

# LL(1) Parser Report

과 목	프로그래밍 언어론 Principles of Programming Languages Autumn, 2023
담당 교수	김진성
학부	소프트웨어학부
팀	20202203 박호근 20204946 이규성

## Table of Contents

-Internal Documents .....	3
1. utils.h.....	3
2. utils.cpp.....	3
3. treenode.h.....	4
4. treenode.cpp.....	5
5. parser.h.....	5
6. parser.cpp.....	6
7. main.cpp.....	7
8. Error check.....	8
-External Documents (Same with README.md).....	11
1. Build.....	11
2. Build and Run.....	11
3. Test environment .....	11
4. OS.....	11
5. Compiler .....	11
Comprehensive explanation .....	12
outline.....	12
Makefile.....	13
Program Structure .....	14
Example.....	15
입력 #1.....	15
입력 #2.....	15
입력 #3.....	16
입력 #4.....	16
Conclusion .....	17

## -Internal Documents

### 1. utils.h

```
#ifndef UTILS_H
#define UTILS_H
#include <iostream>
#include <map>
#include <cmath>
#include <string>

class Token { ... };

class Lexer { ... };

class SymbolTable { ... };

#endif
```

Class Token, Lexer, SymbolTable, 내부 method와 attribute 선언.

### 2. utils.cpp

```
#include "utils.h"

Token::Token(Type type, std::string value) : type(type), value(value) {}

Lexer::Lexer(const std::string& input) : input(input), position(0) {}

Token Lexer::getNextToken() { ... }

void SymbolTable::set(const std::string& name, double value) { ... }

bool SymbolTable::get(const std::string& name, double& value) const { ... }

bool SymbolTable::exists(const std::string& name) const { ... }

void SymbolTable::print_result() const { ... }
```

Utils.h에서 선언한 class들의 선언자와 내부 method body 작성.

Lexer::getNextToken : 현재 position의 character값을 받아 const, ident, op로 분류하고 분류한 token과 lexeme을 반환.

SymbolTable::set : symbol의 name과 value을 받아 symbols array에 저장.

SymbolTable::get : symbol의 name과 value의 주소를 받아 symbols array에 존재하는지 확인 후 존재한다면 value의 주소에 symbols array에 저장된 value값을 저

장하고 true를 반환. 존재하지 않는다면 false를 반환.

SymbolTable::exists : symbol의 name을 받아 symbols array에 존재한다면 true를 반환.

SymbolTable::print\_result : parse 결과값을 출력.

### 3. treenode.h

```
// TreeNode.h

#ifndef TREENODE_H
#define TREENODE_H

#include "utils.h"

class TreeNode { ... };
class FactorNode { ... };
class FactorTailNode { ... };
class TermNode { ... };
class TermTailNode { ... };
class ExpressionNode { ... };
class StatementNode { ... };
class StatementsNode { ... };
class ProgramNode { ... };

#endif //TREENODE_H
```

LL(1)Parser의 문법에 존재하는 nonterminal을 class로 선언, node형식으로 정의하여 parse tree의 생성에 사용.

## 4. treenode.cpp

```
#include "treenode.h"
#include "parser.h"

//treenode
TreeNode::TreeNode(bool isParsed, SymbolTable& symbolTable) : isParsed(isParsed), symbolTable(symbolTable){};
void TreeNode::setIsParsed(){isParsed = true;}

//program
ProgramNode::ProgramNode(StatementsNode* statementsNode) : statementsNode(statementsNode){};

void ProgramNode::calculate_statements() { ... }

//statements
StatementsNode::StatementsNode(StatementNode* statementNode, bool semi_colon, StatementsNode* statementsNode) : statementNode(statementNode),
statementsNode(statementsNode){};

void StatementsNode::calculate_statement() { ... }

//statement
StatementNode::StatementNode(bool isParsed, SymbolTable& symbolTable, std::string ident, bool assignment_op, ExpressionNode* expressionNode) :
isParsed(isParsed), symbolTable(symbolTable), ident(ident), assignment_op(assignment_op), expressionNode(expressionNode){};

double StatementNode::calculate() { ... }

//expression
ExpressionNode::ExpressionNode(bool isParsed, SymbolTable& symbolTable, TermNode* termNode, TermTailNode* termTailNode) { ... }
double ExpressionNode::calculate() { ... }

//term
TermNode::TermNode(bool isParsed, SymbolTable& symbolTable, FactorNode* factorNode, FactorTailNode* factorTailNode) { ... }
double TermNode::calculate() { ... }

//termtail
TermTailNode::TermTailNode(bool isParsed, SymbolTable& symbolTable) : TreeNode(isParsed, symbolTable){};
TermTailNode::TermTailNode(bool isParsed, SymbolTable& symbolTable, int add_op, TermNode* termNode, TermTailNode* termTailNode) { ... }
double TermTailNode::calculate() { ... }
int TermTailNode::get_op() {return add_op;}

//factor
FactorNode::FactorNode(bool isParsed, SymbolTable& symbolTable, bool left_paren, ExpressionNode* expressionNode, bool right_paren) { ... }
FactorNode::FactorNode(bool isParsed, SymbolTable& symbolTable, std::string ident) { ... }
FactorNode::FactorNode(bool isParsed, SymbolTable& symbolTable, double _const) { ... }
double FactorNode::calculate() { ... }

//factortail
FactorTailNode::FactorTailNode(bool isParsed, SymbolTable& symbolTable) : TreeNode(isParsed, symbolTable){};
FactorTailNode::FactorTailNode(bool isParsed, SymbolTable& symbolTable, int mult_op, FactorNode* factorNode, FactorTailNode* factorTailNode) { ... }
double FactorTailNode::calculate() { ... }
int FactorTailNode::get_op() {return mult_op;}
```

