ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 47, No. 1, pp. 52-60, 2020. 1
https://doi.org/10.5626/JOK.2020.47.1.52

LR 오토마타 생성 모듈을 공유하고 범용 프로그래밍언어로 명세를 작성하는 파서 생성 도구

(Parser Generators Sharing LR Automaton Generators and Accepting General Purpose Programming Language-based Specifications)

임 진 택 † 김 가 영 †† 신 승 현 †† 최 광 훈 ††† 김 익 순 ††††
(Jintaeck Lim) (Gayoung Kim) (Seunghyun Shin) (Kwanghoon Choi) (Iksoon Kim)

요 약 본 논문은 LR 파서를 쉽게 개발하기 위하여 두 가지 아이디어를 제안한다. 첫째, 오토마타 생성을 모듈화하여 새로운 프로그래밍 언어를 위한 파서 생성 도구를 쉽게 개발 할 수 있다. 둘째, 파서 명세를 일반 프로그래밍언어로 작성하도록 구성하여 이 언어 개발 환경에서 제공하는 구문 오류, 자동 완성, 타입 오류 검사 기능들을 이용하여 파서 명세의 오류를 바로잡을 수 있다. 이 연구에서 제안한 아이디어로 Python, Java, C++, Haskell로 파서를 작성할 수 있는 도구를 개발하였고, 실험을 통하여 위 두 가지 장점을 보였다.

키워드: LR 파서, 파서 명세 언어, 프로그래밍언어, 컴파일러

Abstract This paper proposes two ways to develop LR parsers easily. First, one can write a parser specification in a general programming language and derive the benefits of syntax error checking, code completion, and type-error checking over the specification from the language's development environment. Second, to make it easy to develop a parser tool for a new programming language, the automata generation for the parser specifications is in a modular form. With the idea proposed in this study, we developed a tool for writing parsers in Python, Java, C++, and Haskell. We also demonstrated the two aforementioned advantages in an experiment.

Keywords: LR parser, parser specification language, programming language, compiler

· 이 논문은 2019년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(NRF-2019R1I1A3A01058608)
· 본 연구는 문화체육관광부 및 한국콘텐츠진흥원의 2019년도 문화기술연구개발 지원사업으로 수행되었음(R2018030393)
· 이 논문은 2019 한국컴퓨터종합학술대회에서 'LR 오토마타 생성과 파서 도구의 분리를 통한 효율적 파서 개발 방법'의 제목으로 발표된 논문을 확장한 것임

논문접수 : 2019년 8월 27일
(Received 27 August 2019)
논문수정 : 2019년 10월 21일
(Revised 21 October 2019)
심사완료 : 2019년 10월 24일
(Accepted 24 October 2019)

† 학생회원 : 전남대학교 전자컴퓨터공학부 학생
wlsxor13@naver.com
†† 비 회 원 : 전남대학교 전자컴퓨터공학부 학생
kirayu15@gmail.com
tlsdorye@gmail.com
††† 정 회 원 : 전남대학교 전자컴퓨터공학부 교수(Chonnam Nat'l Univ.)
kwanghoon.choi@jnu.ac.kr
(Corresponding author임)
†††† 비 회 원 : 한국전자통신연구원 통신미디어연구소 책임연구원
ik-soon.kim@etri.re.kr

정보과학회논문지 제47권 제1호(2020. 1)

1. 서 론

LR 구문 분석 방법은 보편적인 컴파일러 기술이 되었음에도[1-8]. 불구하고, 이 방식으로 파서를 작성하려면 복잡하고 시간이 오래 걸린다. 여전히 쉽게 파서 개발 방법에 대한 연구가 진행되고 있다[9].

이 논문은 LR 방식의 파서를 쉽게 개발하는 방법에 관한 것이다. LR은 상향식으로 리프 노드부터 루트 노드 순서로 트리를 만드는 파서 방식이다. 프로그래밍언어의 문법을 대부분 수정 없이 처리할 수 있는 장점이 있다.

하지만 기존의 LR 파서 생성 도구 방식은 제한된 파서 명세 언어를 이용해 파서 명세를 작성해야하고 특정 프로그래밍 언어로 작성된 파서만 개발할 수 있다. 예를 들어, C/C++를 위한 YACC, O'Caml을 위한 ocamlyacc, Haskell을 위한 Happy의 경우 각각 고유의 파서 명세 언어를 이용하고 해당 프로그래밍 언어로 작성된 파서만 만든다. 따라서 이외의 다른 프로그래밍 언어를 위한 LR 파서 생성 도구를 새로 만들어야 한다. 또한 제한된 파서 명세 언어로 파서를 작성하기 때문에 이 파서 명세를 작성할 때 발생할 수 있는 오류를 점검하기 위한 개발 환경이 없고, 파서를 모듈화 하여 작성하는 방법도 부족하다.

본 논문은 앞에서 서술한 기존의 LR 파서 생성 도구들의 단점을 보완하는 두 가지 방법을 제안한다. 첫째, LR 오토마타 생성을 파서 생성 도구와 분리 구성한다. 새로운 파서 생성 도구를 만들 때 구현 언어에 독립적인 LR 오토마타 생성 모듈을 재사용하면 구문 트리를 만드는데 더 집중할 수 있다.

둘째, 일반 프로그래밍언어로 LR 파서 명세를 작성하도록 한다. 일반 프로그래밍언어에서 지원하는 함수나 클래스를 사용하면 쉽게 문법을 나누어 작성하고 조합할 수 있다. 그리고 파서 명세를 작성할 때 해당 프로그래밍언어의 개발 환경을 이용하면 구문 오류나 타입 오류를 사전에 검사할 수도 있다.

SWLAB 파서 생성 도구로 명명한 개발 방법을 제안하고, Java, Python, Haskell, C++를 파서 구현 언어로 하는 파서 생성 도구를 개발하는 실험을 통하여 앞에서 설명한 두 가지 장점을 설명한다.

