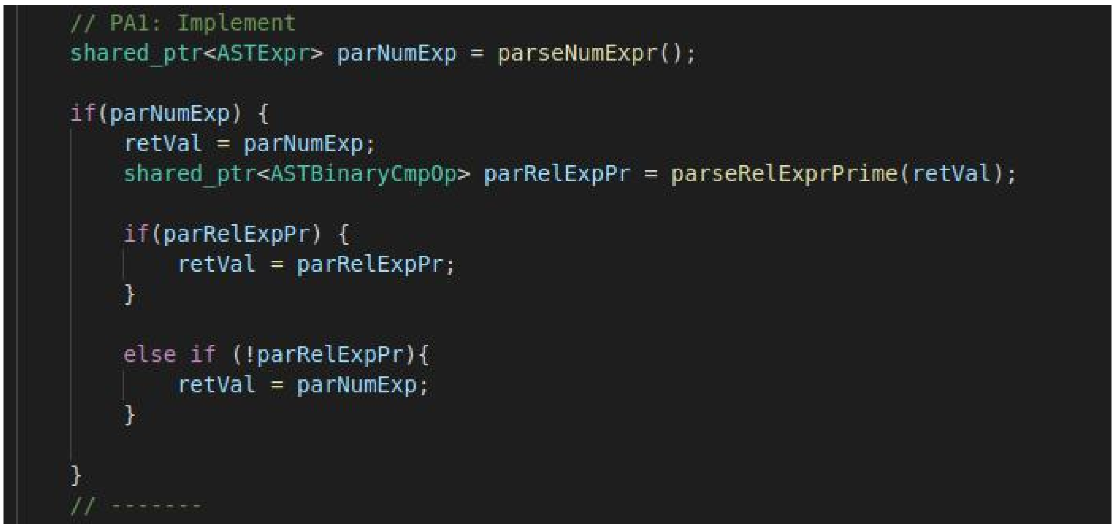
```
    // PA1: This should not directly check factor
    // but instead implement the proper grammar rule


    shared_ptr<ASTExpr> parRelExp = parseRelExpr();

    if (parRelExp) {
        retVal = parRelExp;
        shared_ptr<ASTLogicalAnd> parTrmPr = parseAndTermPrime(retVal);

        if (parTrmPr) {
            retVal = parTrmPr;
        }

        else {
            retVal = parRelExp;
        }

    }
    // -------
```



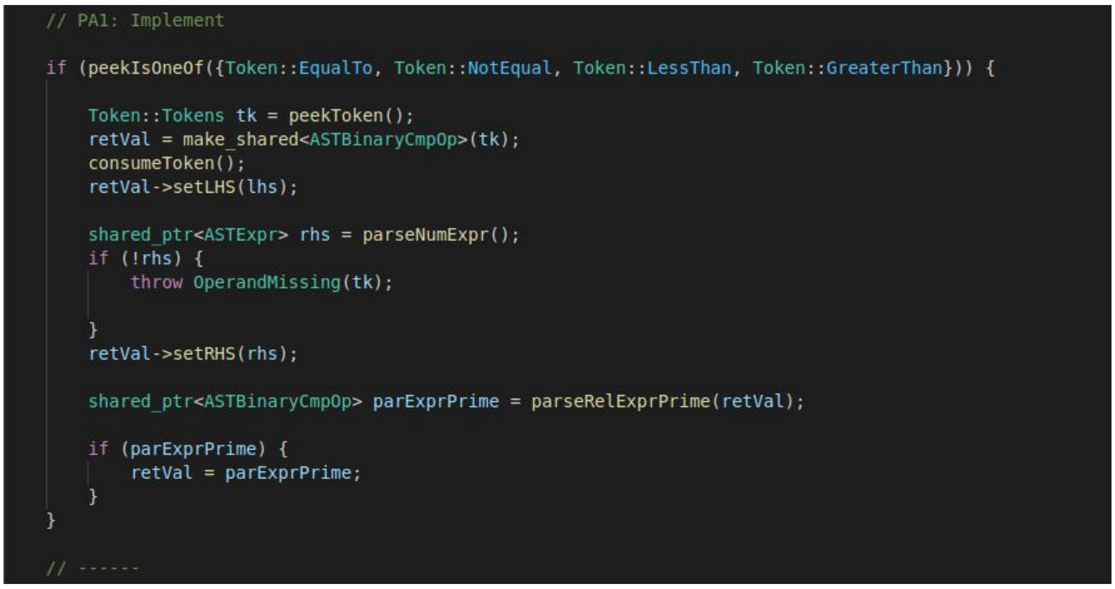
```
    // PA1: Implement

    if (peekToken() == Token::And) {

        scan::Token::Tokens opp = peekToken();
        retVal = make_shared<ASTLogicalAnd>();

        consumeToken();

        shared_ptr<ASTExpr> rhs = parseRelExpr();

        retVal->setLHS(lhs);


        if (!rhs) {
            throw OperandMissing(opp);
        }

        retVal->setRHS(rhs);

        shared_ptr<ASTLogicalAnd> parTmPr = parseAndTermPrime(retVal);
        if (parTmPr) {

            retVal = parTmPr;
        }
    }

    // -------
```