treenode.h에서 선언한 class들의 생성자들과 결과값을 구하기 위해 parse tree의 해당 노드에서 해야하는 연산을 calculate함수로 정의, get\_op 함수로 nonterminal <add\_op>와 <mult\_op>를 결정하여 calculate 함수에 반환.

## 5. parser.h

```
#ifndef PARSER_H
#define PARSER_H

#include "treenode.h"
#include <iostream>
#include <cmath>

class Parser { ... };

#endif
```

class Parser 선언.

## 6. parser.cpp

```
#include "parser.h"
#include "treenode.h"

ProgramNode* Parser::parseProgram() { ... }

StatementsNode* Parser::parseStatements() { ... }

StatementNode* Parser::parseStatement() { ... }

ExpressionNode* Parser::parseExpression() { ... }

TermNode* Parser::parseTerm() { ... }

TermTailNode* Parser::parseTermTail() { ... }

FactorNode* Parser::parseFactor() { ... }

FactorTailNode* Parser::parseFactorTail() { ... }
```

해당 노드의 RHS 호출과 Lexer 에서 반환된 token과 lexeme으로 LHS 생성하여 반환.

## 7. main.cpp

```
#include <fstream>
#include <string>
#include "parser.h"

int main(int argc, char **argv)
{
    std::string input;
    if (argc != 2) {
        std::cerr << "Usage: " << argv[0] << " <filename>" << std::endl;
        return 1;
    }

    std::ifstream file(argv[1]);
    if (!file.is_open()) {
        std::cerr << "Failed to open file: " << argv[1] << std::endl;
        return 1;
    }

    std::getline(file, input, '#');
    file.close();
    std::cout << input << std::endl;
    // std::string input a = "";

    SymbolTable symbolTable;
    Parser parser(input, symbolTable);
    ProgramNode* rootNode = parser.parseProgram();
    rootNode->calculate_statements();

    symbolTable.print_result();

    return (0);
}
```

main에서 file을 읽어 file 내부의 string을 읽어 input string변수에 저장해 parser class instance 생성. ProgramNode 포인터 rootNode를 생성해 parser.Program()으로 start symbol로 input data parsing. rootNode->calculate\_statements();로 parsing된 결과를 연산하여 symbolTable에 저장.

symbolTable.print\_result()로 input data 연산한 결과 출력.

## 8. Error check

### 1. 선언되지 않은 ident 사용

```
b := a + 10  
b := a + 10  
(Error)Cannot use undefined variable.  
ID: 2; CONST:1; OP: 1  
Result ==> b: nan;
```

### 2. 연산자 중복

```
b := 1 + * 10  
b := 1 + 10  
(Warning)Operator is duplicated.  
ID: 1; CONST:2; OP: 1  
Result ==> b: 11;
```