본 논문의 구성은 다음과 같다. 2절에서 관련 연구를 설명한다. 3절에서 SWLAB 파서 생성 도구를 통하여 제안한 아이디어의 구현 방법을 설명하고, 4절에서 파서 명세 언어로써 Python, Java, Haskell, C++의 4가지 프로그래밍언어의 차이점에 대해 논의하고, 5절에서 결론을 맺는다.

2. 관련 연구

표 1은 LR 파서 생성 도구의 대표적 사례들을 나열한다. 파서 명세 언어를 작성하면 파서 구현 언어로 작성된 파서 코드를 생성하는 일반적인 방식과 동일한 프로그래밍언어로 명세도 작성하고 구현하는 방식인 직접 구현하는 방식으로 구분한다.

표 1 LR 파서 툴의 예시
Table 1 Examples of LR parser tools

Code generation method	Direct implementation
YACC(C/C++) HAPPY(Haskell), CUP(Java)	PLY(Python)

파서 명세로부터 파서 코드를 생성하는 방식의 파서 생성 도구들은 문법을 나누어 작성하고 조합하는 방법을 지원하지 않았다[10]. 따라서 파서를 작성할 때 더 복잡하고 유지보수하기 어렵다. 예를 들어 간단한 특징에서 복잡한 특징까지 차례로 추가하며 프로그래밍언어 해석기를 만드는 과정[11]이나 타입 특징을 점차 추가하며 타입 검사기를 만드는 과정[12]에서 모듈화 문법 작성 방법이 없기 때문에 겹치는 부분은 동일한 문법 명세를 반복해서 파서를 작성해야 한다.

동일한 프로그래밍언어로 명세도 작성하고 구현하는 방식인 PLY[13]은 본 논문에서 지향하는 파서 생성 도구의 구성을 가지고 있다. Python으로 파서 명세를 작성할 때 1) Python 언어에서 제공하는 클래스와 함수를 사용하여 모듈화된 문법을 쉽게 작성할 수 있고, 2) 풍부한 Python 개발 환경을 이용하면 구문 오류 검사, 자동 완성, 타입 오류 검사와 같은 기능을 파서 명세 작성할 때 모두 사용할 수 있다.

PLY는 구문 분석 기능(YACC)과 어휘 분석 기능(LEX)을 라이브러리로 제공한다. 그림 1은 PLY를 이용하여 YACC 기능을 이용한 예제 소스이다. PLY에서 제공하는 YACC 모듈을 라이브러리로 불러와 쉽게 사용할 수 있다. 어휘 분석 기능도 역시 동일하게 쉽게 사용가능하다.

PLY가 기존 YACC/LEX과 가장 다른 점은 파서 명세의 형식이다. YACC의 경우 자체 파서 명세 언어 형식에 따라 파서 명세를 작성하고 이를 YACC 도구를 적용하여 파서 구현 언어인 C언어 파서 코드를 만든다. 하지만 PLY의 경우 파이썬 형식으로 파서를 명세하고 이를 통해 파이썬 형식의 파서를 작성한다.

하지만 PLY 도구의 방식이 가능했던 이유는 Python 언어에서 프로그램이 자신의 구문 트리를 스스로 다룰 수 있는 Reflection 특징을 제공하기 때문이다. 그림 1에서 보여주듯이 PLY 도구를 사용하여 파서를 작성할 때 생산 규칙을 각 함수의 첫 번째 문장에 문자열 형식으로 작성한다. PLY 파서 생성 도구는 이 예제 소스의 구문 트리를 받아 파서를 만든다. 이때 각 함수의 첫 번째

```
import ply.yacc as yacc
import mylexer          # Import lexer information
tokens = mylexer.tokens   # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''
def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''
```

그림 1 PLY 예제 소스
Fig. 1 PLY example source

문장의 생산규칙을 모아 전체 문법을 구성하고 그 문장을 제외한 나머지는 해당 생산규칙을 적용할 때마다 실행하는 액션 함수로 사용한다. 따라서 이 Reflection 특징을 제공하지 않는 Java, C++, Haskell 언어에서 PLY 구현 방식을 그대로 따를 수 없다.

표 1에서 나열한 예시 외에도 매우 다양한 LR 파서 생성 도구들이 개발되어 왔다[14]. 본 연구 내용과 관련 있는 사례들을 몇 가지 살펴보면 다음과 같다. CL-Yacc Lisp은 매크로를 사용하여 리스프 프로그램에 문법을 작성할 수 있다. CookCC는 자바의 주석(annotation)을 이용하여 문법의 생산 규칙을 기술하고 메소드로 액션 함수를 기술하였다. 따라서 주석을 처리하는 별도의 처리기가 필요하다. Irony는 C# 클래스로 터미널, 넌터미널, 생산 규칙을 표현하는 클래스를 미리 준비해놓고 파서 명세를 작성하게 하였다. NLT 역시 비슷한 방식을 취한다. Parse는 C++ 템플릿을 이용하여 문법을 작성하고 메타프로그래밍으로 파서 생성 도구를 만드는 방법을 가지고 있다.

