LR 오토마타 생성과 파서 도구의 분리를 통한 효율적 파서 개발 방법

임진택 김가영 신승현 최광훈
전남대학교 전자컴퓨터공학부
{wlsxor10, kirayu15, tlsdorye}@gmail.com
kwanghoon.choi@jnu.ac.kr

김익순
전자통신연구소
ik-soon.kim@etri.re.kr

A Method for Efficient Development of Parsers via Modular LR Automaton Generation

Jintaeck Lim Gayoung Kim Seunghyun Shin Kwanghoon Choi
Dept. Electronics and Computer Engineering, Chonnam National University, Gwangju

Iksoon Kim
Electronics and Telecommunications Research Institute

요 약

본 논문은 LR 파서의 효율적 구성을 위하여 두 가지 아이디어를 제안한다. 첫째, 파서 명세를 일반 프로그래밍언어로 작성하도록 구성하여 그 언어의 개발 환경에서 제공하는 구문 오류, 자동 완성, 타입 오류 등을 이용하여 사전에 오류를 바로잡을 수 있다. 둘째, 오토마타 생성을 모듈화하여 새로운 프로그래밍 언어로 파서 도구를 쉽게 확장 가능하다. 이 연구에서 제안한 아이디어로 Python, Java, C++, Haskell에서 파서를 작성할 수 있는 도구를 개발하였고, 실험을 통하여 위 두 가지 장점을 보였다.

1. 서 론

LL/LR 파싱 방법은 학부 강의를 충실히 수강한 학생이라면 구현할 수 있는 보편적인 컴파일러 기술이 되었다. 그럼에도 불구하고, 파서 작성은 복잡하고 시간이 오래 걸리기 때문에 더욱 효율적인 파서 도구를 위한 다양한 연구가 여전히 진행되고 있다.

이 논문에서는 LR 파싱 방법에 대해서 더 관심이 있다. 상향식으로 리프 노드부터 루트 노드 순서로 트리를 만드는 LR 방식은 프로그래밍언어의 문법을 대부분 수정 없이 처리할 수 있다. 그리고 LR 문법은 LL 문법을 포함한다. 하지만 기존의 LR 파서 도구는 문법을 모듈로 나누어 작성하고 이를 조합하는 방법이 부족하고, 특정 언어를 파서 구현 언어로 염두하고 개발하기 때문에 다른 구현 언어로 확장하기 어렵다.

본 논문은 앞에서 서술한 기존의 LR기반 파서 도구들의 단점을 보완하는 방법을 제안한다. 첫째, 일반 프로그래밍언어로 LR 파서 명세를 작성하도록 한다. 둘째, LR 오토마타 생성을 파서 도구와 분리하여 구성한다.

첫 번째 방법을 사용하면 일반 프로그래밍언어의 함수나 클래스로 문법을 나누어 작성하고 조합할 수 있고, 파서 명세를 작성할 때 해당 프로그래밍언어의 개발 환경을 이용하면 구문 오류나 타입 오류를 사전에 검사할 수 있다. 두 번째 방법으로 파서 도구를 구성하면 새로운 파서 도구를 만들 때 생성된 LR 오토마타를 재사용하기 때문에 파싱 트리를 만드는데 더 집중할 수 있다.

제안한 방법을 기초로 Java, Python, Haskell, C++를 파서 구현 언어로 하는 파서 도구를 개발하는 실험을 통하여 위의 두 가지 장점을 설명한다.

본 논문의 구성은 다음과 같다. 2절에서 관련 연구를 설명한다. 3절에서 SWLAB 파서 도구를 통하여 제안한 아이디어의 구현 방법을 설명하고, 4절에서 결론을 맺는다.

2. 관련 연구

[표 1]은 LL/LR 파서 도구의 대표적 사례들을 나열한다. 파서 명세 언어와 파서 구현 언어가 분리된 방식인 파서 코드를 생성하는 방식과 동일한 프로그래밍언어로 명세도 작성하고 구현하는 방식인 직접 구현하는 방식으로 구분한다.