### 3. 괄호 쌍 체크

```
a := 1;  
b := a + 10;  
c := (a + b * 9  
a := 1;  
ID: 1; CONST:1; OP: 0  
b := a + 10;  
ID: 2; CONST:1; OP: 1  
c := a + b * 9  
(Warning)checking if the parentheses are properly opened and closed  
ID: 3; CONST:1; OP: 2  
Result ==> a: 1; b: 11; c: 100;
```

```
a := 1;  
b := a + 10;  
c := a + b ) * 9  
a := 1;  
ID: 1; CONST:1; OP: 0  
b := a + 10;  
ID: 2; CONST:1; OP: 1  
c := a + b * 9  
(Warning)checking if the parentheses are properly opened and closed  
ID: 3; CONST:1; OP: 2  
Result ==> a: 1; b: 11; c: 100;
```



```

a := 1;
b := a + 10;
c := ( ( a + b ) * 9

a := 1;
ID: 1; CONST:1; OP: 0
b := a + 10;
ID: 2; CONST:1; OP: 1
c := ( a + b ) * 9
(Warning)checking if the parentheses are properly opened and closed
ID: 3; CONST:1; OP: 2
Result ==> a: 1; b: 11; c: 108;

```

```

a := 1;
b := a + 10;
c := ( a + b ) ) * 9

a := 1;
ID: 1; CONST:1; OP: 0
b := a + 10;
ID: 2; CONST:1; OP: 1
c := ( a + b ) * 9
(Warning)checking if the parentheses are properly opened and closed
ID: 3; CONST:1; OP: 2
Result ==> a: 1; b: 11; c: 108;

```

#### 4. 두 op, ident 사이 연산자 생략

```

a := 10 20

a := 10 20
(Error)Invalid location of Token.
ID: 1; CONST:2; OP: 0
Result ==> a: nan;

```

#### 5. RHS 생략

```

a :=

a :=
(Error)Empty RHS
ID: 1; CONST:0; OP: 0
Result ==> a: nan;

```

## 6. File.txt의 마지막 줄에 ; 존재

```
a := 1;
b := a + 10;
c := ( a + b ) * 9;

a := 1;
ID: 1; CONST:1; OP: 0
b := a + 10;
ID: 2; CONST:1; OP: 1
c := ( a + b ) * 9
(Warning)Invalid Location of SEMICOLON.
ID: 3; CONST:1; OP: 2
Result ==> a: 1; b: 11; c: 108;
```

## 7. 기타 예외처리 (위의 중복된 경우들)

같이 첨부드린 exception\_example을 참고해주시길 바랍니다.

```
→ project01 git:(main) x ./parser exception_example.txt
a := );
b := ;
c = 4;
a 1 2;
c := (1 + 2;
d := );
e := 1 + );

a :=;
(Warning)checking if the parentheses are properly opened and closed
ID: 1; CONST:0; OP: 0
b := ;
(Error)Empty RHS
ID: 1; CONST:0; OP: 0
c := 4;
(Error)Expected token type: 2. Got: =
ID: 1; CONST:1; OP: 0
a := 2;
(Error)Expected token type: 2. Got: 1
ID: 1; CONST:1; OP: 0
c := 1 + 2;
(Warning)checking if the parentheses are properly opened and closed
ID: 1; CONST:2; OP: 1
d :=;
(Warning)checking if the parentheses are properly opened and closed
ID: 1; CONST:0; OP: 0
e := 1 +
(Warning)checking if the parentheses are properly opened and closed
(Warning)Invalid Location of SEMICOLON.
ID: 1; CONST:1; OP: 1
Result ==> a: 2; b: Unknown; c: Unknown; d: Unknown; e: Unknown; %
```

-External Documents (Same with README.md)

## 1. Build

```
$ make
```

## 2. Build and Run

```
$> make test
```

Just Run (after Build)

```
$> ./parser file.txt
```

file.txt is the input file used as a argument. So, tester will modify the file.txt for testing.

## 3. Test environment

- macOS 14.0 arm64

## 4. OS

- macOS Sonoma 14.0

## 5. Compiler

- Apple clang version 14.0.3 (clang-1403.0.22.14.1)