LL 파서의 경우 파서 명세에서 파서 코드를 만드는 방식보다 파서 코드를 직접 구현하는 방식이 일반적이다. LL 파서는 루트 노드에서 리프 노드 순서로 파서 트리를 만드는 하향식 파서(top-down parser) 방식이다. 대표적인 예로, 재귀적 하강 파서(Recursive descent parser)와 파서 컴비네이터(Parser combinator)가 있다. 이 논문 연구에서 추구하고자 하는 목적과 유사하게 별도의 파서 명세 언어를 사용하지 않고 일반 프로그래밍 언어를 사용하여 직접 파서를 작성한다. 재귀적 하강 파서는 문법의 각 넌터미널(Non-terminal)을 구현하는 함수들을 모아 파서를 구성하는 방식이다. 파서 컴비네이터는 파서 함수들을 조합하여 새로운 파서를 만드는 고차원 함수로, 파서 개발자는 이 파서 컴비네이터를 이용하여 파서를 구성한다. 재귀적 하강 파서와 파서 컴비네이터는 모두 일반 프로그래밍언어를 사용하여 파서를 개발하기 때문에 해당 언어 프로그래밍 환경의 기능을 사용할 수 있다는 점에서 이 연구에서 추구하는 목적과 동일하다.

하지만 이 LL 파서 방법들은 두 가지 문제점이 있다. 첫째, 왼쪽 재귀 문법(예: A -> Aα | β)을 구현할 때 재귀 함수 호출이 무한 반복될 수 있기 때문에 수작업으로 왼쪽 재귀가 없는 문법으로 수정한 다음(예: A -> β R, R -> αR | ε) 이를 구현해야 한다. 프로그래밍언어의 문법에서 왼쪽 재귀 문법의 패턴은 매우 빈번하게 나타난다. 둘째, LR 파서의 경우 선언적 문법과 구문 분석 액션을 명확히 구분하여 구문 분석 대상 언어의 범위가 분명하지만 LL 파서 방식에서와 같이 문법과 구문 분석 액션을 모두 함수로 구현하면 문법이 지정하는 언어의 범위가 모호해져서 파서의 유지보수가 어렵다.

LL 파서 방식도 ANTLR4를 비롯한 다양한 파서 생성 도구들이 개발되었다[14]. 이 파서 생성 도구들은 앞에서 설명한 LL 방식의 한계를 여전히 지니고 있다.

이 논문의 목적은 파서 생성 도구에서 LR 오토마타 생성을 분리하여 파서를 빠르게 개발할 수 있는 방법을 제안하는 것이다. 그리고 PLY의 경우와 달리 Python 뿐만 아니라 어떠한 프로그래밍언어, 예를 들어 Java, C++, Haskell 언어에도 적용되는 파서 생성 도구의 구조를 설계하는 것이다.

3. SWLAB 파서 생성 도구

본 논문에서 제안하는 파서 생성 도구인 SWLAB 파서 생성 도구의 아키텍쳐를 설명하고 다양한 개발 환경 중 Python에서 파서 작성을 통해 작동 과정을 설명한다.

3.1 SWLAB 파서 생성 도구 아키텍쳐

그림 2는 이 연구에서 제안하는 도구의 구조이다. 파서 개발자는 파서 명세를 일반 프로그래밍언어로 작성하며, 파서 라이브러리를 통해 공통 LR 오토마타 모듈 생성을 사용하여 파서를 준비하고, 입력 프로그램을 받아 구문 분석을 진행하고 추상구문트리를 낸다.

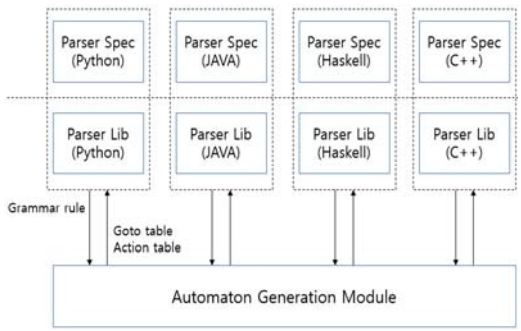

그림 2 SWLAB 파서 생성 도구 구조
Fig. 2 SWLAB parser tool architecture

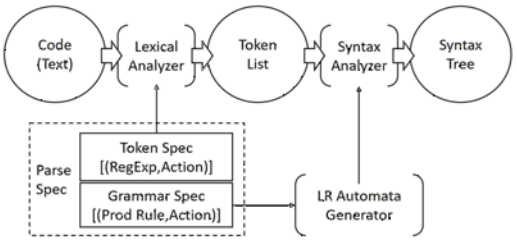

그림 3 구문 분석 과정 및 구성
Fig. 3 Parsing process and configuration

특정 프로그래밍언어를 가정하고 그림 2의 파서 생성 도구를 활용하여 파서를 작성했을 때 구조는 그림 3과 같다. 파서 명세는 토큰 명세와 문법 명세로 구성되어 있다. 토큰 명세로부터 어휘 분석기를 만들고 문법 명세로부터 구문 분석기를 만든다. 입력 프로그램 텍스트를 받은 어휘 분석기는 토큰 리스트를 만들고, 다시 이를 구문 분석기를 거치면 입력 프로그램의 최종 구문 트리가 된다.

3.1.1 파서 명세

파서 명세는 어휘 분석(lexical anaysis)을 위한 토큰 명세와 구문 분석(syntax analysis)을 위한 문법 명세로 구성되어 있다. 이 논문에서 제안하는 툴에서는 파서 개발자로 하여금 아래의 타입에 맞추어 파서 명세를 작성하도록 정의하였다.

```
ParserSpec ::= (TokenSpec, GramSpec)
TokenSpec ::= [ (RegExpr, LexAction) ]
GramSpec ::= [ (ProdRule, RuleAction) ]
RegExpr, ProdRule ::= String
LexAction ::= String -> Token
RuleAction ::= () -> AST
```

토큰 명세는 정규식(regular expression)과 토큰 생성 액션의 쌍들의 리스트이고 문법 명세는 생산규칙(production rule)과 구문 트리 생성 액션의 쌍들의 리스트이다.

("\(", lambda text: Token.OPEN_PAREN)는 파이썬으로 작성한 소괄호 열기 토큰의 정규식과 해당 토큰(OPEN_PAREN)을 생성하는 람다 액션 함수의 쌍 예제이다. 이때 인자 text 문자열은 변수 토큰과 같이 토큰의 종류를 구분할 뿐만 아니라 텍스트 내용도 필요할 때 사용한다.

