The following are my code snippets and comments for each parse-function I implemented.

- **parseCompoundStmt**
  The function parses {} and find any declarations and statements in it.

```
// PA1: Implement
// isFuncBody not used
if (peekAndConsume(Token::LBrace)){

    retVal = make_shared<ASTCompoundStmt>();

    // call parseDecl if encounter int or char
    while (peekIsOneOf({Token::Key_int, Token::Key_char}))
        retVal.get()->addDecl(parseDecl());

    // call parseStmt untill encounter } or EOF
    while (!peekIsOneOf({Token::RBrace, Token::EndOfFile}))
        retVal.get()->addStmt(parseStmt());

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

- **parseReturnStmt**
  The function parses return statement.

```
// PA1: Implement
if (peekAndConsume(Token::Key_return)){
    if (peekAndConsume(Token::SemiColon))
        // return ;
        retVal = make_shared<ASTReturnStmt>(nullptr);
    else
    {
        // return Expr;
        shared_ptr<ASTExpr> expr = parseExpr();
        retVal = make_shared<ASTReturnStmt>(expr);
        matchToken(Token::SemiColon);
    }
}
```

- **parseConstantFactor** and **parseStringFactor**
  Function **parseConstantFactor** parses constant factor while function **parseStringFactor** parses string factor. If the current token is a constant, **parseConstantFactor** gets the text of the token directly and return it. If the current token is a string, **parseStringFactor** stores the string into string table first and then return it with the string table.

```
// PA1: Implement
if (peekToken() == Token::Constant)
{
    retVal = make_shared<ASTConstantExpr>(getTokenTxt());
    consumeToken();
}
```

```
if (peekToken() == Token::String)
{
    std::string tempStr(getTokenTxt());
    ConstStr* string = mStrings.getString(tempStr);
    retVal = make_shared<ASTStringExpr>(string->getText(), mStrings);
    consumeToken();
}
```

- **parseParenFactor**
  The function parse () and the expression in it. When errors occur, find next ")" and match it.

```
// PA1: Implement
if (peekAndConsume(Token::LParen))
{
    try
    {
        shared_ptr<ASTExpr> expr = parseExpr();
        if (!expr)
            throw ParseExceptMsg("Not a valid expression inside parenthesis");

        retVal = expr;
    }
    catch (ParseExcept& e)
    {
        reportError(e);
        consumeUntil(Token::RParen);
        if (peekToken() == Token::EndOfFile)
            throw EOFExcept();
    }
    matchToken(Token::RParen);
}
```

- **parseIncFactor** and **parseDecFactor**
  Function **parseIncFactor** parses increased factor while function **parseDecFactor** parses decreased factor. If the current token is "++" and the next token is a pure identifier, **parseIncFactor** parses them. If the current token is "--" and the next token is a pure identifier, **parseStringFactor** parses them.

```
// modified from parseIdentFactor
if (peekAndConsume(Token::Inc))
{
    try
    {
        // only accept pure identifier
        if (peekToken() == Token::Identifier || mUnusedIdent != nullptr)
        {
            Identifier* ident = nullptr;
            if (mUnusedIdent)
            {
                ident = mUnusedIdent;
                mUnusedIdent = nullptr;
            }
            else
            {
                ident = getVariable(getTokenTxt());
                consumeToken();
            }
            retVal = make_shared<ASTIncExpr>(*ident);
        }
        else
            throw ParseExceptMsg("++ must be followed by an identifier");
    }
    catch (ParseExcept& e)
    {
        reportError(e);
        // find next ;
        consumeUntil(Token::SemiColon);
        if (peekToken() == Token::EndOfFile)
            throw EOFExcept();
    }
}
```

```
if (peekAndConsume(Token::Dec))
{
    try
    {
        if (peekToken() == Token::Identifier || mUnusedIdent != nullptr)
        {
            Identifier* ident = nullptr;
            if (mUnusedIdent)
            {
                ident = mUnusedIdent;
                mUnusedIdent = nullptr;
            }
            else
            {
                ident = getVariable(getTokenTxt());
                consumeToken();
            }
            retVal = make_shared<ASTDecExpr>(*ident);
        }
        else
            throw ParseExceptMsg("-- must be followed by an identifier");
    }
    catch (ParseExcept& e)
    {
        reportError(e);
        consumeUntil(Token::SemiColon);
        if (peekToken() == Token::EndOfFile)
            throw EOFExcept();
    }
}
```

- **parseValue**
  The function parses “!” token. If that token is followed by a bad expression, it returns a **BadExpr** instead.