- Target: arm64-apple-darwin23.0.0

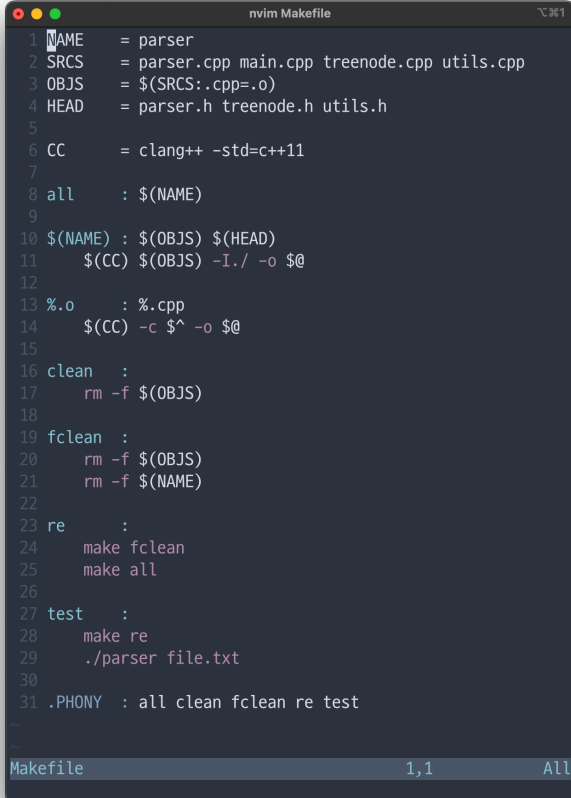
## Comprehensive explanation

### outline

해당 소스 코드를 빌드하기 위해서는, 앞서 밝힌 External Documents에서의 환경이 필요합니다. 물론, Standard library들을 사용하였기 때문에 다른 OS환경과 Compiler를 사용하더라도, 컴파일 및 실행이 가능할 것으로 생각됩니다. 다만, 정확한 테스트를 위해 위와 같은 환경에서 코드를 테스트해주시면 감사하겠습니다. 예외 처리와 관련한 예시는 exception\_example.txt를 확인하시고, 추가적인 테스트가 필요하시다면 해당 파일을 기반으로 수정해가시면서 처리되는 것을 테스트해보시면 좋을 것 같습니다.

저희 팀은, Makefile을 만들어 컴파일을 자동화하려고 하였습니다. 이규성 팀원은 Mac환경에서 해당 Makefile이 잘 동작하는지 확인하였고, 기본적인 lexer와 parser의 골격을 만들었습니다. 박호근 팀원은 Window10 환경에서, 해당 소스 코드를 visual studio IDE를 활용해 오류 처리부를 맡았습니다.

Makefile은, 총 5가지 명령어를 지원합니다.



```
1 NAME = parser
2 SRCS = parser.cpp main.cpp treenode.cpp utils.cpp
3 OBJS = $(SRCS:.cpp=.o)
4 HEAD = parser.h treenode.h utils.h
5
6 CC = clang++ -std=c++11
7
8 all : $(NAME)
9
10 $(NAME) : $(OBJS) $(HEAD)
11     $(CC) $(OBJS) -I./ -o $@
12
13 %.o : %.cpp
14     $(CC) -c $^ -o $@
15
16 clean :
17     rm -f $(OBJS)
18
19 fclean :
20     rm -f $(OBJS)
21     rm -f $(NAME)
22
23 re :
24     make fclean
25     make all
26
27 test :
28     make re
29     ./parser file.txt
30
31 .PHONY : all clean fclean re test
```

## Makefile

```
$> make
```

또는

```
$> make all
```

명령어는 소스코드를 통해 컴파일하여 실행파일을 만드는 일련의 절차를 수행합니다.

```
$> make clean
```

명령어는 효율적인 컴파일에 필요할 수 있는 \*.o file들, 즉 목적파일을 지우는 명령을 수행합니다.

```
$> make fclean
```

명령어는 \*.o 파일과 타겟 파일(실행가능한 바이너리파일)을 삭제합니다.

```
$> make re
```

명령어는 컴파일된 모든 파일을 지우고 다시 컴파일을 시행합니다.

```
$> make test
```

명령어는 컴파일 후 나온 실행파일을, file.txt파일을 인자로 주어 실행시킵니다.

## Program Structure

저희가 개발한 프로그램은 다음과 같은 과정을 거칩니다.

1. 인자로 들어온 파일을 읽습니다.
2. 파일 내부의 문자들을 읽어가며 lexer가 토큰으로 분리, parser가 적절한 함수들을 호출하며 parse tree를 만들어갑니다.
3. Statement 단위로 오류가 생겼다면, internal docs에서 기술한 방식대로 최대한 오류를 표기하고 복구하려고 합니다. 또한 statement 단위로 해당 statement에서 몇 개의 OP, CONST, IDENT가 나왔는지도 표기합니다.
4. Parse tree가 잘 그려졌다면, Parse tree의 최상단 노드에서 calculate 함수를 호출하여 각각의 statement들이 내놓는 결과를 하향식으로 계산하기 시작합니다.
5. Statement 노드는 그 하위로 가지고 있는 노드들의 calculate 함수를 호출해가며 결과적으로 assignment 되는 <IDENT>의 값을 심볼 테이블에 최신화 합니다.
6. 과정이 정상적으로 끝난 이후에는 symbol table을 순회하며, <IDENT>의 값들을 RESULT로 내놓습니다.