생산규칙과 구문 트리 생성 액션의 예를 보면, ("AdditiveExpr -> AdditiveExpr + MultiplicativeExpr", lambda: begin(BinOp (BinOp.ADD, lib.get(1), lib.get(3))))과 같다. 생산규칙은 화살표 왼쪽에 넌터미널을 배치하고 화살표 오른쪽에 임의의 터미널(Terminal)과 넌터미널들을 나열한 형식의 문자열이다. 이 생산규칙으로 덧셈 연산의 왼쪽 피연산자로 넌터미널 AdditiveExpr의 구문 트리를, 오른쪽 피연산자로 넌터미널 MultiplicativeExpr의 구문 트리를 갖는 덧셈식 구문트리를 만든다. 파이썬 람다로 액션을 작성하여 생산규칙에 해당하는 구문 트리를 만들어 반환한다. 덧셈식을 구성하는 파이썬 클래스 BinOp 생성자에 연산 종류를 지정하는 파이썬 상수 BinOp.ADD와 lib.get(1)과 lib.get(3)로 가져온 두 개의 덧셈 피연산자 서브 구문트리들을 지정하여 덧셈식 구문트리를 만든다. 라이브러리 lib.get은 생산규칙의 오른쪽에서 지정한 숫자에 위치한 심볼의 구문트리를 가져오는 함수다.

파서 명세에서 참조하는 Token 타입과 AST 타입은 파서 개발자가 어휘 분석과 구문 분석에 따라 정한다. 따라서 임의의 파서 명세를 처리하는 파서 라이브러리는 이 타입들을 어떻게 정의하는지 상관없는 독립적인 구조를 갖추어야 한다. 3.1.2절에서 이와 관련하여 추가로 논의한다.

앞서 설명한 타입 형식에 맞는 파서 명세를 Python, Java, Haskell, C++으로 쉽게 작성할 수 있음을 실험을 통하여 확인하였다. 이러한 파서 명세를 작성하기 위해 필요한 최소 요구사항은 액션 함수를 작성하기 위한 람다 지원 여부이다. Python 2.x, Java1.8, C++11, (함수형 언어는 기본적으로 람다를 지원하지만) Haskell98이나 그 이후의 버전을 사용하면 모두 람다를 지원한다. 현대 객체 지향 언어의 경우 대부분 람다를 도입하는 추세이기 때문에 논문에서 제안하는 파서 명세를 작성하는 프로그래밍언어로 사용할 때 특별한 문제가 없을 것으로 판단한다.

다음 절에서 파서 명세에 따라 구문 분석을 진행하기 위해 필요한 오토마타 생성 공통 모듈과 파서 라이브러리를 설명한다.

3.1.2 파서 라이브러리와 오토마타 생성 공통 모듈

앞 절에서 설명한 파서 명세를 지원하기 위해 파서 라이브러리와 오토마타 생성 공통 모듈을 도입한다. 파서 라이브러리는 아래의 두 함수를 지원한다.

```
tokenLib :: ∀token. TokenSpec -> [Char] -> [token]
ruleLib :: ∀token.∀ast. GramSpec -> [token] -> ast
```

함수 tokenLib는 첫 번째 인자로 토큰 명세를 받아 어휘 분석기를 만들고, 두 번째 인자로 구문 분석할 소스 프로그램 텍스트를 받아 토큰 리스트를 낸다. 함수 ruleLib도 비슷한 방식으로 동작한다. 첫 번째 인자로 문법 명세를 받아 구문 분석기를 만든다. 이때 오토마타 생성 공통 모듈을 이용한다. 두 번째 인자로 토큰 리스트를 받은 다

음 생성된 오토마타를 이용하여 구문 트리를 만든다.

위 두 함수의 타입에서 토큰 타입과 구문 트리 타입은 ∀로 한정된 다형 타입이다. 파서 개발자가 어떤 토큰 타입과 구문 트리 타입을 정의하더라도 동작할 수 있어야 하기 때문이다. 이 논문의 실험에서 Java와 C++ 프로그래밍언어로 두 라이브러리 함수를 구현할 때 제네릭과 템플릭을 이용하였고, Haskell의 경우는 다형 타입을 이용하면 이 함수들을 구현할 수 있었다. 동적 타입 시스템의 Python에서는 다형 타입에 대해 특별히 고려하지 않아도 되었다.

함수 ruleLib는 문법 명세를 받은 다음 오토마타 생성 공통 모듈을 이용하여 지정한 문법으로 부터 액션 테이블과 Goto 테이블을 생성한다. 이 모듈은 전통적인 LR 파서 테이블 생성 알고리즘을 사용한다. 그 다음 이 두 가지 테이블의 오토마타를 이용하여 구문 트리를 만든다.

이 두 함수를 이용하여 구문 분석하는 과정은 다음과 같다. 첫째, 함수 tokenLib에서 주어진 입력 문자열(의 prefix)과 매치하는 토큰 명세의 정규식을 찾을 때마다 해당하는 액션 함수를 호출하여 토큰을 생성한다. 만일 그러한 정규식을 토큰 명세에서 찾을 수 없다면 어휘 분석 에러를 낸다.

둘째, 함수 ruleLib는 먼저 함수 tokenLib를 이용하여 입력 문자열을 토큰 리스트로 변환하고, 이 토큰 리스트를 문법 명세의 생산규칙에 따라 구문 트리를 만든다. 토큰 리스트로부터 구문 트리를 만들 수 있는 특정 생산규칙을 찾을 때마다(reduce) 해당 액션을 호출하여 구문 트리를 만든다.

