**컴파일러 Syntax analyzer 보고서**

**7조: 서정현, 이창민**

**목차**

1. **Ambiguity 제거**
2. **SLR Parsing Table 제작**
3. **구현**
4. **실행 방법 및 참고**
5. **Ambiguity 제거**

총 2개의 ambiguous한 rule을 발견했는데, PDF에 있는 5번 rule인 EXPR -> EXPR addsub EXPR | EXPR multdiv EXPR, 14번 rule인 COND -> COND comp COND | boolstr 부분에서 ambiguity를 확인하였다. 5번 rule의 경우는 사칙연산에 대한 우선순위가 정해지지 않아서 생기는 ambiguity이다. 예를 들어, id+id\*id는 곱셈 연산을 하는 두 id가 먼저 parse tree를 구성하고, 그 결과가 덧셈 연산의 id와 더해져야 하는데, 단순히 5번 rule만을 가지고는 정확하게 parse tree를 구성할 수 없다. 그렇기 때문에, 연산의 우선순위를 정하여서 ambiguity를 제거하였다. 5번 rule의 형태를 바꾸고 추가로 EXPR\_MD, EXPR\_T라는 rule을 추가하였다. 원래 EXPR은 EXPR -> EXPR\_MD addsub EXPR | EXPR\_MD로 형태를 바꾸었고, EXPR\_MD -> EXPR\_T multdiv EXPR\_MD | EXPR\_T, EXPR\_T -> lparen EXPR rparen | id | num의 형태를 가진다. 이 형태를 가지면 연산에 대한 우선순위가 곱셈과 나눗셈에 대해서 더 높게 설정되기 때문에 ambiguity를 제거할 수 있다. 또 하나의 ambiguity를 가진 rule인 14번은 같은 표현에 대해서 여러 parse tree의 형태가 나올 수 있다. 예를 들어, false==true!=true라는 식의 경우, 파싱 중에 COND comp boolstr과 boolstr comp COND라는 두 개의 중간 형태를 가지게 된다. 그래서 COND의 형태를 COND -> boolstr comp COND | boolstr의 형태로 바꿔서 단 하나의 parse tree의 형태가 생성되도록 하였다. Ambiguity를 제거한 CFG는 <Ambiguity_Fixed_CFG.docx> 라는 문서에 따로 작성하였다.

1. **SLR Parsing Table 제작**

과제 설명 PDF에 있는 웹사이트를 통해 CFG를 수정하지 않고 Parsing Table을 구축한 경우, ambiguity가 있는 rule이었기 때문에 ACTION Table에서 각 state에서의 action이 하나로 정해지지 않는 문제가 있었다. 그래서 ambiguity를 제거한 후 수정된 CFG를 바탕으로 Parsing Table을 다시 만들었더니, 문제가 사라진 것을 확인할 수 있었다. 전체 SLR Parsing Table은 <SLR_Parsing_Table.xlsx> 파일에 정리하였다. 시작 state에 임시 시작 state를 추가하고, ACTION이나 GOTO가 정해지지 않은 부분은 모두 reject의 대상이기 때문에 error로 설정하였다.

1. **구현**

Syntax Analyzer의 구현에 사용할 언어는 C++을 선택하였다. 개발 툴은 Visual Studio 2019와 Visual Studio Code를 사용하였다. 총 5가지의 cpp 파일을 제작하였다. 그 중 이름에 loader가 포함된 파일들은 syntax analyzer에 필요한 데이터를 읽어들이는 함수만을 가진다. 실질적인 analyzing을 하는 코드들은 main() 함수에 구현되어 있다.

1. **syntax\_analyzer.cpp**

프로그램이 실행될 때 필요한 파일들의 이름과 main() 함수가 구현되어 있다. 파일들을 저장하는 변수들은 모두 개발 중이나 프로그램이 실행 중일 때 값이 변하지 않도록 const로 선언하였다. 전체적인 흐름은 첫번째로 SLR Parsing Table, CFG, 그리고 Lexical Analyzer의 결과물 파일을 Load한다. line\_cnt는 오류가 발생했을 시 어디에서 오류가 났는지를 저장하는 변수로, 현재 검사 중인 line number, 즉 몇 번째 토큰인지를 저장한다. state\_stack은 지금까지 검사한 state들을 저장하는 stack이며, top에는 현재 state가 저장된다. next\_symbol에는 검사할 token이 symbol (terminal) 형태로 변환되어 저장된다. 그 후 동작 방식은 다음과 같다.

1. state\_stack에 초기 state인 0 푸시, next\_symbol 업데이트.
2. SLR Parsing Table에서 state\_stack.top() 상태에서 next\_symbol을 받았을 때 행동을 act에 받아온다.
3. act의 첫 문자에 따라 다음의 행동을 수행한다.