ii. parseRelExpr & parseRelExprPrime

They are similar to parseNumExpr and parseExprPrime, except they are relational operators, which can pass the test008.



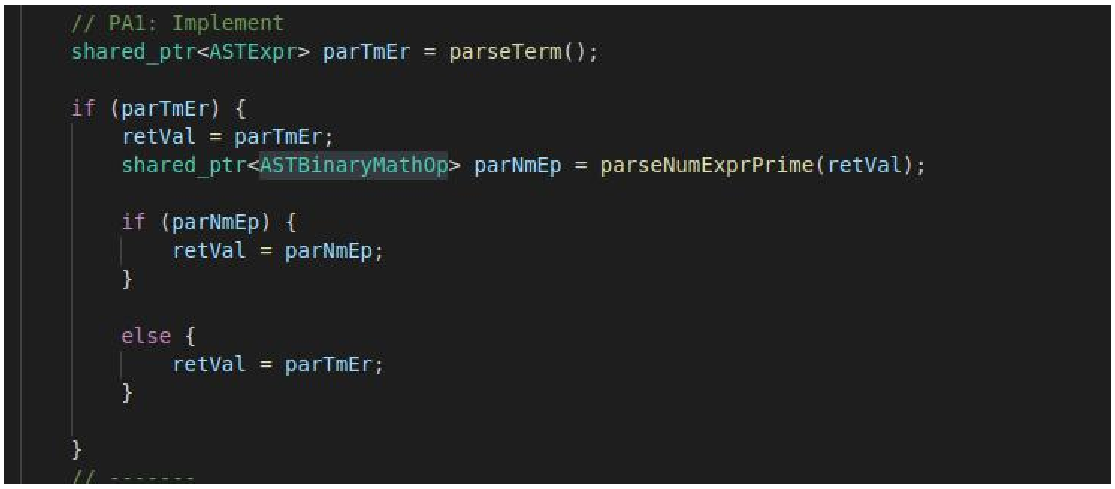
```
// PA1: Implement
shared_ptr<ASTExpr> parNumExp = parseNumExpr();

if(parNumExp) {
    retVal = parNumExp;
    shared_ptr<ASTBinaryCmpOp> parRelExpPr = parseRelExprPrime(retVal);

    if(parRelExpPr) {
        retVal = parRelExpPr;
    }

    else if (!parRelExpPr){
        retVal = parNumExp;
    }

}
// -------
```