오토마타 생성 공통 모듈은 문법 명세를 입력으로 받아 오토마타 테이블(Action과 GoTo)을 출력한다. LR 문법에서 오토마타를 생성하는 전통적인 알고리즘을 사용하였다. 각 프로그래밍언어로 이 모듈을 공유하도록 파서 라이브러리 함수 tokenLib와 ruleLib를 쉽게 작성할 수 있다.

마지막으로 그림 3의 파서 생성 도구 구조는 프로그램 소스를 파싱할 때마다 파서 명세로부터 LR 오토마톤을 새로 만들도록 구성되어 있다. 하지만 파서 명세를 변경하지 않으면 파싱할 때마다 LR 오토마톤을 다시 만들 필요가 없다. 따라서 이러한 문제를 해결하기 위하여 LR 오토마톤을 파일로 저장하고 파서 명세가 변경되지 않으면 새로 만들지 않고 파일에서 오토마톤을 읽도록 최적화하였다. 파서 명세 중 생산 규칙 문자열들을 입력으로 해쉬 값을 계산해서 비교하면 생산 규칙이 변경되었는지 여부를 쉽게 확인할 수 있다.

3.2 SWLAB 파서 생성 도구 기반 파서 작성 예시

앞에서 설명한 아키텍처의 파서 생성 도구를 활용하여 파서를 만드는 과정을 예시로 설명한다. 간단한 Arith 언어를 가정하고 어휘 분석기와 구문 분석기를 작성하고 실행하는 예를 살펴본다.

3.2.1 프로그래밍언어 Arith

Arith는 4가지 정수 연산(덧셈, 뺄셈, 곱셈, 나눗셈)과 변수를 지원하는 간단한 언어이다. 괄호(“(”, “)”)를 통해 먼저 계산할 식을 지정하고, 세미콜론(“;”)을 통해 식과 식을 구분한다. 예시는 아래와 같다.

```
test = 100 * 200 - ( 800 / 400 ) ;
test = test + 123
```

3.2.2 어휘 분석기 및 구문 분석기 작성 방법

어휘 분석기는 문자 리스트를 입력 받아 토큰(Token) 리스트를 출력한다. Arith 언어에 필요한 토큰은 파이썬 클래스 Token에 상수로 IDENTIFER, NUMBER, EQ, MUL, SEMICOLON 등을 먼저 정의해두었다.

그림 4는 어휘 분석기 실행 예를 보여준다. 왼쪽 Arith 프로그램 예제를 입력받아 오른쪽과 같이 18개의 토큰으로 구성된 리스트를 출력한다.

그림 5는 Python으로 작성한 Arith 언어의 어휘 분석기의 토큰 명세를 보여준다. 앞서 정의한 Arith 언어의 각 토큰의 정규식과 해당 액션의 쌍을 라이브러리 함수로 lib.lex로 등록하여 토큰 명세를 작성한다. 이 정규식에 매칭되는 문자 시퀀스를 찾으면 해당 액션을 호출하여 해당하는 토큰을 만든다.

구문 분석기는 어휘 분석기에서 생성한 토큰(Token) 리스트를 입력 받아 AST를 출력한다. 그림 6은 Python으로 작성한 Arith 언어 구문 분석기의 문법 명세를 보여준다.

Arith 프로그램의 구문 트리를 구성하기 위해 파이썬 클래스 AST를 상속받은 클래스 Expr, Assign, BinOp, Lit, Var를 먼저 작성하였다. 파서 명세에서 이 클래스의 객체를 구문 트리 노드로 삼고 이 노드들을 조합하여 Arith 프로그램의 전체 구문 트리를 만든다.

Code	Token list
test = 100 * 200 - (800 / 400)	Token.IDENTIFIER Token.EQ Token.INTEGER_NUMBER Token.MUL Token.INTEGER_NUMBER Token.SUB Token.OPEN_PAREN Token.INTEGER_NUMBER Token.DIV Token.INTEGER_NUMBER Token.CLOSE_PAREN Token.END_OF_TOKEN

그림 4 Arith 어휘 분석기의 입력과 출력
Fig. 4 Input and output of Arith’s lexer

```
class Lexer:
  def __init__(self);
    lib = CommonParserUtil()
    lib.lex( "[\s]", lambda text: None)
    lib.lex( "[0-9]+", lambda text:
                        Token.INTEGER_NUMBER)
    lib.lex( "\(", lambda text: Token.OPEN_PAREN)
    lib.lex( "\)", lambda text: Token.CLOSE_PAREN)
    lib.lex( "\+", lambda text: Token.ADD)
    lib.lex( "\-", lambda text: Token.SUB)
    lib.lex( "\*", lambda text: Token.MUL)
    lib.lex( "\/", lambda text: Token.DIV)
    lib.lex( "\=", lambda text: Token.EQ)
    lib.lex( "\;", lambda text: Token.SEMICOLON)
    lib.lex( "[a-zA-Z]+[a-zA-Z0-9]*", lambda text:
                        Token.IDENTIFIER)
```