```
// PA1: Implement
if (peekAndConsume(Token::Not))
{
    // check Expr
    shared_ptr<ASTExpr> expr;
    try
    {
        expr = parseExpr();
        if (!expr)
            throw ParseExceptMsg("! must be followed by an expression.");
        retVal = expr;
    }
    catch (ParseExcept& e)
    {
        reportError(e);
        // use BadExpr for further parsing
        expr = make_shared<ASTBadExpr>();
    }
    retVal = make_shared<ASTNotExpr>(expr);
}
else
    retVal = parseFactor();
```

- **parseTerm** and **parseTermPrime**

They are very similar to **parseExpr** and **parseExprPrime**. The differences are **parseExpr** -> **parseTerm**, **parseExprPrime** -> **parseTermPrime,** and some math operators are included for **peekIsOneOf** to check. In fact, the code is modified from **parseExpr** and **parseExprPrime**.

```
// PA1: Implement
shared_ptr<ASTExpr> value = parseValue();

if (value)
{
    retVal = value;
    shared_ptr<ASTBinaryMathOp> termPrime = parseTermPrime(retVal);

    if (termPrime)
        retVal = termPrime;
}
```

```
// PA1: Implement
if (peekIsOneOf({Token::Mult, Token::Div, Token::Mod}))
{
    Token::Tokens op = peekToken();
    retVal = make_shared<ASTBinaryMathOp>(op);
    consumeToken();
    retVal->setLHS(lhs);

    shared_ptr<ASTExpr> rhs = parseValue();
    if (!rhs)
        throw OperandMissing(op);
    retVal->setRHS(rhs);

    // PA2: Finalize op

    shared_ptr<ASTBinaryMathOp> termPrime = parseTermPrime(retVal);
    if (termPrime)
        retVal = termPrime;
}
```

- **parseNumExpr/Prime**

  They are very similar to **parseExpr** and **parseExprPrime**, just like above.

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

if (term)
{
    retVal = term;
    shared_ptr<ASTBinaryMathOp> numExprPrime = parseNumExprPrime(retVal);

    if (numExprPrime)
        retVal = numExprPrime;
}
```

```
// PA1: Implement
if (peekIsOneOf({Token::Plus, Token::Minus}))
{
    Token::Tokens op = peekToken();
    retVal = make_shared<ASTBinaryMathOp>(op);
    consumeToken();
    retVal->setLHS(lhs);

    shared_ptr<ASTExpr> rhs = parseTerm();
    if (!rhs)
        throw OperandMissing(op);
    retVal->setRHS(rhs);

    // PA2: Finalize op

    shared_ptr<ASTBinaryMathOp> numExprPrime = parseNumExprPrime(retVal);
    if (numExprPrime)
        retVal = numExprPrime;
}
```

- **parseRelExpr/Prime**

  They are very similar to **parseExpr** and **parseExprPrime**, just like above.

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

if (numExpr)
{
    retVal = numExpr;
    shared_ptr<ASTBinaryCmpOp> relExprPrime = parseRelExprPrime(retVal);

    if (relExprPrime)
        retVal = relExprPrime;
}
```

```
// PA1: Implement
if (peekIsOneOf({Token::EqualTo, Token::NotEqual, Token::LessThan, Token::GreaterThan}))
{
    Token::Tokens op = peekToken();
    retVal = make_shared<ASTBinaryCmpOp>(op);
    consumeToken();
    retVal->setLHS(lhs);

    shared_ptr<ASTExpr> rhs = parseNumExpr();
    if (!rhs)
        throw OperandMissing(op);
    retVal->setRHS(rhs);

    // PA2: Finalize op

    shared_ptr<ASTBinaryCmpOp> relExprPrime = parseRelExprPrime(retVal);
    if (relExprPrime)
        retVal = relExprPrime;
}
```

- **parseAndTerm/Prime**

  They are very similar to **parseExpr** and **parseExprPrime**, just like above.