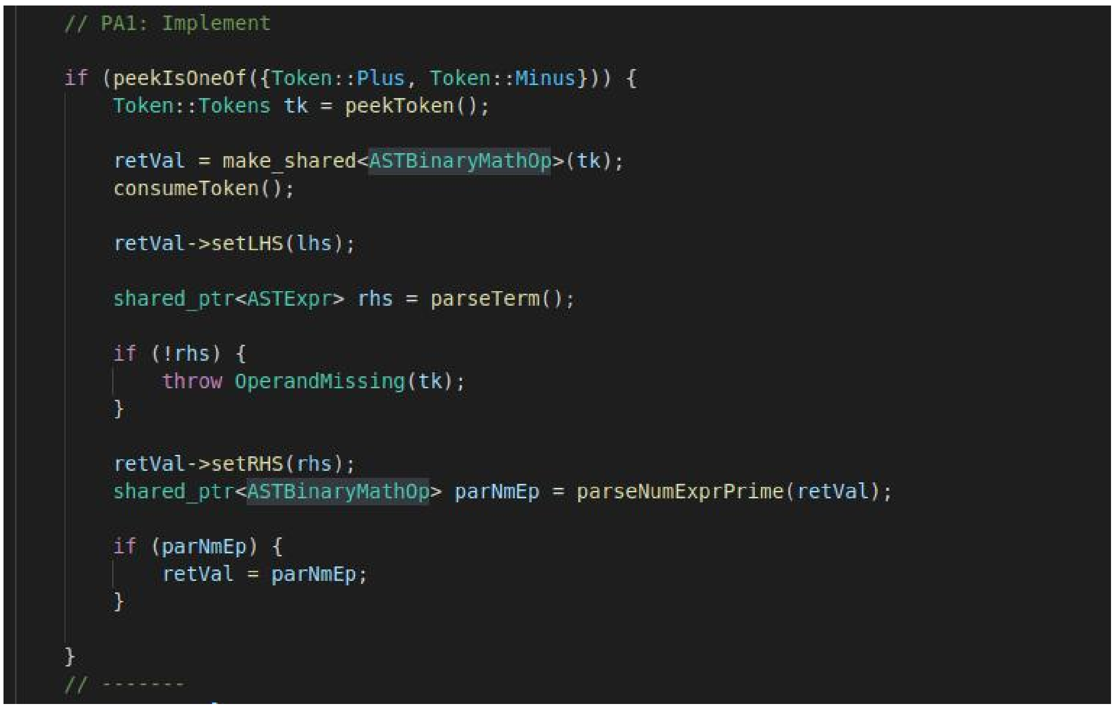
```
// PA1: Implement

if (peekIsOneOf({Token::EqualTo, Token::NotEqual, Token::LessThan, Token::GreaterThan})) {

    Token::Tokens tk = peekToken();
    retVal = make_shared<ASTBinaryCmpOp>(tk);
    consumeToken();
    retVal->setLHS(lhs);

    shared_ptr<ASTExpr> rhs = parseNumExpr();
    if (!rhs) {
        throw OperandMissing(tk);

    }
    retVal->setRHS(rhs);

    shared_ptr<ASTBinaryCmpOp> parExprPrime = parseRelExprPrime(retVal);

    if (parExprPrime) {
        retVal = parExprPrime;
    }
}

// ------
```

iii. parseNumExpr & parseNumExprPrime

These binary operators can pass the test007 and test013.