그림 5 Arith 어휘 분석기의 명세
Fig. 5 Arith's lexer specification

```
class Parser:
 def __init__(self):
  lib = CommonParserUtil();
  lib.rule( "SeqExpr' -> SeqExpr",
    lambda : begin(lib.get(1)))
  lib.rule( "SeqExpr -> SeqExpr ; AssignExpr",
    lambda : begin(seqexpr.append(lib.get(3)),
                   seqexpr))
  lib.rule( "SeqExpr -> AssignExpr",
    lambda: begin(seqexpr.append(
                   lib.get(1)), seqexpr))
  lib.rule( "AssignExpr ->
              identifier = AssignExpr",
    lambda: begin(Assign(lib.getText(1), lib.get(3))))
  lib.rule( "AssignExpr -> AdditiveExpr",
    lambda: begin(lib.get(1)))
  lib.rule( "AdditiveExpr ->
              AdditiveExpr + MultiplicativeExpr",
    lambda: begin(BinOp
                (BinOp.ADD, lib.get(1), lib.get(3))))
  lib.rule( "AdditiveExpr -> MultiplicativeExpr",
    lambda: begin(lib.get(1)))
  lib.rule( "MultiplicativeExpr ->
              MultiplicativeExpr * PrimaryExpr",
    lambda: begin(BinOp
              (BinOp.MUL, lib.get(1), lib.get(3))))
  lib.rule( "MultiplicativeExpr -> PrimaryExpr",
          lambda: begin(lib.get(1)))
  lib.rule( "PrimaryExpr -> identifier",
          lambda: begin(Var (lib.getText(1))))
  lib.rule( "PrimaryExpr -> integer_number",
          lambda: begin(Lit (int(lib.getText(1)))))
  lib.rule( "PrimaryExpr -> ( AssignExpr )",
          lambda: begin(lib.get(2)))
```

그림 6 Arith 구문 분석기의 명세
Fig. 6 Arith's parser specification

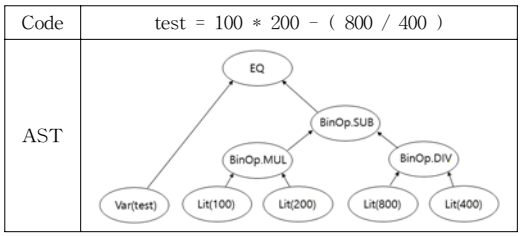

그림 7 Arith 구문 분석기의 코드와 추상구문트리
Fig. 7 Code and AST of Arith's parser

그림 7은 그림 6의 구문 분석기 명세를 이용하여 Arith 프로그램의 구문 분석 예를 보여준다.

3.3 SWLAB 파서 생성 도구의 장점

3.3.1 파서 명세 개발 환경

이 논문에서 제안한 SWLAB 파서 생성 도구를 사용하면 파서 명세와 파서 구현을 동일한 언어로 작성할 수 있다. 앞에서 예시로 설명한 바와 같이 Python 프로그램으로 파서 명세를 작성하고 실행했다. 따라서 파서 명세를 작성할 때 Python 프로그래밍을 위한 다양한 개발 환경의 지원을 받을 수 있다.

예를 들어, Python 구문 오류 검사, 자동완성, 타입 오류 검사 등의 기능을 파서 명세를 작성할 때 그대로 사용할 수 있었다. 그림 8은 파이썬 개발 환경 PyCharm에서 SWLAB 파서 생성 도구에 정의된 rule 메소드를 자동 완성하는 기능을 보여준다. 이 기능을 활용해서 파서 명세를 작성할 때 오류를 사전에 점검할 수 있었다.

그림 8 Pycharm의 코드 자동완성 기능
Fig. 8 Code assistant in Pycharm

3.3.2 새로운 프로그래밍 언어 대상 파서 생성 도구 개발

3.2절에서 SWLAB 파서 생성 도구의 Python기반 활용 예시만 설명하였지만 임의의 프로그래밍언어를 선택하더라도 이 도구를 활용할 수 있도록 쉽게 확장 가능하다. 이 주장을 보이기 위해서 이 연구에서는 Python 뿐만 아니라 Java, C++, Haskell에서도 동일한 방식으로 파서를 작성할 수 있도록 SWLAB 파서 생성 도구를 확장하였다.

표 2 파서 생성 도구 개발 시간에 관한 실험 결과
Table 2 Experimental result on the development time of parser tools

Developer	PL	Parsing knowledge level	Parser tool Development time
A	Java[1]	medium	7MD
B	Haskell[2]	high	3MD
C	Python[3]	low	10MD
D	C++[4]	low	10MD

표 2는 논문에서 제안한 방식으로 파서 생성 도구들을 만들고 그 시간을 측정한 간단한 실험 결과를 보여준다. 4명의 다른 개발자가 4가지 언어에서 SWLAB 파서 생성 도구를 개발하는데 걸린 시간을 측정하였다. 표의 3번째 컬럼은 구문 분석에 관한 지식 수준을 분류하였다. 오토마타 생성 모듈을 직접 구현할 수 있는 수준을 high, 생성 방법을 이해하고 사용하는 수준을 medium, 사용만 가능한 수준을 low로 분류한다. 각 언어별 파서 라이브러리 개발에 소용한 시간은 평균 7.5MD(Man Day)로 각 언어로 SWLAB 파서 생성 도구를 확장하는데 오랜 시간이 소요되지 않는 것을 확인할 수 있었다. 비록 실험에서 제작해본 파서 생성 도구는 LR 파서의 최소한의 기능만을 포함한 점을 고려해야하지만 새로운 프로그래밍언어로 파서 생성 도구를 만들 때 쉽게 개발할 수 있을 것으로 판단한다. 추후에 실제 수준의 파서 생성 도구를 만들면서 추가 실험이 필요하다.

4. 논의 사항

네 가지 서로 다른 프로그래밍 언어로 파서 생성 도구를 개발하면서 구문 분석 액션을 구현하는 람다 함수를 작성할 때 각 언어별 도구에서 미묘한 차이가 있음을 발견하였다. 이 절에서는 이 차이를 논의하고자 한다. 참고로 그림 9에서 동일한 생산 규칙에 대한 파서 명세를 Java, C++, Haskell로 작성한 사례를 보여준다. Python으로 작성한 사례는 그림 6에 포함되어 있다.

