논·문 18-1-?

파서생성기 생성기의 구현

Implementing a Parser-Generator Generator

김태경 · 박성우
포항공과대학교 컴퓨터공학과 프로그래밍 언어 연구실
{strikerz,gla}@postech.ac.kr

요 약

언어의 문법 표현으로부터 구문 분석기(Parser, 이하 파서)를 자동으로 생성해주는 파서생성기(Parser-Generator)에 대한 연구는 오래전부터 이루어져 왔고, 이미 다양한 프로그래밍 언어를 위한 파서생성기가 개발되어 널리 이용되고 있다. 하지만 기존의 파서생성기들은 특정한 하나의 언어에 대응되도록 개발되어왔기 때문에 컴파일러 개발자들은 새로운 언어를 설계할 때마다 그에 맞는 파서생성기를 새롭게 개발해야만 했다. 본 논문에서는 간단한 문법 번역 모듈(Translation Module) 입력을 통해 사용자가 원하는 언어에 대응되는 파서 생성기를 생성해주는 *파서생성기 생성기(Parser-Generator Generator)*의 구현에 대해 소개한다.

1 서 론

파서는 어휘분석기(lexical analyzer, 이하 렉서)에서 생성된 토큰들의 나열 속에서 문법 의미를 인식하여 주어진 문법의 의미를 AST(Abstract Syntax Tree)와 같은 구조적인 의미 형태로 변환 해주는, 컴파일러 개발에 있어서 매우 필수적인 도구이다. 하지만 문법 구조가 바뀔 때마다 새로운 파서를 개발해야하는 불편함으로 인해 파서 생성을 자동화하고자 하는 시도가 이어졌고, 결국 YACC(Yet Another Compiler-Compiler)[1]과 같은 파서생성기가 개발되는데 이르렀다.

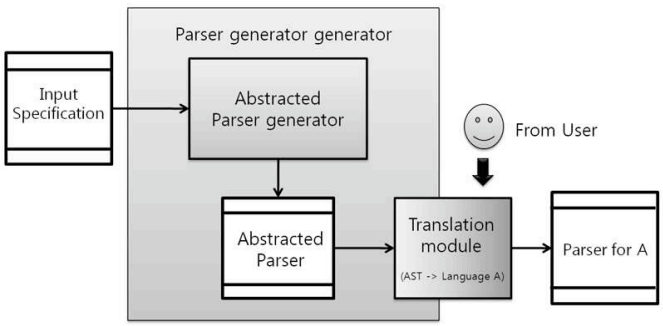
파서생성기란 사용자로부터 정의된 문법 표현을 묘사하는 입력을 바탕으로 그 문법 표현을 인식하는 파서를 자동으로 생성하는 도구를 말하며, 사용자는 자신이 원하는 문법 표현을 위한 파서를 생성하고자 할 때 이러한 파서생성기를 이용, 간단하게 파서를 만들어 낼 수가 있게 된다.

그런데 이러한 파서생성기는 모든 프로그래밍 언어마다 서로 다르게 구현될 필요가 있다. 파서생성기가 생성해내는 파서는 결국 어떤 특정한 하나의 프로그래밍 언어로 쓰여질 필요가 있는데, 이는 각각의 프로그래밍 언어에 따라 그 언어로 쓰여진 파서를 생성해주는 파서생성기가 따로 필요하다는 의미가 되기 때문이다. 예를 들어 C언어로 쓰여진 파서를 생성해내기 위해서는 C언어를 위한 파서생성기가 필요하고, Java언어로 쓰여진 파서를 생성해내기 위해서는 Java언어를 위한 파서생성기가 필요하게 되는 것이다.

물론, C와 Java와 같은 주요 프로그래밍 언어를 포함한 대부분의 프로그래밍 언어에는 이미 그 언어에 맞는 파서 생성기가 개발되어 있다. 하지만 만약 누군가가 새로운 프로그래밍 언어를 설계하였고 그 언어로 만들어진 파서를 생성해내길 원한다면 다른 언어를 위해 구현된 파서생성기는 무용지물이 되고 만다. 결국 새로운 언어를 위한 파서 생성기를 그 언어에 맞도록 새롭게 구현해주어야만 하는데, 완전 새롭게 자신만의 파서생성기를 개발해내거나 이미 구현되어 있는 파서생성기의 소스 코드를 분석, 참고하여 자신만의 파서

생성기를 만들어 내는 일은 물론 불가능한 일은 아니지만, 많은 시간과 노력을 필요로 한다. 본 연구는 이러한 노력을 최소화하기 위해 파서 생성기를 이용해 파서를 생성하듯 파서생성기 자체를 생성해주는 파서생성기 생성기 개발을 목표로 한다.

이를 위해, 기존 파서생성기가 생성해내는 파서의 구조를 일반화(Generalization) 하고 추상화(Abstraction) 하는 시도를 통하여, 특정 언어로 쓰여진 파서를 출력하는 것이 아닌 일반적인 프로그래밍 언어 요소를 바탕으로 하는 추상화된 형태의 파서(Abstracted Parser)를 1차적으로 출력하도록 한다. 그리고 이러한 추상화된 형태의 파서를 사용자가 원하는 언어로 변환해주는 변역 모듈(Translation Module)을 적용해 최종적으로 사용자가 정의한 언어로 쓰여진 파서가 출력되도록 하는 것이다(그림 1).