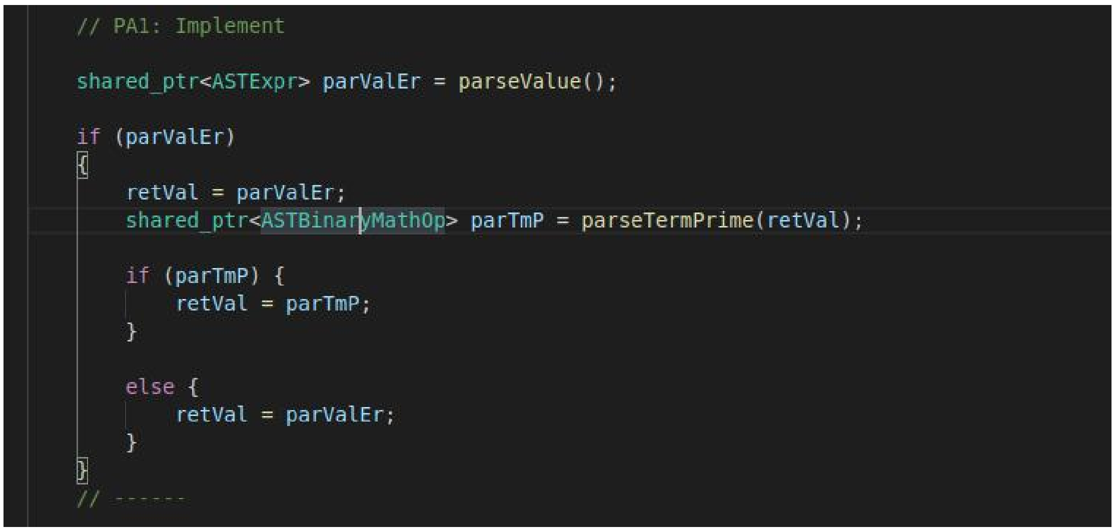


```
// PA1: Implement
shared_ptr<ASTExpr> parTmEr = parseTerm();

if (parTmEr) {
    retVal = parTmEr;
    shared_ptr<ASTBinaryMathOp> parNmEp = parseNumExprPrime(retVal);

    if (parNmEp) {
        retVal = parNmEp;
    }

    else {
        retVal = parTmEr;
    }

}
// -------
```

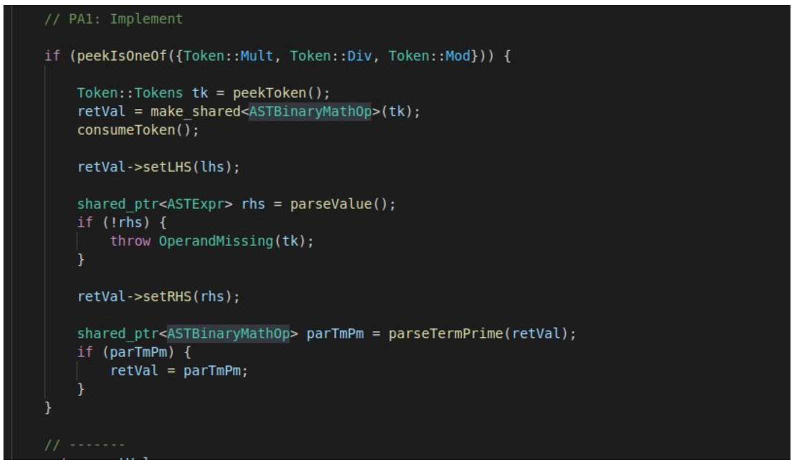


```cpp
    // PA1: Implement

    if (peekIsOneOf({Token::Plus, Token::Minus})) {
        Token::Tokens tk = peekToken();

        retVal = make_shared<ASTBinaryMathOp>(tk);
        consumeToken();

        retVal->setLHS(lhs);

        shared_ptr<ASTExpr> rhs = parseTerm();

        if (!rhs) {
            throw OperandMissing(tk);
        }

        retVal->setRHS(rhs);
        shared_ptr<ASTBinaryMathOp> parNmEp = parseNumExprPrime(retVal);

        if (parNmEp) {
            retVal = parNmEp;
        }

    }
    // -------
```

