논문 2011-1-26

LL과 LR에서의 효율적인 수식 타입 체킹에 대한 연구

A Research on the Efficient Type-Checking for an Expression Using LL and LR

정용주[*]

Yongju Chung

요 약 수식의 자료형 확인을 위하여 많이 사용하는 방법의 하나가 속성문법이다, 그런데 이 속성문법은 구성하기가 어려운 단점이 있다. 파싱 과정을 잘 이해하고 속성문법의 두 개의 속성들을 사용하여야 하기 때문이다. 그래서 본 논문에서는 이 수식 속성문법의 구성을 보다 쉽게 하기 위한 방법을 제시한다. 문법 구성을 통하여 하나의 속성만으로 수식의 자료형 확인이 가능함을 보여준다.

Abstract One of the methodologies for the type-checking of an expression is the attribute grammar. But this attribute grammar is difficult to write because two attributes should be used with the full understanding of parsing steps. So this paper proposes a methodology to construct an expression attribute grammar easily. It shows the possibility to check the type of an expression with only one attribute through a grammar construction method.

Key Word : 수식, 파싱, type checking, 속성문법, 상속속성, 합성속성, LL, LR, 형변형

Ⅰ. 수식과 속성문법

수식(expression)은 보통의 프로그래머들에게는 쉽다고 생각한다. 이것은 일상에서 많이 접했기 때문인데 그러나 수식은 언어 개발자에게는 복잡한 부분이다. 수식에는 산술식, 논리식, 관계식 등이 있고 괄호 또 우선순위가 존재하기 때문이다. 더욱이 수식에는 또 하나 고려하여야 할 것이 자료형 확인이다. 수식은 구문적으로 맞았다고 하여도 의미론적으로 올바르게 구성되었는지 확인하여야 한다.

자료형 확인을 위해서 많이 사용되는 방법이 속성문법이다[1][2][3][4]. 속성 문법은 1968년 Knuth가 발표한 정적 의미론을 위한 규칙들인데 파싱할 때 구문 안에서의 의미가 일치하는 지 확인하기 위한 규칙들이다. 이 문법에는 상속 속성(inherited attribute)와 합성 속성(synthesized attribute) 두 속성을 사용하여 표현하며 수식이 의미론적으로, 즉 자료형들이 옳게 구성이 되어있는지 확인한다. 이 속성문법은 기존 문법에 의미론 부분만, 두 가지 속성만, 첨가하여 사용할 수 있으므로 현재 언어 개발을 할 때는 물론 구문 지향 편집 시스템, 자연언어 처리 시스템 등 많이 사용하는 방법이다[5][6]. 그러나 그 작성 방법이 어려운 단점이 있다[7]. 파싱되는 과정 과정을 이해하여야 하고 거기에 맞추어 넌터미널들의 속성을 계산 혹은 결정하여야 하기 때문이다. 그래서 이 속성문법을 보다 쉽게 작성할 수 있으면 좋으나 아직은 그렇지 못한 것 같다. 그래서 본 논문에서는 새로운 방식을 보여준다. 이 방식은 하나의 속성만을 사용하는 방식으로 속성값을 단순히 상향으로 전달하는 방식이므로 속성문법 구성을 보다 쉽게 할 수 있다. 이것은 수식의 특성을 고려하여 수식문법을 구성한 방식이다. 그러나 이 방

[*]정회원, 단국대학교 컴퓨터학부
접수일자: 2011.1.3 수정일자: 2011.1.29
게재확정일자: 2011.2.11

식에는 단점이 있다. 이 방식은 수식 안에서 어느 정도의 부분적인 속성 확인은 가능하나, 기존 방식에 비교하여 속성 확인의 세밀함이 떨어진다. 즉, 어느 특정 변수가 전체 수식 속성에 맞지 않는다든지 어느 부분의 결정 속성이 전체 속성과 맞지 않는다든지 하지는 못한다. 어느 정도의 부분적인 속성 확인은 가능하지만, 전체 최종 수식 속성이 사용된 문장에서 요구된 속성과 일치하지 않는다든지 하는 정도이다. 그러나 소규모 집단을 위한 언어나 아니면 간단한 수식을 많이 사용하여 수식에서 변수들의 속성들을 어렵지 않게 확인할 수 있는 언어라면 유용하게 사용될 수 있다고 판단된다.

본 논문의 구성은 2절에서는 수식 속성문법의 어려운 점을 기술하고 3절에서는 본 논문에서의 LR 방식을 위한 수식 문법을 보여주고 4절은 본 논문에서의 방식을 LL 파싱의 경우에, 그리고 5절은 결론 부분이다. LR을 먼저 기술하는 이유는 LR의 수식 문법이 더 익숙하기 때문이다.

II. 수식을 위한 속성문법

보통의 수식은, 앞 절에서 기술하였듯이 여러 가지 고려하여야 할 사항들 때문에 파서를 만드는 경우에는 복잡하다. 그래서 많은 언어론 교재나 컴파일러 교재에서 수식을 위한 문법으로, 간단한 산술식이라도 5-6개의 규칙들을 사용하여 표기한다. 다음은 산술식을 나타내는 대표적인 수식 문법이다[2][3][4][7].