이 논문에서 제안한 파서 생성 도구 구조에서는 파서 명세의 생산 규칙으로 서브 구문 트리를 만들 때마다(reduce) 해당 구문 분석 액션으로 작성된 람다 함수들을 호출한다. 즉, 파서 명세의 모든 구문 분석 액션 람다 함수들의 타입은 파서 라이브러리에서 호출할 타입과 일치해야 한다. 각 람다 함수는 생산 규칙에 해당하는 트리를 각각 반환하도록 작성되기 때문에 이 트리의

```
[Java]
pu.rule("SeqExpr -> SeqExpr ; AssignExpr",
 () -> { ArrayList<Expr> seqexpr =
                        (ArrayList<Expr>)pu.get(1);
         Expr assignexpr = (Expr)pu.get(3);
         seqexpr.add(assignexpr);
         return seqexpr;
 });
[C++]
parser_util.AddTreeLambda(
 "SeqExpr -> SeqExpr ; AssignExpr",
 [this] ()->CONTAINER<AST*> {
       CONTAINER<AST*> seqexpr =
                        parser_util.GetStackInTrees(1);
       CONTAINER<AST*> assignexpr =
                        parser_util.GetStackInTrees(3);
       seqexpr.insert( seqexpr.end(),
          assignexpr.begin(), assignexpr.end() );
       return seqexpr;
   });
[Haskell]
("SeqExpr -> SeqExpr ; AssignExpr",
    \rhs -> toAstSeq (
        fromAstSeq (get rhs 1)
          ++ [fromAstExpr (get rhs 3)]) )
```

그림 9 Java, C++, Haskell 언어로 작성한 생산 규칙과 액션 함수 예제
Fig. 9 Examples of writing production rules and action functions in Java, C++ and Haskell

타입이 모두 일치해야 한다. 예를 들어 Arith 언어의 문법에서 넌터미널 PrimaryExpr로 만드는 구문 트리는 변수(identifier), 정수(integer_number), 할당식(Assign-Expr) 중 한 가지가 된다. 보통 변수, 정수, 할당식 구문 트리를 표현하기 위해 각각 다른 클래스를 정의하지만, 이 다른 클래스들을 파서 생성 도구에서 일치시키기 위해서 이 클래스들을 공통 기반 클래스(예를 들어, AST)에서 상속받도록 정의하면 된다.

이 논문에서 사용한 네 가지 프로그래밍언어는 각각 다른 타입 시스템을 갖추고 있다. Python은 동적 타이핑을 사용하기 때문에 파서 명세의 액션 람다 함수들에서 리턴하는 트리의 타입이 달라도 이 함수들을 실행할 때 런타임 오류가 발생하지 않는한 아무런 문제없다.

또한 Java의 경우도 모든 클래스는 Object 클래스를 상속받기 때문에, 파서 생성 도구에서 람다 함수의 리턴 타입을 Object라고 정의하면 여러 람다 함수들에서 서로 다른 클래스의 트리를 만들어 리턴해도 이를 Object로 일치시킬 수 있다.

하지만 C++의 경우 Object와 같은 공통 기반 클래스가 존재하지 않고, Haskell의 경우 엄격한 타입 시스템을 사용하기 때문에 이 프로그래밍 언어들로 파서 생성

1) https://github.com/kwanghoon/swlab_parser_builder
2) https://github.com/kwanghoon/genlrparser
3) https://github.com/limjintack/swlab_parser_python
4) https://github.com/tlsdorye/swlab-parser-lib

도구를 개발할 때 구문 분석 액션 람다 함수의 리턴 타입을 포괄적으로 정의해야하는 문제가 발생하였다.

C++의 경우 한 가지 쉬운 해결 방법은 파서 생성 도구에서 Java의 Object와 같은 역할을 담당할 기반 클래스를 미리 정의할 수 있다. 이 방법의 단점은 모든 파서 개발자가 자신의 명세를 작성할 때 이 특별한 기반 클래스를 상속받아 자신의 구문 트리 클래스를 정의해야 하는 점이다. 특히 구문 트리의 리스트를 만들어 리턴하는 경우 보통 C++ 라이브러리에서 제공하는 리스트 클래스를 사용하면 되는데 이 리스트 클래스조차도 이 파서 생성 도구에서 미리 정의한 특별한 기반 클래스를 상속 받도록 작성해야하는데 이것은 불편하다.

Haskell의 경우 보통 모든 구문 트리 종류들을 하나의 데이터타입의 컨스트럭터로 각각 표현한다. C++에서 논의한 바와 같이 구문 트리의 리스트를 리턴하는 액션 람다 함수의 타입을 일치시키기 위해서 이 데이터타입에 트리 리스트를 표현할 수 있는 부가적인 컨스트럭터를 추가하도록 작성해야했다. 이 역시 부자연스럽다.

파서 명세로부터 C++나 Haskell 파서 프로그램을 생성하는 기존의 파서 개발 방법을 사용한다면 이러한 문제가 발생하지 않았을 것이다. 왜냐하면 구문 분석 액션을 담당하는 람다 함수를 호출하는 포인트가 각 생산 규칙 별로 따로 존재하기 때문에 모든 람다 함수의 타입들을 하나로 일치시킬 필요가 없기 때문이다.

이 논문에서 제안하는 파서 구조를 엄격한 타입 시스템에서도 자연스럽게 사용하는 한 가지 해결책으로 의존 타입(Dependent types)을 사용하는 것을 고려해 볼 수 있다. 의존 타입은 값에 의존하여 타입을 결정하는 특징을 가지고 있다. 각 액션 람다 함수에서 리턴하는 구문 트리 종류에 따라 달라지는 타입들을 하나의 의존 타입으로 공통적으로 표현할 수 있을 것으로 기대한다.