iv. parseTerm & parseTermPrime



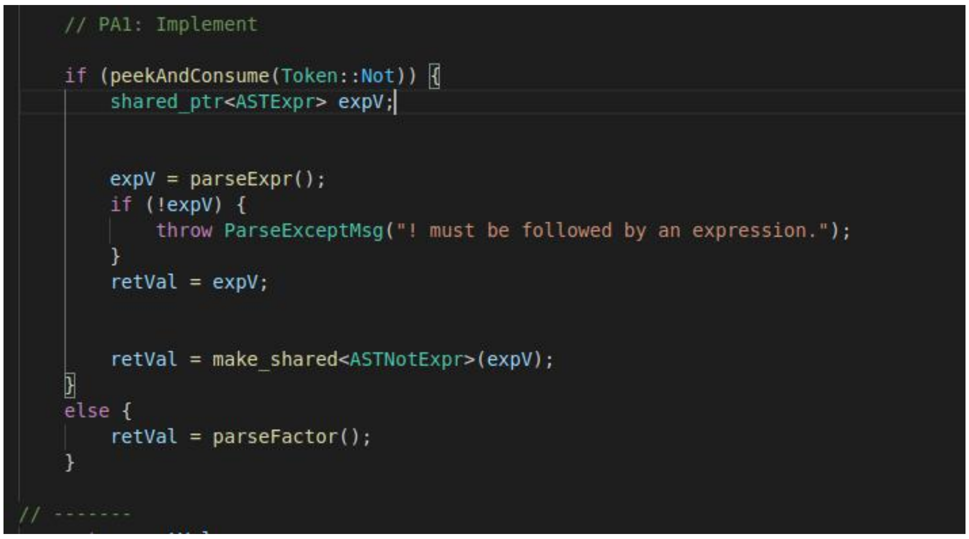
```cpp
    // PA1: Implement

    shared_ptr<ASTExpr> parValEr = parseValue();

    if (parValEr)
    {
        retVal = parValEr;
        shared_ptr<ASTBinaryMathOp> parTmP = parseTermPrime(retVal);

        if (parTmP) {
            retVal = parTmP;
        }

        else {
            retVal = parValEr;
        }
    }
    // ------
```



```cpp
    // PA1: Implement

    if (peekIsOneOf({Token::Mult, Token::Div, Token::Mod})) {

        Token::Tokens tk = peekToken();
        retVal = make_shared<ASTBinaryMathOp>(tk);
        consumeToken();

        retVal->setLHS(lhs);

        shared_ptr<ASTExpr> rhs = parseValue();
        if (!rhs) {
            throw OperandMissing(tk);
        }

        retVal->setRHS(rhs);

        shared_ptr<ASTBinaryMathOp> parTmPm = parseTermPrime(retVal);
        if (parTmPm) {
            retVal = parTmPm;
        }
    }

    // -------
```

v. parseValue

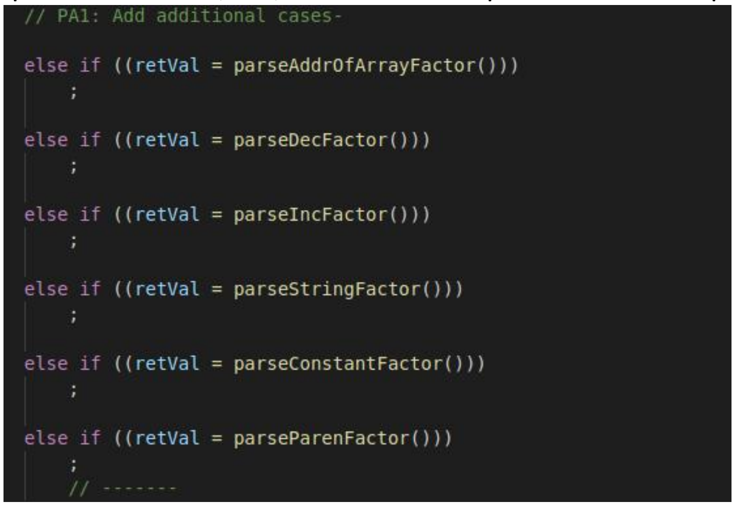
The part is used to parse token ‘not’. If that token is followed by the fault expression, it will throw the ParseExceptMsg warning.



```
    // PA1: Implement

    if (peekAndConsume(Token::Not)) {
        shared_ptr<ASTExpr> expV;


        expV = parseExpr();
        if (!expV) {
            throw ParseExceptMsg("! must be followed by an expression.");
        }
        retVal = expV;


        retVal = make_shared<ASTNotExpr>(expV);
    }
    else {
        retVal = parseFactor();
    }

// -------
```