## Example

과제의 예시 실행 결과

### 입력 #1

```
→ project01 git:(main) ✗ ./parser file.txt
operand1 := 3 ;
operand2 := operand1 + 2 ;
target := operand1 + operand2 * 3

operand1 := 3;
(OK)
ID: 1; CONST:1; OP: 0
operand2 := operand1 + 2;
(OK)
ID: 2; CONST:1; OP: 1
target := operand1 + operand2 * 3
(OK)
ID: 3; CONST:1; OP: 2
Result ==> operand1: 3; operand2: 5; target: 18;
```

- 정상적으로 인지하는 것을 확인했습니다.

### 입력 #2

```
→ project01 git:(main) ✗ ./parser file.txt
operand2 := operand1 + 2 ;
target := operand1 + operand2 * 3

operand2 := operand1 + 2;
(Error)Cannot use undefined variable.
ID: 2; CONST:1; OP: 1
target := operand1 + operand2 * 3
(Error)Cannot use undefined variable.
ID: 3; CONST:1; OP: 2
Result ==> operand2: Unknown; target: Unknown;
```

- 다르게, 저희 팀은 정의되지 않은 변수를 사용 시 Error를 출력했습니다.

### 입력 #3

```
→ project01 git:(main) x ./parser file.txt
operand1 := 1;
operand2 := (operand1 * 3) + 2 ;
target := operand1 + operand2 * 3

operand1 := 1;
(OK)
ID: 1; CONST:1; OP: 0
operand2 := ( operand1 * 3 ) + 2;
(OK)
ID: 2; CONST:2; OP: 2
target := operand1 + operand2 * 3
(OK)
ID: 3; CONST:1; OP: 2
Result ==> operand1: 1; operand2: 5; target: 16;
```

- 괄호 역시 잘 처리하는 모습을 볼 수 있습니다.

### 입력 #4

```
→ project01 git:(main) x ./parser file.txt
operand1 := 3 ;
operand2 := operand1 + + 2 ;
target := operand1 + operand2 * 3

operand1 := 3;
(OK)
ID: 1; CONST:1; OP: 0
operand2 := operand1 + 2;
(Warning)Operator is duplicated.
ID: 2; CONST:1; OP: 1
target := operand1 + operand2 * 3
(OK)
ID: 3; CONST:1; OP: 2
Result ==> operand1: 3; operand2: 5; target: 18;
```

- 연산자 중복을 잘 처리하는 것을 확인할 수 있습니다.



## Conclusion

해당 프로젝트를 진행하면서 다음과 같은 것들을 느낄 수 있었습니다.

- C++ 언어로 LL(1) parser를 개발하였습니다. 이는 C나 C++, 기타 다른 언어들을 인식하는 컴파일러를 만드는 것보다 굉장히 간단한 프로젝트에 속한다는 것을 체감하였습니다. 예전에 처음 언어가 만들어졌을 때에는, 어셈블리어로 초기 언어들을 인식하고 기계어로 바꾸는 컴파일러를 작성하였을 텐데, 개발하는 데 많은 어려움이 있었을 것이라고 사료됩니다.
- 새로운 언어를 만들어보고 싶은 욕심이 생겼습니다. 특정 도메인에서 사용되기 좋은 특정한 규칙을 가진 언어를 만드는 프로젝트는 굉장히 매력적이라는 생각이 들었습니다. PL과목에서 언어를 평가하는 요소와 구성하는 요소들을 배울 수 있었는데, 그러한 점을 잘 고려하여 새로운 프로그래밍 언어를 인식하는 컴파일러를 만든다면 (기계어로까지 변환되도록) 흥미로운 프로젝트가 될 것이라고 생각하였습니다. 현재 배우고 있는 MIPS 기반 컴퓨터 구조에서 활용 될만한 프로젝트를 제작해보려고 합니다.
- LL(1) parser는 하나의 input token을 바라보고 파싱을 하였습니다. 추후에는 재귀적 하향 파싱 방식이 아닌 테이블을 활용하는 방법, LR parser 관련한 알고리즘 (테이블을 만드는)에도 기회가 된다면 탐구해보고 싶습니다.