[표 1] Examples of LR parser tools

파서 코드 생성	파서 직접 구현
YACC(C/C++) HAPPY(Haskell), CUP(Java)	PLY(Python)

이러한 파서 도구들은 문법을 나누어 작성하고 조합하는 방법을 지원하지 않는다[1]. 따라서 파서 도구 작성이 더 복잡하고 유지보수하기 어려워진다. 예를 들어 간단한 특징에서 복잡한 특징까지 추가하며 프로그래밍언어

해석기를 만드는 과정[2]이나 타입 특징을 점차 추가하며 타입 검사기를 만드는 과정[3]에서 모듈화 문법 작성 방법이 없기 때문에 동일한 문법 명세를 반복해서 파서를 작성하는 문제가 있었다.

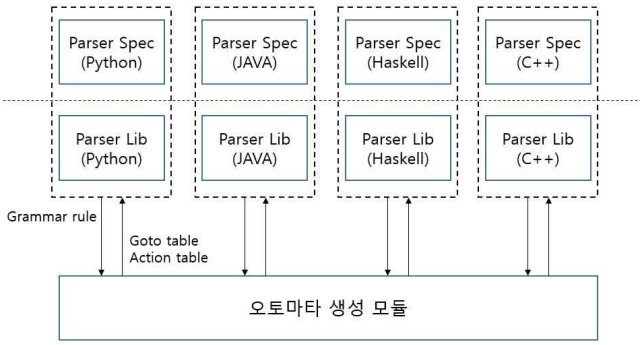
PLY[4]는 본 논문에서 지향하는 파서 도구의 구성을 가지고 있다. Python으로 파서 명세를 작성할 때 1) Python 언어 특징인 클래스와 함수를 사용하여 문법을 모듈로 나누어 작성할 수 있고, 2) 풍부한 Python 개발 환경에서 구문 오류 검사, 자동 완성, 타입 오류 검사와 같은 기능을 모두 사용할 수 있다.

하지만 PLY 도구의 방식이 가능했던 이유는 Python 언어의 Reflection과 매우 자유로운 동적 타이핑 특징 때문이다. 따라서 이 특징을 제공하지 않는 Java, C++, Haskell 언어에서 이 구현 방식을 그대로 따를 수 없다.

이 논문의 목적은 LR 오토마타 생성과 파서 도구의 분리를 통한 효율적 파서 개발 방법을 제안하는 것이다. 그래서 PLY의 경우와 달리 Python 뿐만 아니라 어떠한 프로그래밍언어, 예를 들어 Java, C++, Haskell 언어에서도 PLY 도구의 구성을 사용할 수 있는 파서 개발 도구의 구조를 설계하는 것이다.

3. SWLAB 파서 도구

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



[그림 1] SWLAB Parser tool architecture

3.1 SWLAB 파서 도구 아키텍쳐

제안하는 도구의 구조는 [그림 1]과 같다. LR 오토마타 모듈 생성은 각 언어별 파서 도구에서 공유하고, 파서 명세를 동일한 언어로 정의하며, 각 언어별 파서 라이브러리는 Action table과 Goto table을 해석하여 파싱을 진행한다. 파서 명세는 원하는 문법을 작성하여 LR 오토마타 생성 모듈에 넘긴다.

LR 오토마타 생성 모듈은 전달받은 LR 문법으로부터 LR 오토마타를 출력한다. 전통적인 LR 파서 테이블 생성 알고리즘을 사용하고 파서 라이브러리에서 사용할 수 있는 액션테이블과 Goto table을 파일로 출력한다.

파서 라이브러리는 앞서 출력한 LR 오토마타의 Action table과 Goto table과 함께 소스프로그램을 읽어 파싱을 진행한다. Action table에서 문법의 생산규칙에 따라 트리를 만들 때 마다 파서 명세의 해당 액션을 호출하여 AST를 만든다.

3.2 SWLAB 파서 도구 기반 파서 작성 예시

앞에서 설명한 아키텍처의 파서 도구를 활용하여 파서를 만드는 과정을 예시로 설명한다. 간단한 Arith 언어를 가정하고 Lexer와 Parser를 작성하는 예를 살펴보자.

Arith는 4가지 정수 연산(덧셈, 뺄셈, 곱셈, 나눗셈)과 변수를 지원하는 간단한 언어이다. 예시는 아래와 같다.