[그림 1] 파서생성기 생성기 (parser-generator generator)

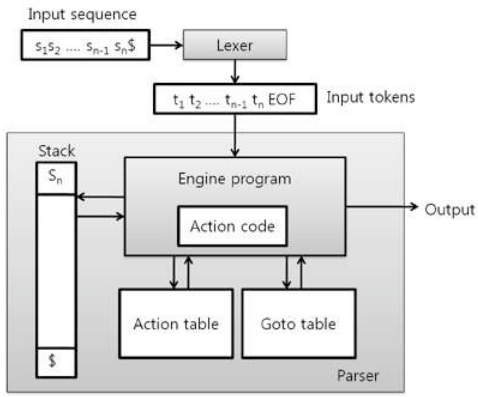
이때 번역 모듈은 추상화된 파서를 묘사하는데 사용된 일반적인 프로그래밍 언어요소를 사용자가 원하는 프로그래밍 언어로 변환해주는 역할을 하는 것으로, 파서생성기 생성기의 입력이 된다. 결국 파서생성기 생성기는 추상화된 파서를 출력해주는 일을 핵심으로 한다고 할 수 있으며, 사용자로부터 변환 모듈을 입력받아 사용자가 원하는 언어로 쓰여진 파서를 생성하는 파서생성기를 출력으로 하는 형태의 프로그램이라고 말할 수 있다.

여기서 중요한 것이 파서를 구성하는데 일반적인 프로그래밍 언어 요소가 무엇이냐 하는 것인데, 우선 생성되는 파서의 형태를 최대한 간단한 구조로 바꾸어 그것을 표현하는데 필요한 언어요소를 최소화하는 것이 필요하다. 예를 들어 if then else 구문만을 이용해 파서생성기에서 생성되는 파서의 구조를 표현할 수 있다면 if then else와 같은 언어 요소를 포함하는 모든 프로그래밍 언어에 대응되는 파서생성기를 만들어 낼 수 있게 되겠지만, 기타 복잡하고 다양한 언어 요소를 필요로 할 경우 파서생성기 생성기가 만들어 낼 수 있는 파서 생성기의 수는 제한될 것이기 때문이다. 본 연구에서는 파서의 구조를 일반화 하고 추상화 하는 것은 물론 이에 사용되는 언어 요소의 최소화를 통해 가능한 일반적인 언어를 대상으로 하는 파서생성기 생성기 개발에 대해 논의한다. 이때 기본이 되는 프로그래밍 언어요소는 함수형 언어를 바탕으로 한다.

본 논문은 다음과 같이 구성된다. 2장에서 우리는 기존에 구현된 파서생성기의 기능과 구조를 살펴보고 그것을 우리가 개발할 파서생성기 생성기에 어떻게 이용할 수 있는지 살펴본다. 3장에서는 출력될 파서를 구성하는 방법과 여기에 필요한 프로그래밍 언어 요소가 무엇이 있는지 살펴본다. 4장에서는 예제를 바탕으로 실제 파서생성기 생성기가 어떤 방식으로 동작하는지 알아보며, 5장에서는 현재 파서생성기 생성기의 한계와 확장 가능성에 대해 논의하며 결론을 맺는다.

2 기존 파서생성기의 분석 및 이용

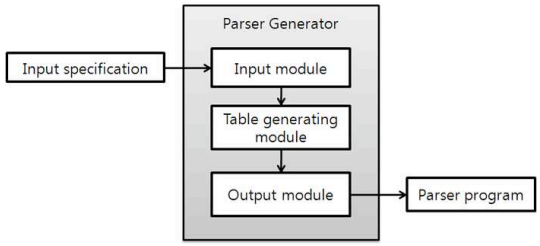
일반적으로, 파서[1]는 스택(Stack)과 액션코드(Action Code)등을 포함하는 파서 엔진(Parser Engine)과 파서 테이블(Parser Table)로 이루어진다(그림 2).



[그림 2] 파서(parser) 의 구조

여기서 파서 테이블은 다시 ACTION 테이블과 GOTO 테이블 두 가지로 나뉘며, 이 두 가지 테이블을 바탕으로 파서 엔진은 입력된 토큰(Token)의 나열 속에서 사용자가 정의한 문법 의미를 인식하게 되고, 사용자로부터 정의된 액션 코드(Action Code)를 이용해 인식한 문법에 해당하는 행동(Action)을 취하게 된다. 여기서 주목해야할 부분은, 모든 문법 구조에 대해 파서 엔진 부분은 동일하고 두 가지 테이블 부분만 변화하게 된다는 사실이다. 이제 이러한 테이블을 만들어내기 위한 파서생성기의 개략적인 구조에 대해 알아보자.

일반적인 파서생성기의 구조는 다음(그림 3) 과 같다.



[그림 3] 파서생성기(parser generator)의 구조

파서생성기는 그림과 같이 크게 입력 모듈(Input Module), 테이블 생성 모듈(Table Generating Module) 그리고 출력 모듈(Output Module)로 나뉠 수 있는데, 입력 모듈은 사용자에 의해 정의된 문법 표현을 나타내는 입력 명세(Input Specification)를 분석하여 그것을 테이블 생성 모듈에서 사용할 수 있도록 데이터를 구조화하는 역할을 한다. 테이블 생성 모듈에서는 말 그대로 파싱(parsing)에 필요한 두 가지 테이블인 ACTION 테이블과, GOTO

[1]본 논문에서는 Lookahead LR(LALR) 방법을 사용하는 파서만을 다루도록 한다.