```
shared_ptr<ASTExpr> relExpr = parseRelExpr();

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

    if (andTermPrime)
        retVal = andTermPrime;
}
```

```
// PA1: Implement
if (peekToken() == Token::And)
{
    Token::Tokens op = peekToken();
    retVal = make_shared<ASTLogicalAnd>();
    consumeToken();
    retVal->setLHS(lhs);

    shared_ptr<ASTExpr> rhs = parseRelExpr();
    if (!rhs)
        throw OperandMissing(op);
    retVal->setRHS(rhs);

    // PA2: Finalize op

    shared_ptr<ASTLogicalAnd> andTermPrime = parseAndTermPrime(retVal);
    if (andTermPrime)
        retVal = andTermPrime;
}
```

- **parseWhileStmt**
  The function parses while statement. It first peeks and consumes “while” token and match “(” . Then it checks if the expression “(” is valid. If it is valid, then it matches “)”. Otherwise, it finds next “)” or “;” until EOF.

```
// PA1: Implement
shared_ptr<ASTExpr> expr;
shared_ptr<ASTStmt> stmt;

if (peekAndConsume(Token::Key_while)){

    // match ( and check Expr
    matchToken(Token::LParen);
    try
    {
        expr = parseExpr();
        if (!expr)
            throw ParseExceptMsg("Invalid condition for while statement");
        matchToken(Token::RParen);
    }
    catch (ParseExcept& e)
    {
        reportError(e);
        // find next ) or ;
        consumeUntil({Token::RParen, Token::SemiColon});
        if (peekToken() == Token::EndOfFile)
            throw EOFExcept();
        consumeToken();
    }
    stmt = parseStmt();
    retVal = make_shared<ASTWhileStmt>(expr, stmt);
}
```

- **parseExprStmt** and **parseNullStmt**
  Function **parseExprStmt** parses regular expression statements while function **parseNullStmt** parses null statements.

```
// PA1: Implement
// if the current token is not ;, int, or char
// if there is a unused identifier or array
if (!peekIsOneOf({Token::SemiColon, Token::Key_int, Token::Key_char}) ||
    mUnusedIdent != nullptr || mUnusedArray != nullptr)
{
    // parse the Expr and match ;
    shared_ptr<ASTExpr> expr = parseExpr();
    retVal = make_shared<ASTExprStmt>(expr);
    matchToken(Token::SemiColon);
}
```

```
// PA1: Implement
if (peekAndConsume(Token::SemiColon))
    retVal = make_shared<ASTNullStmt>();
```

- **parseIfStmt**
  Function **parseIfStmt** parse if statements. The implementation is like function **parseWhileStmt**. The only difference is that we need to check if an if statement is followed by an “else” token.

```
// PA1: Implement
shared_ptr<ASTExpr> expr;
shared_ptr<ASTStmt> stmtTrue, stmtFalse;

if (peekAndConsume(Token::Key_if)){

    // match ( and check Expr
    matchToken(Token::LParen);
    try
    {
        expr = parseExpr();
        if (!expr)
            throw ParseExceptMsg("Invalid condition for if statement");
        matchToken(Token::RParen);
    }
    catch (ParseExcept& e)
    {
        reportError(e);
        consumeUntil({Token::RParen, Token::SemiColon});
        if (peekToken() == Token::EndOfFile)
            throw EOFExcept();
        consumeToken();
    }
    stmtTrue = parseStmt();
    // check if encounter else token (optional)
    if (peekAndConsume(Token::Key_else))
        stmtFalse = parseStmt();
    retVal = make_shared<ASTIfStmt>(expr, stmtTrue, stmtFalse);
}
```

- **parseAddrOfArrayFactor**
  I briefly explain my idea here since the code is much longer. The implementation is modified from **parseIdentFactor**. It peeks and consumes “&” first, and then it checks if the current token is an identifier or there is an unused array. The rest is like **parseIdentFactor**, but it only focuses on array identifiers this time. Thus, it matches “[“ instead of using if statement.