컴파일러 Syntax analyzer 보고서

7조: 서정현, 이창민

목차

1. Ambiguity 제거
2. SLR Parsing Table 제작
3. 구현
4. Ambiguity 제거

전체 rule들 중에서 총 2개의 ambiguous한 rule을 발견했는데, PDF에 있는 5번 rule인 EXPR -> EXPR addsub EXPR | EXPR multdiv EXPR과 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, (false==true)!=true와 false==(true!=true)라는 두 개의 형태로 표현이 가능하게 된다. 그렇기 때문에, 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 별도로 작성하여 정리하였다.

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 않고 체크박스가 생기는 문제가 있었다. 그래서 ambiguity를 제거한 후 수정된 CFG를 바탕으로 Parsing Table을 다시 만들었더니, 문제가 사라진 것을 확인할 수 있었다. 그리고 SLR 방식으로 parsing을 하기 위해서, 기존의 시작 state인 CODE 앞에 임시 시작 state를 추가하여 테이블을 구축하였다. 전체 SLR Parsing Table은 SLR\_Parsing\_Table.xlsx 파일에 정리하였다. 테이블에서 shift, reduce, goto, accept의 값이 정해지지 않고 비어있는 부분은 모두 에러, 즉 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 부분은 main() 함수에 구현되어 있다.

1. syntax\_analyzer.cpp

프로그램이 실행될 때 필요한 파일들의 이름과 main() 함수가 구현되어 있다. 읽어 올 파일들의 이름은 개발 중, 혹은 프로그램이 실행 중일 때 바뀌지 않도록 모두 const 선언을 하였다. 프로그램이 실행될 때 필요한 모든 변수를 main() 함수에서 저장하고 있다. ACTION과 GOTO 테이블의 정보를 가지고 있는 parse\_table, reduce 동작에서 이동해야 할 state값과 stack에서 pop할 횟수를 저장하고 있는 cfg, 그리고 분석해야 할 token 정보인 token\_input을 변수로 가진다. syntax analyzer가 동작할 때 non-terminal과 terminal의 이름을 Parsing Table에 매칭하는 방법으로 해싱을 사용하기로 하였다. 이 방식을 사용하면 따로 문자열을 변환하는 함수를 구현할 필요 없이 바로 테이블에 접근할 수 있다. 직접 해싱 함수를 구현하는 것보다는 C++의 STL 중 unordered\_map을 사용하여 string을 ACTION과 GOTO로 바로 매핑하여 O(1)의 시간복잡도로 탐색을 하도록 구현하였다. 정보를 모두 읽어오면 syntax analyze를 하는 while문으로 진행한다. 처음 if-else문에서는 act 변수의 값에 따라 ACTION과 GOTO로 진행을 분리하였다. GOTO로 진행할 경우 state 값만을 업데이트하고 다시 진행하게 된다. ACTION으로 진행한 경우는 shift, reduce, error, accept 네 개의 상태를 구분하는 switch문으로 진입한다. 현재 act값의 첫 번째 글자가 s인 경우는 shift이므로, 다음 토큰을 받아오고 state를 최신화한 다음 다시 진행한다. 만약 글자가 r이라면 reduce이므로, cfg 변수를 참조하여 next\_symbol 값을 최신화하고 stack에서 rule의 RHS의 길이, 즉 stack에서 pop될 횟수만큼 pop을 실행한다. 만약 e라면 error인 경우이므로, Reject 메시지를 출력하고 즉시 프로그램을 종료한다. 그리고 a인 경우는 accept이므로, Accept 메시지를 출력하고 while문을 탈출한다.

1. token.h & token,cpp

미리 실행된 lexical analyzer에서 결과물로 만들어진 output.txt 파일을 해석하기 위하여 lexical analyzer에서 구현한 token 클래스와 token의 종류를 표현하는 enum을 공유한다. 단지 입력의 끝을 나타내기 위한 ENDMARKER 타입 하나만 추가하였다. 추가로 token의 이름과 cfg에서 사용하는 state의 이름을 맞춰주기 위한 변환 함수가 구현되어 있다. tokenToTerminal() 함수를 호출하게 되면 인자로 주어진 token을 terminal state의 이름으로 변환하여 반환한다. 반환한 값은 main() 함수에서 next\_symbol을 최신화하는 과정에서 사용한다.

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

(추후 추가 예정)

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

lelxical analyzer의 출력을 파일로 입력받아 syntax anlyzer가 사용 가능한 형태로 변환하는 함수인 getInput() 함수가 헤더에 선언되어 있다. 하지만 cpp 파일에는 헤더 파일에는 선언하지 않은 문자열을 다루기 위한 함수들, strip()과 trim()이 정의되어 있는데, 전체 프로젝트의 다른 부분에서 사용하지 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. 변환한 token을 C++의 STL 중 list를 사용하여 저장한다. token을 추가, 삭제, 그리고 참조하는 작업은 모두 정보의 head와 tail에서만 이루어지기 때문에 array나 vector를 사용하는 것보다 효율적인 list를 사용하였다. 결과적으로는 구축이 완료된 list에 ENDMARKER를 추가하여 main() 함수에 반환하게 된다. 반환한 값은 main() 함수에서 analyzing 과정에서 직접 사용하게 된다.

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

(추후 추가 예정)