테이블을 만들어내어 출력 모듈로 넘겨준다. 출력 모듈에서는 테이블 생성 모듈에서 받아온 테이블 및 기타 정의된 파서 엔진 부분 등을 더하여 파서를 출력해낸다.

파서생성기 생성기 역시 입력 명세로부터 테이블을 생성하고, 이를 바탕으로 추상화된 파서를 출력하는 기능을 수행하므로 위에 언급된 파서생성기와 비슷한 구조를 가질 것이라고 생각할 수가 있다. 실제로 테이블을 구성하는 부분까지는 기존의 파서생성기와 완전히 같은 일을 수행한다고 할 수 있는데, 여기서 만약 우리가 파서 생성기에서 사용하던 입력 명세 형식을 수정 없이 사용할 수 있다면 우리는 기존 파서생성기에 구현되어 있는 입력 모듈과 테이블 생성 모듈을 그대로 이용할 수가 있다. 실제로 사용자가 정의한 문법 표현을 나타내는 입력 명세의 형식은 최종적으로 출력될 파서에 사용되는 프로그래밍 언어와 관련이 없기 때문에, 기존의 파서생성기에서 사용되는 입력 명세를 그대로 사용해도 문제가 없다. 만약 추상화된 파서를 구성하는데 추가적인 정보가 필요하다 하더라도 기존의 입력 명세 형식을 크게 수정하지 않는다면 입력 모듈과 테이블 생성 모듈은 새롭게 구현할 필요가 없이 약간의 수정을 통해 사용이 가능하다.

결국 우리는 기존에 구현된 파서생성기의 입력 모듈과 테이블 생성 모듈을 최대한 이용하고, 우리가 원하는 형태의 파서를 출력하도록 출력 모듈만 새롭게 구현한다면 파서생성기 생성기를 더욱 효율적으로 개발할 수 있다는 결론에 도달하게 된다. 이를 위하여 소스 코드가 공개된 기존의 파서생성기중 하나를 골라 구조를 분석하여 그것이 제공하는 각종 정보에 접근하는 방법을 공부할 필요가 있는데, 본 논문에서는 대표적인 YACC의 변형 중 하나인 Berkeley YACC(BYACC)[2]의 소스 코드를 이용한다.

BYACC의 소스 코드를 분석해보면 입력 명세로부터 얻어낸 각종 정보 및 테이블 생성 모듈에서 생성된 ACTION 테이블과 GOTO 테이블이 저장되어 있는 자료구조에 접근하는 방법을 알 수 있다 (예를 들어 ACTION 테이블은 `struct action`이라는 자료 구조의 배열인 `parser[]`를 통해 접근가능하고, GOTO 테이블은 `struct shift`라는 자료구조의 배열인 `shift_table[]`을 통해서 접근이 가능하다). 이제 남은 일은 위의 두 테이블의 정보를 이용해 스택 및 파서 엔진을 포함한 파서를 구성하는 것이다. 이제 주어진 정보를 바탕으로 파서를 어떻게 구성하며, 그에 필요한 프로그래밍 언어 요소가 무엇이 있는지 살펴보도록 하자.

3 파서의 구성 및 필수적 프로그래밍 언어 요소

우리는 파서생성기 생성기가 출력하는 파서를 구성하는데 있어서 구현의 효율성과 기능의 다양성보다는, 가장 일반적인 프로그래밍 언어요소를 최대한 적게 사용하여 핵심적인 기능을 구현함을 최우선 목표로 한다. 이는 프로그래밍 언어 요소를 적게 사용할수록, 파서생성기 생성기가 지원하는 언어의 수가 늘어나기 때문이다. 여기서 파서의 핵심적인 기능은, 토큰의 나열 속에서 문법 의미를 인식하여 사용자가 작성한 액션 코드를 실행하는 파서의 가장 기본적인 기능에 더하여 현재 파서가 읽어 들인 토큰의 위치(location) 정보를 제공하는 기능을 포함한다.

이를 위해 우리가 사용할 프로그래밍 언어 요소는 함수형 프로그래밍 언어를 바탕으로 한다. 물론, 대다수의 프로그래밍 언어가 포함하고 있는 요소만을 이용하여 파서를 구성하는 것이 가장 바람직하지만, 최근의 프로그래밍 언어는 매우 다양한 방향으로 발전하고 있고, 모든 프로그래밍 언어의 방식을 공통으로 아우르는 가장 일반적인 요소를 찾아내기란 매우 힘든 일이다. 본 연구에서는 우선적으로 함수형 언어의 요소를 바탕으로 하여 파서를 구성하되, 이에 사용되는 언어 요소를 가능한 최소화하여 파서생성기 생성기에서 생성된 파서를 함수형 언어와 다른 형식의 언어에서도 가능한 사용할 수 있도록 하는 것을 목표로 한다.

이제 BYACC의 파서 테이블 생성 모듈에서 생성된 테이블 정보를 바탕으로 파서를 어떤 형식으로 구성할 것인지에 대해 논의하고, 이를 위해서는 어떤 프로그래밍 언어 요소가 필요한지 알아보도록 한다. 이를 위해 OCaml(Objective Caml)[3]의 문법을 바탕 으로하는 의사 코드(pseudo-code)를 사용한다.