vi. parseFactor

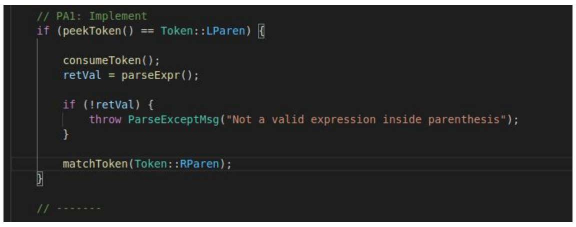
Some additional classes are added here, like 'parseAddrOfArrayFactor', 'parseDecFactor', etc, to meet the requirement of compiler running.



```
// PA1: Add additional cases-

else if ((retVal = parseAddrOfArrayFactor()))
    ;

else if ((retVal = parseDecFactor()))
    ;

else if ((retVal = parseIncFactor()))
    ;

else if ((retVal = parseStringFactor()))
    ;

else if ((retVal = parseConstantFactor()))
    ;

else if ((retVal = parseParenFactor()))
    ;
    // -------
```

vii. parseParenFactor

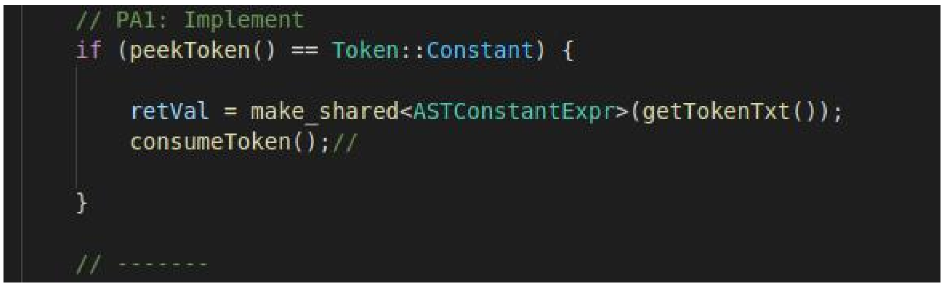
This class is used to parse left parent '(' and right parent ')'. When the left parent cannot be parsed, the error will be thrown and match the right parent.



```
// PA1: Implement
if (peekToken() == Token::LParen) {

    consumeToken();
    retVal = parseExpr();

    if (!retVal) {
        throw ParseExceptMsg("Not a valid expression inside parenthesis");
    }

    matchToken(Token::RParen);
}

// -------
```

viii. parseConstantFactor

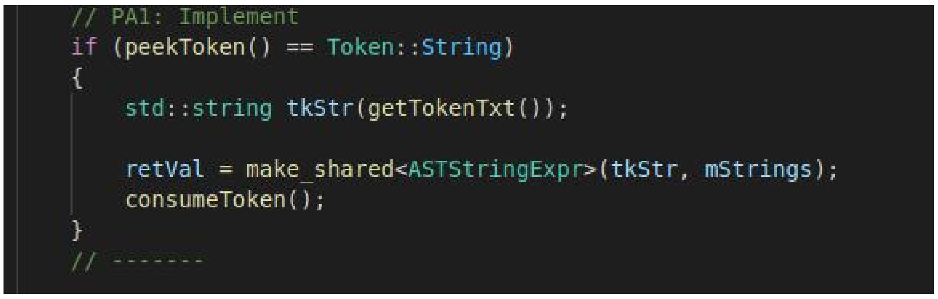
This class is used to parse the constant factor. When the input token is a constant, the text of the token can be returned.



```
// PA1: Implement
if (peekToken() == Token::Constant) {

    retVal = make_shared<ASTConstantExpr>(getTokenTxt());
    consumeToken();//

}

// -------
```