의존 타입을 사용하면 파서 명세를 작성할 때 잘못된 트리를 반환하는 오류를 컴파일 시점에 찾을 수 있는 추가적인 장점이 있다. 즉, Object 클래스로 서로 다른 클래스를 하나의 클래스로 볼 수 있도록 일치시켰더라도 파서를 실행하면 Object 클래스를 실제 필요한 트리의 클래스로 변환하는 작업이 필요하다. 이때 파서 명세를 잘못 작성하는 등의 이유로 Object 클래스로 표현된 구문 트리이지만 원하는 종류의 구문 트리가 아니면 런타임 에러를 내고 구문 분석을 중단해야한다. Object 타입은 세부 정보를 모두 무시하고 하나의 타입으로 일치시키는 반면 의존 타입을 사용하면 세부 정보를 유지하면서 하나의 타입으로 일치시킬 수 있기 때문에 파서 명세 오류를 검출하는 기능으로서도 유용하다. 물론 이 논문의 파서 구조에서 의존 타입을 사용하기 위해서는 파서 명세로 사용하는 프로그래밍언어에서 지원해야 한다.

5. 결론 및 향후 연구

이 논문에서 LR 오토마타 생성 모듈화로 다양한 언어 환경에서 파서 생성 도구를 쉽게 구현하는 방법과 일반 프로그래밍 언어로 파서 명세를 작성하여 그 언어 환경의 장점을 모두 활용할 수 있는 방법을 제안하였다.

Python, Java, Haskell, C++의 4가지 언어로 Arith 프로그램의 파서를 작성하는 실험을 통하여 이 논문에서 제안하는 파서 생성 도구의 구조의 장점을 보였다.

향후 파서 명세를 작성할 때 표현력이 풍부한 타입을 사용하여 구문 분석 액션 람다 함수의 리턴 타입이 구문 트리의 타입과 일치해야 함으로 해서 발생할 수 있는 부자연스러운 명세 작성 부분을 개선하는 연구를 수행할 필요가 있다.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (2nd Edition), Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Rina Cohen and Karel Culik. LR-regular grammars An extension of LR(k) grammars. Proc. of the 12th Annual Symposium on Switching and Automata Theory (SWAT '71). IEEE Computer Society, Washington, DC, USA, 153-165, 1971.

[3] Aycock J., Horspool N. Directly-Executable Earley Parsing. Wilhelm R. (eds) *Compiler Construction (CC 2001)*. Lecture Notes in Computer Science, Vol. 2027. Springer, Berlin, Heidelberg. 2001.

[4] François Pottier, Yann Régis-Gianas, Towards Efficient, Typed LR Parsers, *Electronic Notes in Theoretical Computer Science*, Volume 148, Issue 2, pp. 155-180, ISSN 1571-0661, 2006.

[5] Karel Čulik, Rina Cohen, LR-regular grammars—an extension of LR(k) grammars, *Journal of Computer and System Sciences*, Vol. 7, Issue 1, pp. 66-96, ISSN 0022-0000, 1973.

[6] Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Trans, Program. Lang. Syst.* 22, 2 (March 2000), 224-264, 2000.

[7] Ralf Hinze and Ross Paterson. Derivation of a typed functional LR parser, 2002. in preparation.

[8] P. Thiemann and M. Neubauer. Parameterized LR parsing. In G. Hedin and E. van Wyk, editors, *Fourth Workshop on Language Descriptions, Tools and Applications (LDTA 2004)*, volume 110 of ENTCS, pp. 115-132, Barcelona, Spain, Elsevier Science. 2004.

[9] Neelakantan R. Krishnaswami, Jeremy Yallop. A Typed, Algebraic Approach to Parsing. PLDI 2019 *Proc. of the 40th ACM SIGPLAN Conference on Programming Language Design and Implemen-*

tation (PLDI 2019), pp. 379-393, 2019.
[10] van den Brand, M., Sellink, A., and Verhoef, C. Current Parsing Techniques in Software Renovation Considered Harmful in *Proc. of the 6th International Workshop on Program Comprehension (IWPC'98)*, Ischia, Italy, Jun. 24-26, 1998, 108-117.
[11] Friedman, D. P. & Wand, M. *Essentials of programming languages*, MIT Press, 2008.
[12] Benjamin C. Pierce, *Types and Programming Languages*, MIT Press, 2002.
[13] David Beazley, Writing Parsers and Compilers with PLY, *PyCon '07*, Feb. 03, 2007.
[14] Comparison of parser generators, https://en.wikipedia.org/wiki/Comparison_of_parser_generators

임 진 택
2018년 전남대학교 전자컴퓨터공학부 컴퓨터정보통신공학과 졸업(학사). 2018년~현재 전남대학교 전자컴퓨터공학과 석사과정. 관심분야는 프로그래밍언어, 컴파일러, 인공지능

김 가 영
2018년 전남대학교 전자컴퓨터공학부 소프트웨어공학전공(공학사). 2018년~현재 전남대학교 전자컴퓨터공학과 석사과정 관심분야는 프로그래밍언어, 컴파일러, 소프트웨어공학

신 승 현
2014년~현재 전남대학교 전자컴퓨터공학과 학사과정. 관심분야는 프로그래밍언어, 컴파일러

최 광 훈
1994년 KAIST 전산학과(공학사). 1996년 KAIST 전산학과(공학석사). 2003년 KAIST 전산학과(공학박사). 2006년~2010년 LG 전자 책임연구원. 2011년~2016년 연세대학교(원주) 컴퓨터정보통신공학부 조교수 2016년~현재 전남대학교 전자컴퓨터공학부 부교수. 관심분야는 프로그래밍언어, 컴파일러, 프로그램 분석, 소프트웨어공학

김 익 순
1994년 POSTECH 컴퓨터공학과(공학사). 1995년 KAIST 전산학과(공학석사) 2002년 KAIST 전산학과(공학박사). 2005년~2006년 프랑스 Ecole Polytechnique/ENS 방문 연구원. 2006년~현재 ETRI 통신 미디어 연구소 책임 연구원. 관심분야는 블록체인, 프로그래밍 언어, 프로그램 분석, 타입이론, 오토마타 이론