3.1 파서의 구성

2절에서 언급했듯, 파서는 크게 파서 엔진과 파서 테이블로 이루어진다. 이를 다시 세분화하면, 파서 엔진은 파싱 엔진(Parsing Engine), 파싱 스택(Parsing Stack) 그리고 액션코드 적용 (Applying action code) 부분으로 나눌 수 있으며 테이블은 ACTION 테이블과 GOTO 테이블로 나눌 수 있다. 이제 각각을 대략적으로 구성해 보면서 이를 위해 최소한으로 필요한 프로그래밍 언어 요소에 대해 알아보도록 하자.

3.1.1 *파서 테이블.* 다음은 파서 테이블이 어떻게 구성되고 사용되는지 알아보기 위한 간단한 문법 표현 예제이다.

1. $LIST \rightarrow LIST\ ;\ ELEMENT$
2. $LIST \rightarrow ELEMENT$
3. $ELEMENT \rightarrow a$

위의 문법 표현은 a, a;a, a;a;a 등과 같은 문자열을 인식한다. 이러한 문법 표현을 바탕으로 구성한 ACTION 테이블과 GOTO 테이블은 그림 4 와 같다.

Symbol / State	Action Table			Goto Table	
	a	;	$	LIST	ELEMENT
0	Shift 3			1	2
1		Shift 4	Accept		
2		Reduce 2	Reduce 2		
3		Reduce 3	Reduce 3		
4	Shift 3				5
5		Reduce 1	Reduce 1		

[그림 4] ACTION 테이블과 GOTO 테이블

테이블의 형태를 보면 알 수 있듯 두 테이블 모두 현재 파서의 상태와 주어진 심볼(symbol)의 종류를 바탕으로 접근이 가능하다. 그러므로 우리는 함수의 선언과 이용, 패턴 매칭 (pattern-matching)과 같은 기본적인 프로그래밍 언어 요소를 바탕으로, 다음과 같은 타입과 형태를 가지는 함수로 두 가지 테이블을 각각 나타낼 수 있다.

```
(* action_table : state -> symbol -> action *)
let action_table state symbol =
  match (state, symbol) with
  | (0, CHAR_A) -> Shift 3
  | (1, SEMI) -> Shift 4
  | (1, EOF) -> Accept
...
```

```
(* goto_table : state -> symbol -> state *)
let goto_table state symbol =
  match (state, symbol) with
  | (0, LIST) -> 1
  | (0, ELEMENT) -> 2
...
```

3.1.2 *파싱 엔진 (parsing engine).* 파싱 엔진은 파서의 가장 핵심적인 부분으로서, 렉서로 부터 토큰을 읽어오고 파서 테이블과 파싱 스택을 이용하여 문법에 알맞은 액션 코드를 실행해주는 역할을 한다. 다음은 개략적인 엔진 프로그램의 모습이다.

```
let engine lexfun lexbuf init_state =
  let rec engine' token' loc' lexbuf' state' stack' =       (* line 3 *)
    match (action_table state token) with
    | Shift next_state ->
       push(current_state, token);
      ...
    | Reduce rule ->
       apply_action(rule, state, stack);
      ...
    | Error ->
      ...
    let (token, loc, lexbuf') = lexfun lexbuf              (* line 12 *)
    in engine' token loc lexbuf' init_state empty_stack    (* line 13 *)
```

우선 엔진 프로그램이 처음 호출되면 렉서로부터 토큰(token)과 그 위치정보(loc), 갱신된 버퍼(lexbuf')를 얻어내고(12번째 줄), 얻어낸 정보와 함께 초기 상태정보, 비어있는 스택을 재귀 함수인 engine'에게 넘겨준다(13번째 줄). engine'는 주어진 상태정보와 토큰을 이용해 ACTION 테이블을 참조해 Shift, Reduce등 수행해야할 액션을 결정하는데(3번째 줄), 이때 결정된 액션에 따라 현재 상태와 토큰정보를 push 하거나 액션코드를 적용(apply)하는 등의 작업을 수행한다. 그 후 업데이트된 정보를 바탕으로 자기 자신을 호출해나가면서 계속적으로 파싱을 수행하다가, 에러가 발생하거나 Accept가 되었을 경우 프로그램을 종료한다.

위의 프로그램은 재귀 함수, let-binding 과 같은 프로그래밍 언어 요소를 이용해 작성이 가능하다. 또한 렉서를 이용하는데 필요한 특정한 타입을 가지는 lexfun이라는 함수를 인자로 받아올 필요가 있으므로 이를 지원하는 프로그래밍 언어 요소 역시 필요하다.

3.1.3 *파싱 스택 (parsing stack).* 파싱 스택은 상태 번호(state number)와 심볼(symbol) 정보를 동시에 보유하고 있어야 한다. 그러므로 우선 상태번호와 심볼 정보를 포함하는 스택 원소(stack element) 타입을 선언해야만 하는데, 다음은 그 예제이다.

```
type stack_elem =
  | Stack_0_
  | Stack_element of state * location * char
  | Stack_list of state * location * char list
  | Stack_SEMI of state * location
  | Stack_CHAR of state * location * char
  | Stack_EOF of state * location
```

Stack_0_은 스택의 바닥(bottom)을 나타내기 위한 것이며, 나머지는 각각 Stack_ 뒤에 각 심볼들의 이름을 붙여둔다. 그리고 각각의 스택 원소는 그 심볼에 해당하는 상태 번호 (state)

및 그 심볼의 위치 정보 (location)를 가지고 있게 된다. 여기에 추가적으로 해당 심볼의 의미 값(semantic value)을 가지고 있을 필요가 있는데, 위의 예제에서 보면 SEMI와 EOF는 의미 값을 가지지 않는 심볼이고, CHAR를 비롯한 element, list등은 각각의 타입(char, char list등)으로 나타나는 의미 값을 가지는 심볼이다.