ix. parseStringFactor

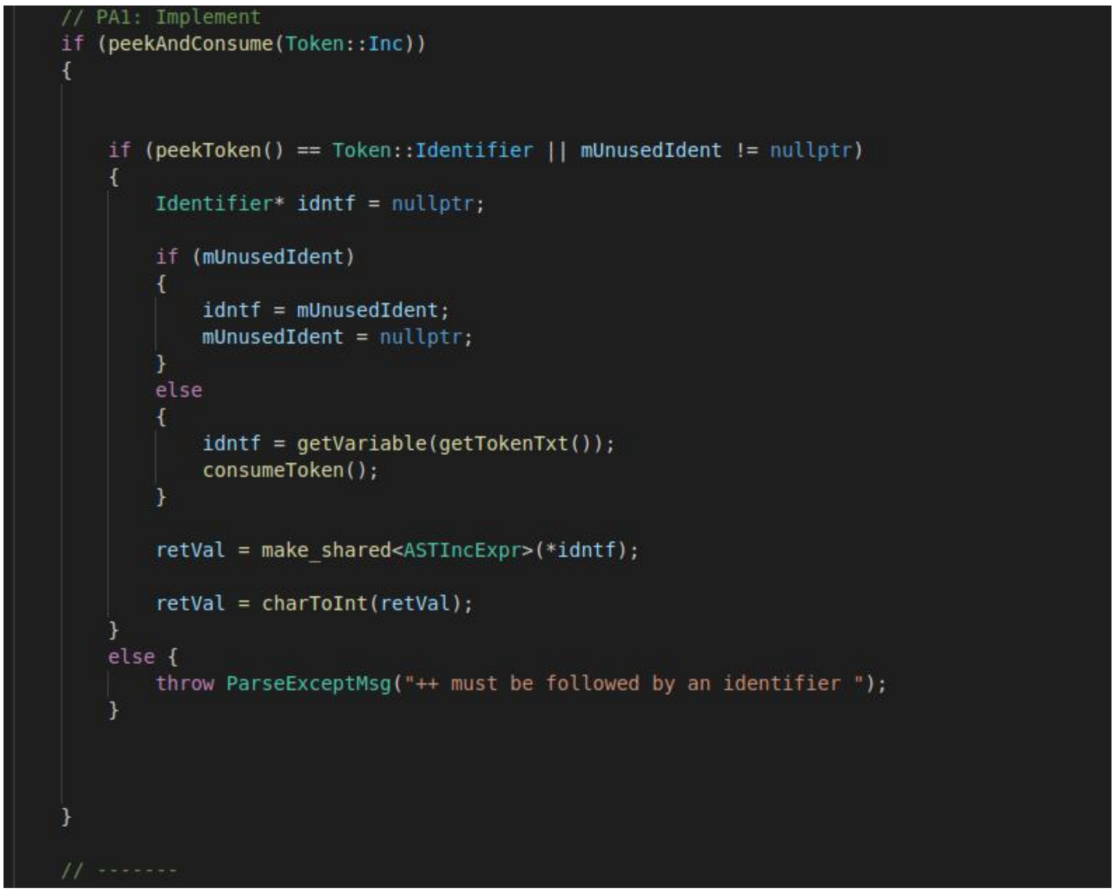
This class is used to parsed the String factor. When the input token is a String type, this class can get the text of the token.