* 숫자로 시작하면 그 수를 state\_stack에 푸시한다. (GOTO)
* s로 시작하면 next\_symbol을 다음 토큰으로 업데이트하고, 수를 state\_stack에 푸시한다. (SHIFT)
* r로 시작하면 번호에 해당하는 CFG의 Transition을 참조하여 next\_symbol을 Trainsition의 LHS로 업데이트하고, RHS의 길이만큼 state\_stack에서 pop한다. (REDUCE)
* e로 시작하면 Syntax Error가 발생한 경우로, rejected 메시지와 함께 검사 중이었던 token과 그 번호를 출력하고 프로그램을 종료한다. (ERROR)
* a로 시작하면 Syntax Error가 발생하지 않은 입력이므로 Accepted 메시지를 출력하고 반복을 종료한다. (ACCEPTED)

1. 2~3번 과정을 에러가 발생하거나 입력의 끝을 만날때까지 반복하고, 상황에 맞는 출력을 콘솔 창에 띄운다.
2. **token.h & token.cpp**

미리 실행된 lexical analyzer에서 결과물로 만들어진 output.txt 파일을 해석하기 위하여, 저번 과제에서 구현한 lexical analyzer의 소스코드 중 token 타입을 가져와 사용한다. 헤더에는 token들의 종류가 enum 타입으로 선언되어 있다. 추가로 token의 이름과 cfg에서 사용하는 state의 이름을 일치시키기 위한 변환 함수가 구현되어 있다. tokenToTerminal() 함수를 호출하게 되면 인자로 주어진 token을 terminal state의 이름을 값으로 갖는 문자열로 변환하여 반환한다. 문자열을 다루기 때문에 C++의 STL 중 unordered\_map을 사용하여 O(1)의 시간복잡도로 문자열 탐색을 하도록 구현하였다.

1. **table\_loader.h & table\_loader.cpp**

파일 입력, 그중에서도 Parsing Table만을 위해서 기능 별로 분리해놓은 부분이다. 헤더 파일에는 SLR Parsing Table을 읽어서 저장하는 함수와 Parsing Table을 구축하면서 사용한 CFG에서의 terminal, non-terminal state들의 이름들이 모두 저장되어 있다. 파일의 내용인 문자열들을 그대로 읽어들여 unordered\_map에 저장한 후 반환한다.

1. **input\_loader.h & input\_loader.cpp**

lexical analyzer의 출력을 파일로 입력받아 syntax anlyzer가 사용 가능한 형태로 변환하는 함수인 getInput() 함수가 헤더에 선언되어 있다. 또한 cpp 파일에는 헤더 파일에는 선언되지 않은 문자열을 다루기 위한 함수들이 정의되어 있는데, 전체 프로젝트의 다른 부분에서 쓰이지 않을 것이라고 판단하여 이 cpp 파일 안에서만 선언하였다. 입력 파일의 이름을 인자로 받아서 파일을 읽은 후, lexical analyzer에서 출력을 위해서 같이 쓰여 있던 ‘<’, ‘>’을 없애고, 남은 문자열에서 정보를 분리하여 token 클래스로 변환한다. 변환된 Token list의 끝에 ‘$’(ENDMARKER)를 추가하여 main() 함수에 반환하게 된다.

1. **cfg\_loader.h & cfg.loader.cpp**

이 부분도 마찬가지로 CFG에 대한 정보를 읽어 들이기 위해서 분리한 부분이다. 헤더에는 Transition Rule을 저장하기 위한 구조체가 선언되어 있고, 파일에서 문자열을 읽어 들여 lhs와 rhs\_len을 저장하는데, lhs는 transition의 좌측에 해당되는 non-terminal symbol이 저장되며, rhs\_len에는 transition의 우측에 RHS의 길이를 저장하여 반환한다.

1. **실행 방법 및 참고**
2. **Lexical Analayzer 실행**

제출물의 Binary\_File\_and\_Test\_Input\_and\_Output 폴더에서 lexical-analyzer [파일이름]을 입력하면 lexical analyzer가 실행된다. 만약 오류가 있다면, 오류 메시지를 출력하고 결과물을 만들지 않는다. 문제가 없다면, output.txt라는 이름의 결과물을 출력한다.

1. **Syntax Analyzer 실행**

제출물의 Binary\_File\_and\_Test\_Input\_and\_Output 폴더에서 syntax-analyzer [파일이름]을 입력하면 syntax analzyer가 실행된다. 만약 lexical analyzer와 연결하여 사용한다면 output.txt를 파일 이름으로 입력하면 된다. 오류가 있을 경우, 오류 메시지를 출력하면서 Reject된다. 오류가 없다면, Accepted를 출력하고 종료된다.

1. **참고**

**Github 주소**

1. **Lexical Analyzer**

https://github.com/ChangminYi/compiler-lexical-analyzer

1. **Syntax Analyzer**

https://github.com/ChangminYi/compiler-syntax-analyzer