위와 같이 스택 원소 타입을 생성해내기 위해서는 모든 심볼의 이름과 그 심볼들이 가지는 의미 값의 타입을 미리 알 수 있어야 한다. 심볼의 이름은 BYACC에서 제공하는 정보를 이용해 접근이 가능하지만, 각각의 심볼이 가지는 의미 값은 기존의 BYACC에서 제공하는 정보만으로는 미리 알아낼 방법이 없다. 그러므로 우리는 각 심볼들의 의미 값의 타입을 입력 명세에 추가적으로 명시하도록 할 필요가 있다.

위의 조건이 충족되면 스택 원소의 정의가 가능해진다. 이제 이러한 스택 원소를 스택에 넣고 빼내는 함수인 push와 pop 함수가 필요하다. 스택을 구현하는 방법은 매우 다양하고, 사용자가 사용하는 언어에 종속되어 있는 경우가 많기 때문에, 이 두 가지 함수는 파서생성기 생성기에서 직접 만들어내기 보다는 사용자로부터 직접 입력을 받아 적용하는 편이 유리하다. 파서생성기 생성기는 두 함수의 타입을 다음과 같이 정의한다.

```
push : stack_elem -> stack -> stack
pop : stack -> stack_elem * stack
```

push는 스택에 집어넣을 스택 원소와 함께 기존의 스택을 인자로 받아 스택 원소를 스택에 넣은 후 갱신된 스택을 되돌려준다. pop은 스택을 인자로 받아 가장 위에 있는 스택 원소와 그 스택 원소가 제거된 갱신된 스택을 함께 되돌려준다.

결국, 파싱 스택을 구성하기 위해 우리는 기존 BYACC의 입력 명세에서 각 심볼의 의미 값의 타입을 명시하도록 하는 부분을 추가할 필요가 있으며 스택의 push와 pop의 구현을 사용자 입력으로 받아들이도록 해야 한다. 이를 위해 타입을 선언하는 프로그래밍 언어 요소가 추가적으로 필요하다.

3.1.4 *액션 코드 적용 (applying action code).* 액션 코드 적용 부분에서는 규칙 번호(rule number)와 현재 상태 번호 그리고 스택을 인자로 입력 받아, 해당 규칙(rule)에 맞게 스택에서 심볼들을 꺼내 소비하고 액션코드를 수행하며 그 결과를 다시 스택에 저장해 갱신된 스택을 돌려주는 역할을 한다. 다음은 3.1.1절에서 소개된 문법 규칙 예제에 해당하는 액션 코드 적용 부분 일부이다.

```
let apply_action rule state stack =
  match rule with
  | 1 ->
    pop (elem1, elem2, elem3) from stack                    (* line 4 *)
    match (elem1, elem2, elem3) with
    | Stack_ELEMENT(_,L3,C3), Stack_SEMI(_ ,L2), Stack_LIST(s,L1,C1) ->
      let code = executing action code
      in (goto_table LIST s, push Stack_LIST (s, L1, code) stack)
    | _ -> (error_state, ...))                              (* line 9 *)
  | 2 ->
    ...
```

위의 코드는 첫 번째 규칙인 $LIST \rightarrow LIST$; $ELEMENT$ 를 묘사하고 있다. 규칙에 따라 세 개의 심볼을 스택으로 부터 꺼내고 비교한 후 (4 ∼ 6번째 줄),액션 코드를 실행하고 (7번째 줄), 결과로 나오는 심볼을 다시 스택에 넣고 있는 것을 볼 수 있다 (8번째 줄). 이때 각각의 심볼이 가지는 위치정보와 의미 값에 접근하기 위해 L1, L2나 C1, C3과 같은 변수를 도입해 사용자가 정의한 액션 코드에서 해당 심볼의 위치정보나 의미 값에 접근할 수 있도록 한다. 이에 관한 사항은 4절에서 더 자세히 다루도록 한다.

9번째 줄은 에러 처리를 위한 코드이다. 여기서 에러를 예외처리 (exception handling)등의 언어 요소를 이용하여 처리할 수도 있지만, 사용되는 문법 요소를 최소화하기 위해 위와 같이 특정한 `error_state`를 도입하여 에러를 처리한다.

3.2 필수적 프로그래밍 언어 요소

파서생성기 생성기가 필요로 하는 필수적 프로그래밍 언어 요소는 아래와 같이 크게 네 가지 요소로 나눌 수 있다.