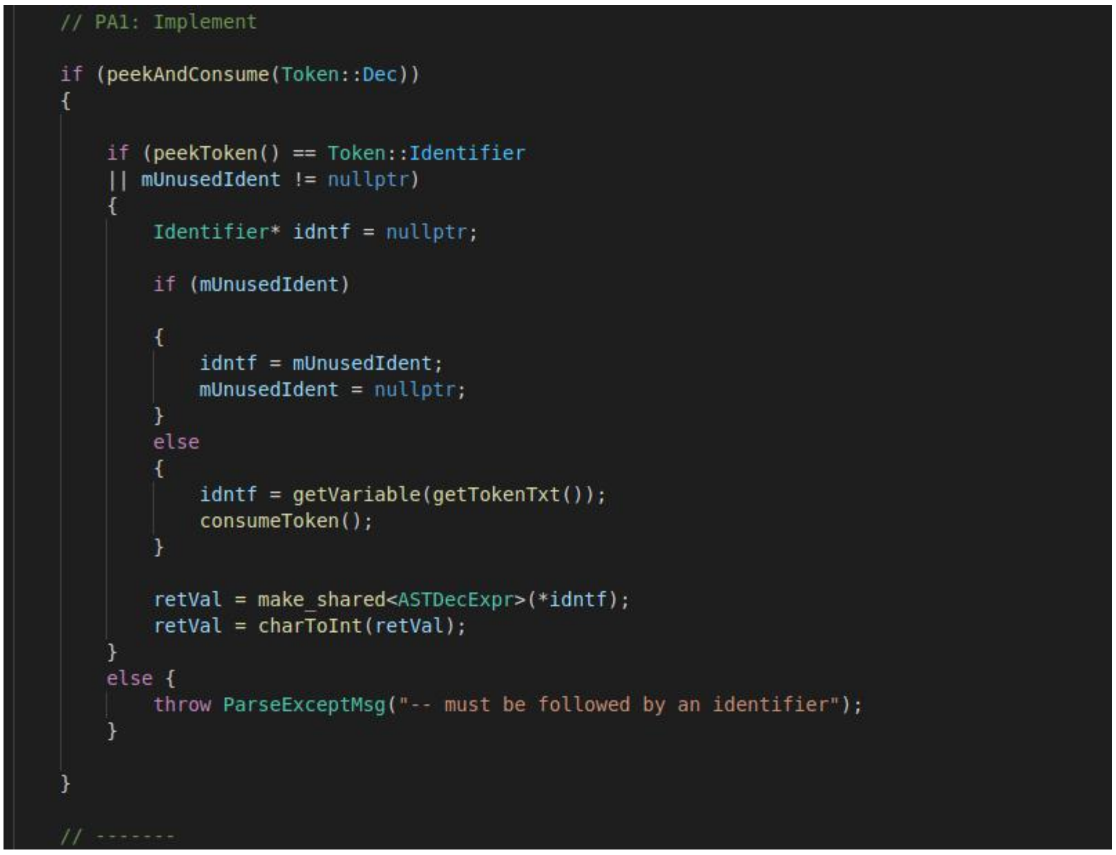
```
// PA1: Implement
if (peekToken() == Token::String)
{
    std::string tkStr(getTokenTxt());

    retVal = make_shared<ASTStringExpr>(tkStr, mStrings);
    consumeToken();
}
// -------
```

x. parseIncFactor & parseDecFactor

xi. The class ‘parseIncFactor’ is used to parse increasing factor ‘+’ and the class ‘parseDecFactor’ is used to parse decreasing factor ‘-’.



```
// PA1: Implement
if (peekAndConsume(Token::Inc))
{


    if (peekToken() == Token::Identifier || mUnusedIdent != nullptr)
    {
        Identifier* idntf = nullptr;

        if (mUnusedIdent)
        {
            idntf = mUnusedIdent;
            mUnusedIdent = nullptr;
        }
        else
        {
            idntf = getVariable(getTokenTxt());
            consumeToken();
        }

        retVal = make_shared<ASTIncExpr>(*idntf);

        retVal = charToInt(retVal);
    }
    else {
        throw ParseExceptMsg("++ must be followed by an identifier ");
    }



}

// -------
```



```
// PA1: Implement

if (peekAndConsume(Token::Dec))
{

    if (peekToken() == Token::Identifier
    || mUnusedIdent != nullptr)
    {
        Identifier* idntf = nullptr;

        if (mUnusedIdent)

        {
            idntf = mUnusedIdent;
            mUnusedIdent = nullptr;
        }
        else
        {
            idntf = getVariable(getTokenTxt());
            consumeToken();
        }

        retVal = make_shared<ASTDecExpr>(*idntf);
        retVal = charToInt(retVal);
    }
    else {
        throw ParseExceptMsg("-- must be followed by an identifier");
    }

}

// -------
```