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

Lexer는 문자 리스트를 입력 받아 토큰(Token) 리스트를 출력한다. Arith 언어에 필요한 토큰은 IDENTIFER, NUMBER, EQ, MUL, SEMICOLON 등으로 미리 정의해두었다.

[그림 2]는 (Python 기반) SWLAB 파서 도구를 활용하여 작성한 Arith 언어의 Lexer를 보여준다. 앞서 정의한 Arith 언어의 각 토큰을 인식하는 정규식을 (Python 프로그램으로) 작성한 것이다. 이 정규식에 매칭되는 문자 시퀀스를 찾으면 해당하는 토큰을 만든다.

```
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 2] Arith's Lexer specification

Parser는 Lexer에서 생성한 토큰(Token) 리스트를 입력받아 AST를 출력한다. Arith 프로그램의 AST를 구성하기 위해 Expr, Assign, BinOp, Lit, Var 노드를 (Python 클래스로 작성하여) 미리 준비하였다. 이 노드들을 조합하여 Arith 프로그램의 모든 AST를 만든다.

[그림 3]은 (Python 기반) SWLAB 파서 도구를 활용하여 작성한 Arith 언어의 파서이다. Arith 언어 문법의 각 생산 규칙(Production Rule)과 해당하는 AST를 만드는 람다 함수로 구성되어 있다.

```
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)))
```

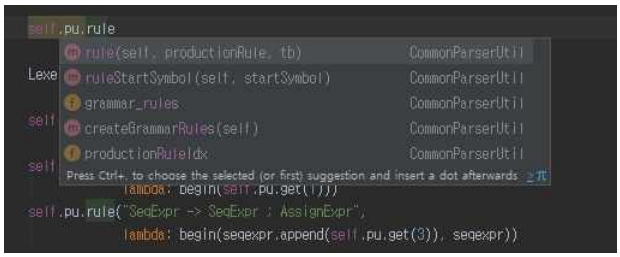
[그림 3] Arith's Parser Specification

3.3 SWLAB 파서 도구의 장점

3.3.1 파서 명세 개발 환경

이 논문에서 제안한 SWLAB 파서 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 4]는 파이썬 개발 환경 PyCharm에서 SWLAB 파서 도구에 정의된 rule 메소드를 자동 완성하는 기능을 보여준다. 이 기능을 활용해서 파서 명세를 작성할 때 오류를 사전에 점검할 수 있었다.



[그림 4] Code assistant in Pycharm

3.3.2 새로운 언어로 쉽게 확장

3.2절에서 SWLAB 파서 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 도구를 확장하였다.

[표 2]는 4명의 다른 개발자가 4가지 언어에서 SWLAB 파서 도구를 개발하는데 걸린 시간이다. 표의 3번째 컬럼에서 오토마타 생성 모듈을 직접 구현할 수 있는 수준을 (상), 생성 방법을 이해하고 사용하는 수준을 (중), 사용만 가능한 수준을 (하)로 분류한다. 각 언어별 파서 라이브러리 개발에 소용한 시간은 평균 7.5MD(Man Day)로 각 언어로 SWLAB 파서 도구를 확장하는데 오랜 시간이 소요되지 않는 것을 확인할 수 있었다. 이러한 실험 결과를 기반으로 4가지 이외의 다른 프로그래밍언어 환경으로도 쉽게 확장할 수 있을 것으로 판단한다.

[표 2] Development time of parser tool in 4 language environment

개발자	프로그래밍 언어	파싱 이론 사전 지식	파서 도구 개발 시간
A	Java[1)]	중	7MD
B	Haskell[2)]	상	3MD
C	Python[3)]	하	10MD
D	C++[4)]	하	10MD

4. 결론 및 향후연구

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

5. 참고 문헌

[1] van den Brand, M., Sellink, A., and Verhoef, C. Current Parsing Techniques in Software Renovation Considered Harmful in Proceedings of 6th International Workshop on Program Comprehension (IWPC'98) (Ischia, Italy, June 24-26, 1998), 108 - 117

[2] Friedman, D. P. & Wand, M. Essentials of programming languages MIT Press. 2008

[3] Benjamin C. Pierce. MIT Press, Types and Programming Languages, 2002

[4] David Beazley, Writing Parsers and Compilers with PLY, PyCon'07, February 03, 2007

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