- **Declaration** : type declaration, function declaration
- **Type expression** : identifier, tuple type
- **Pattern** : identifier, tuple pattern, constructor, constant
- **Expression** : identifier, tuple expression, constructor, constant, application, let-binding, pattern-matching

선언(Declaration) 부분은 타입 선언과, 함수 선언으로 나눌 수 있다. 타입 선언을 통해 새로운 타입 생성자(type constructor)를 도입하여 새로운 타입을 정의하거나 기존의 타입을 재명명(aliasing)할 수 있으며 함수 선언을 통해 재귀 함수(recursive function) 선언이 가능하여야 한다. 타입 표현(Type expression)은 특정한 하나의 타입을 나타내는 식별자(identifier)와 여러 개의 타입을 나타내는 튜플(tuple) 두 가지로 구성된다. 패턴(Pattern)은 식별자와 튜플외에 생성자(constructor), 상수(constant)로 구성된다. 표현(Expression)에는 함수와 그 인자의 적용 (application)을 위한 표현과 지역 변수 및 함수를 선언하기 위한 let-binding 및 패턴 비교를 위한 pattern-matching과 같은 표현이 포함된다. 여기서 함수는 그 자체를 인자로 주고받을 수 있어야 한다.

이제 이러한 언어 요소들을 3.1절에서 구성한 파서를 묘사하는데 사용할 수 있도록 추상화하기 위해, 위의 언어 요소들을 포함하는 AST(Abstract Syntax Tree)를 도입한다.

```
Declaration ::= Dtype (string, (string, Type) list)                          (* type declaration *)
              | Dtype_alias (string, Type)                                   (* type alias *)
              | Dfunc (rec_flag, string, string list, Expression)            (* function declaration *)

Type ::= Tident (string)                                                     (* identifier *)
       | Ttuple (Type list)                                                  (* tuple *)

Pattern ::= Pident (string)                                                  (* identifier *)
          | Pconstant (int)                                                  (* constant *)
          | Ptuple (Pattern list)                                            (* tuple *)
          | Pconstructor (string, Pattern)                                   (* constructor *)
          | Pwildcard                                                        (* wildcard *)

Expression ::= Eident (string)                                               (* identifier *)
             | Econstant (int)                                               (* constant *)
             | Etuple (Expression list)                                      (* tuple *)
             | Econstructor (string, Expression)                             (* constructor *)
             | Eapply (Expression, Expression list)                          (* apply *)
             | Elet (Pattern, Expression, Expression)                        (* let binding *)
             | Ematch (Expression, (Pattern, Expression) list)               (* pattern matching *)
```

[그림 5] 파서생성기 생성기에 필요한 필수 언어 요소로 이루어진 AST의 정의

그림 5에 정의된 AST를 이용하면 3.1절에서 구성하였던 파서에 필요한 모든 부분을 작성 할 수 있다. 그러므로 이제 사용자로부터 여기에 정의된 AST를 사용자가 원하는 언어로 번역해주는 함수로 이루어진 번역 모듈을 입력 받아 여기에 적용하기만 하면 사용자가 원하는 언어로 구성된 파서를 얻을 수 있게 되는 것이다.

4 파서생성기 생성기의 이용

이 절에서 우리는 파서생성기 생성기를 이용해 실제 OCaml에 대응되는 파서생성기를 만들어 볼 것이다. 이를 위해 우선 그림 5에 정의된 AST를 OCaml 언어의 문법으로 번역하는 변역 모듈을 구현하고, 파서생성기 생성기에 제공되어야할 스택 관련 함수 및 위치 정보와 관련된 사항들을 정의한다. 이러한 과정을 통해서 우리는 OCaml을 위한 파서생성기를 만들어 낼 수 있는데, 여기서 만들어진 파서생성기가 입력으로 받아들이는 입력 명세 형식이 기존 BYACC의 입력 명세 형식과 어떻게 다른지 알아보도록 하고, 이를 이용해 생성해낸 파서와 토큰 생성에 필요한 렉서를 연결하는 방법 역시 알아보도록 한다.

4.1 번역 모듈의 작성

번역 모듈은 파서생성기 생성기에서 1차적으로 생성된 추상화된 파서를 사용자가 원하는 언어로 번역해주는 역할을 한다. 그림 5에 정의된 AST를 이용해 만들어진 트리(tree) 구조를 순회(traverse)하며 각각의 추상화된 언어요소를 사용자가 원하는 언어요소로 번역 및 출력하게 되는데, 이때 트리구조를 순회하는데 필요한 코드등은 모두 파서생성기 생성기에서 제공하므로, 사용자는 단순히 제공된 함수의 뼈대(skeleton)를 바탕으로 번역 함수 의 몸통(function body) 부분만을 구현하기만 하면 된다. 예를 들어 그림 5에 정의된 AST에서 Pwildcard는 OCaml에서 _ 로 번역이 되는데, 이는 다음과 같이 간단히 구현이 가능하다.

```
void prt_pattern_WILDCARD ()
{
   fprintf(output_file,"_");
}
```

함수의 뼈대는 미리 제공되므로, 사용자는 `fprintf(output_file,"_");` 부분만 구현하면 된다. 여기서 `output_file`은 출력으로 나올 파서 프로그램이 저장될 파일의 포인터이다.

이번엔 조금 복잡한 표현인 pattern-matching을 번역해보도록하자. Ocaml에서 pattern-matching 표현은 다음과 같은 형식을 가진다.

```
match exp with
| pat1 -> exp1
| pat2 -> exp2
..
```

이러한 형식을 나타내기 위해 파서생성기 생성기는 `e_match`라는 구조체(struct)를 도입하고, 번역 함수는 이 구조체의 정보를 다음과 같이 이용해 번역한다. 물론 `e_match`의 멤버 변수들에 대한 상세한 설명은 주석 등으로 제공된다.

```
void prt_expression_MATCH (struct e_match *e)
{
    int i;
    fprintf(output_file,"match ");
    prt_expression(e->expr);
    fprintf(output_file," with");
    for (i=0; i < e->num_pelist; i++)
    {
        fprintf(output_file,"\n| ");
        prt_pattern(e->pelist[i]->pat);
        fprintf(output_file," -> ");
        prt_expression(e->pelist[i]->expr);
    }
}
```

위의 구현 예와 같이 사용자는 파서생성기 생성기에서 제공하는 뼈대를 이용하여 간단히 번역 함수를 구현 할 수가 있다. 이와 같은 방법으로 파서생성기 생성기가 요구하는 모든 번역 함수를 작성한다.

다음으로 필요한 것은 스택 관련 함수 및 위치 정보와 관련된 사항을 정의 및 구현하는 것이다. 다음은 Ocaml 언어에 맞도록 이들을 구현한 예이다.

```
char *define_location_type = "int";
char *define_stack_type = "stack_elem list";
char *define_stack_empty = "[]";
char *define_stack_pop = "
  match st with
    | [] -> (Stack_0_,empty_stack)
    | head::st' -> (head,st')";
char *define_stack_push = "elem::st";
```

첫 번째 줄의 define_location_type는 렉서가 넘겨주는 토큰의 위치 정보의 타입이다. 위의 예제에서는 렉서에서 제공 받는 위치 정보가 단순히 토큰의 시작지점만을 나타내는 것을 가정하였으므로 이를 int로 정의하였으나, 렉서가 더 자세한 위치 정보를 제공해준다면 그에 해당하는 위치 정보의 타입을 직접 정의해주면 된다. 두 번째 줄부터는 파싱 스택의 구현에 관한 부분인데 위 예제에서는 스택을 간단히 리스트(list)로 구현하였다. define_stack_empty에는 비어있는 스택에 해당하는 코드를 입력하면 되고, define_stack_pop과 define_stack_push에는 각각 스택의 pop과 push에 해당하는 함수의 몸통 부분의 코드를 입력하면 된다.

4.2 입력 명세(input specification)의 형식

파서생성기 생성기에서 만들어진 파서생성기의 입력 명세 형식은 BYACC의 입력 명세 형식을 바탕으로 하고 있다. BYACC의 입력 명세 형식은 이미 잘 알려져 있으므로 자세한 설명은 생략하도록 하고 ([4] 참조), 여기서는 BYACC의 입력 명세와 우리가 만들어낸 파서생성기의 입력 명세간의 차이점만을 설명하도록 한다. 차이점은 심볼들의 의미 값(semantic value)의 타입 선언 여부와 위치 정보에 대한 접근 방법 두 가지로 설명할 수 있는데, 이를 알아보기 위해 우선 3.1.1절에서 소개한 문법 표현을 나타내는 BYACC 입력 명세를 예제로 소개한다.

```
%{
    /* header section*/
%}
%token SEMI                                              (* line 4 *)
%token <char> CHAR_A
%token EOF
%start <char list> start_list
%%
/*rules section */
start_list  : list EOF { $1 }
list        :  list SEMI element {$3 :: $1}              (* line 11 *)
            |  element          { [$1] }
;
element     :  CHAR_A { $1 }
;
%%
/* trailer section*/
```

위의 입력 명세에서는 토큰에 해당하는 최종 심볼(terminal-symbol) 및 시작 심볼의 의미 값의 타입만을 선언해주고 있다(4∼7번째 줄). 하지만 우리가 생성해낸 파서생성기는 모든 심볼의 의미 값의 타입 정보를 알고 있어야하므로(3.1.3절 참조) 중간 심볼(nonterminal-symbol)의 의미 값의 타입정보를 선언해주는 부분을 다음과 같이 추가할 필요가 있다.

```
%type<char list> list
%type<char> element
```

또한 우리는 액션 코드에서 각 토큰의 위치 정보에 접근하는 방법으로 `@`라는 기호를 도입한다. 이는 Byacc에서 의미 값 접근을 위해 사용하는 기호인 `$`와 비슷한 방식으로 사용된다. 예를 들어 각 문법 규칙에 정의된 첫 번째, 두 번째 심볼의 의미 값에 각각 접근하기 위해 `$1`, `$2`와 같은 명령어를 사용하는 것과 비슷하게 해당 위치 정보에 접근하기 위해 우리는 `@1`, `@2`와 같은 명령어를 사용한다. 실제로 위의 입력 명세 11번째 줄의 액션 코드 부분에서 두 번째 심볼인 `SEMI`의 위치 정보에 접근하고자 할 때 우리는 간단히 `@2`와 같은 기호를 이용할 수 있다.

4.3 렉서(lexer)와의 연결 방법

4.1절에서 만들어진 파서생성기가 생성해낸 파서는 토큰의 생성을 위해 `lexfun`이라는 렉싱(lexing) 함수와 `lexbuf`라는 입력 흐름(input stream)을 저장하는 버퍼(buffer)를 필요로 한다. 여기서 `lexfun`은 `lexbuf -> (token * location * lexbuf)`와 같은 타입을 가져야만 하는데, `token`은 렉서가 출력할 토큰의 타입을, `location`은 4.1장에서 정의한 위치 정보 타입을 의미하며 출력으로 나오는 `lexbuf`는 출력된 토큰이 제거된 갱신된 버퍼를 나타낸다.

예제로서 OCaml에서 제공하는 렉서 생성기인 ocamllex[5]가 만들어내는 렉서와 4.1절에서 만들었던 Ocaml을 위한 파서생성기가 생성해내는 파서와의 연결방법을 소개한다. OCaml로 작성된 다음 코드는 파서가 요구하는 타입에 맞게 `lexfun`과 `lexbuf`를 만들어 파서에게 그것들을 넘겨주는 모습을 보여준다.

```
let program input =
  let lexfun lb =
    let token = Lexer.token lb
  in (token,lb.Lexing.lex_curr_pos,lb) in
  let lexbuf = Lexing.from_channel input in
  let result = Parser.start lexfun lexbuf
in result
```

위 코드에서 우리는 `lexfun`을 `lexbuf -> (token * location * lexbuf)` 타입에 맞게 만들고 (2~4번째줄) , `lexbuf`를 Ocaml에서 지원하는 라이브러리를 이용해 만들어 낸 후 (5번째 줄), 이를 파서를 시작하기 위해 호출해야하는 함수인 `Parser.start`에 인자로 넘겨준다. (6번째 줄). 마지막으로 파싱 결과는 result라는 변수에 저장된다.

5 결 론

우리는 이 논문에서 기존에 구현되어있는 파서생성기인 Byacc을 바탕으로, 출력되는 파서의 구조를 일반화하고 추상화한 후 사용자로부터 번역 모듈을 입력 받는 방법을 통해 파서생성기 생성기를 구현하는 방법을 살펴보았다. 또한 구현된 파서생성기 생성기를 실제로 어떻게 사용하는지 예제를 통해 알아보았다. 이를 통해 프로그래밍 언어 개발자는 파서 생성을 위한 복잡한 알고리즘을 구현할 필요가 없이 간단한 번역 모듈 구현을 통해 원하는 언어에 맞는 파서생성기를 간편하게 만들어 낼 수 있게 되었다.

하지만 파서생성기 생성기는 분명한 한계도 가지고 있다. 우선 사용되는 프로그래밍 언어 요소가 함수형 언어를 바탕으로 하고 있으므로, 함수형 언어를 위한 파서생성기를 만들때는 간편하게 구현이 가능하나, 다른 형태의 언어를 위한 파서생성기를 만들고자하는 경우에는 적합하지 않을 수 있다. 사용된 언어 요소의 수가 무척 적기는 하지만 함수형 프로그래밍 언어 요소를 해당언어에서 사용할 수 있으려면 복잡한 과정을 통해 변환하는 과정을 거쳐야할 때가 많기 때문이다. 또한 프로그램 구조 간결화를 위해 핵심적인 기능만을 포함하였으므로 기존의 파서생성기에서 제공하는 다양한 기능을 모두 활용하지 못한다. 그리고 렉서와의 연결시 파서는 고정된 형식의 렉싱 함수와 버퍼를 요구하는데, 만약 렉서가 이러한 형식을 지원하지 않는다면 해당 렉서는 사용할 수 없다는 문제점도 있다.

우리는 이러한 문제를 해결하기위해 함수형 언어뿐만이 아닌 다른 언어 형식을 바탕으로 하는 파서생성기 생성기 역시 개발해 이들을 옵션에 따라 사용자가 원하는 언어 형식으로 사용할 수 있도록 확장하는 방법을 생각할 수 있다. 또한 생성되는 파서의 구조가 매우 간단하므로 사용자가 원하는 기능을 확장하는 것도 기존의 파서 생성기를 바탕으로 확장하는 방법에 비해서 상대적으로 수월하다. 무엇보다도 파서생성기와 늘 함께 사용되곤 하는 렉서 생성기(Lexer-Generator)를 위한 렉서생성기 생성기(Lexer-Generator Generator)를 본 논문에서 소개한 방법과 비슷한 방법을 이용해 개발하여, 사용자로 부터 입력받은 하나의 번역 모듈을 통해 렉서생성기와 파서생성기가 통합된 형태의 어휘· 구문 분석기를 생성할 수 있게된다면, 렉서와의 연결시 제한사항이 발생하던 문제를 해결함은 물론 두 가지 유용한 도구를 하나의 번역 모듈을 통해 간단히 만들어 낼 수 있으므로 컴파일러 전단부(front-end) 개발의 효율은 한층 더 향상될 수 있을 것이다.

참고문헌

[1] Stephen C. Johnson. YACC: Yet Another Compiler-Compiler. Unix Programmer's Manual Vol 2b, 1979.

[2] Berkeley YACC, http://invisible-island.net/byacc/byacc.html
[3] Objective Caml, http://caml.inria.fr/ocaml/
[4] A Compact Guide To Lex & Yacc, http://epaperpress.com/lexandyacc/download/lexyacc.pdf
[5] Ocamllex Tutorial, http://plus.kaist.ac.kr/ shoh/ocaml/ocamllex-ocamlyacc/ocamllex-tutorial/

김 태 경

– 2001-2007 포항공과대학교 학사
– 2007-현재 포항공과대대학교 석사 과정

<관심분야> 컴파일러 설계, 프로그래밍 언어 이론

박 성 우

– 1991-1996 한국과학기술원 학사
– 1996-1998 한국과학기술원 석사
– 1999-2005 Carnegie Mellon University 박사
– 2006-현재 포항공과대학교 교수

<관심분야> 프로그래밍 언어 이론, 전산 논리