<expr> → <expr>+<term> | <term>
<term> → <term>*<factor>| <factor>
<factor> → <id> | (<expr>) … 문법 (i)

이 수식 문법이 좋은 이유는 '+', '*' 그리고 괄호만 있는 산술식이지만 '-', '/' 들을 포함시키면 이러한 연산자들로 이루어진 어떠한 산술식도 표현이 가능하므로 많이 사용되는 방법이다. 이 표시 방법의 또 다른 장점은 우선순위를 자연스럽게 해결한다는 것이다. LR 파싱을 하면 터미널로부터, 즉 문장에서부터 시작하므로 '*'이 먼저 처리되고 뒤에 '+'가 처리되므로 자연스럽게 우선순위를 해결한다.

(i)의 문법으로 어떠한 수식이 작성되었다고 하자. 예를 들면 y = x1 + x2이다. 문법적으로는 틀림이 없다. 그러나 y가 int일 때 잘 알려져 있듯이 x1과 x2는 int이어야 한다. real이나 boolean이 오면 에러이다. 파서는 이것을 확인하여야 한다. 문법적으로는 맞았지만 의미론적으로 맞는지를 확인하여야 하기 때문이다. 이러한 타입 체킹, 즉 속성 확인이 어려운 이유는 파싱을 하며 동시에 속성도 확인하여야 하기 때문이다. 식에 대한 파싱이 끝난 다음 다시 식 안의 변수나 상수들을 확인하기는 곤란하다. 이 속성 확인을 위하여 많이 사용하는 방법이 속성문법이다. 그러나 이 속성문법 작성은, 앞서 기술하였지만, 파싱 과정을 정확히 이해를 하여야 하기에 복잡하다. 더구나 두 개의 속성을 사용하므로 더욱 그렇다.

앞서 기술하였지만 <expr>을 수식을 대표하는 넌터미널로 하고 수식을 위한 문법으로 (i)을 많이 사용한다고 하였다. 그래서 프로그래밍 언어의 여러 구문에서 수식이 사용되는 부분에는 이 <expr>을 넣어 문법 표기를 한다. 예를 들면 배정문, if문, while문 등에서 배정문의 경우라면 <assign> → <id> '=' <expr>, if문이라면 <id> →'if' '(' <expr> ')' <stmts>와 같은 방식이다. 그래서 이 <expr>로부터 모든 수식이 생성된다.

수식은 그 종류가 산술식, 관계식 그리고 논리식으로 나뉜다. 이것은 보통 언어에서 수식에 사용되는 십여 개 이상의 연산자를 중심으로 나눈 방식이다. 연산자에는 우선순위가 존재하며 또한 연산자의 조건이 있다. 산술 연산자와 관계 연산자 좌우에는 수치적 변수나 수치적 값이 올 수 있으나 논리 연산자 좌우에는 오직 논리형 변수, 논리값이 있어야 한다. 그리고 수식이 계산을 마친 후 결과의 자료형으로는 논리값(true, false) 혹은 int 혹은 real이 될 수 있다. 수식 속성문법 작성의 어려움은 수식의 결과가 이와 같이 여러 자료형이 될 수 있기 때문이다.

수식은 그 자체가 문장을 이루지는 않고 문장 안의 한 일부로 사용된다. 앞에 예를 든 배정문이나 if문과 같다. 그리고 수식은 확인하여야 할 속성이 파싱 전에 미리 정해진다. if문이라면 수식은 논리형이어야 하고 배정문에서는 등호의 좌측이 int이라면 수식이 int, real이라면 real이어야 한다. 그리고 이 미리 정해진 속성값을 수식 부분에서 확인하여야 한다. 바로 이 여러 개의 속성값을 가질 수 있기에 수식은, 수식을 대표하는 <expr> 하나를 사용하여, 속성확인이 어렵다. 수식을 나타내는 문법 부분 하나이고 여러 개의 자료형 결과가 될 수 있는 상황이다. 어떠한 언어에서 산술식과 관계식만 있으면 속성확인은

비교적 간단하다. 심볼 테이블에서 찾아진 속성값을 단순히 상향으로 전달하여 산술식 혹은 관계식 부분에서 속성 확인을 하면 되기 때문이다. 그러나 수식에 논리형 수식까지 포함하면 간단하지 않다. 논리식은 단순히 논리 연산자와 논리형 변수만이 있기도 하겠지만 관계식을 사용하여 그 관계식의 결과로 true, false를 나타내면서 논리 연산자를 사용하는 경우가 더 많아서 수치적 자료의 변수와 논리형 변수가 혼합되어 있는 상황이다. 그래서 산술식이나 관계식만이 있는 언어의 경우와 같이 단순하게 처리하기에는 무리가 있다. 즉, 수식을 나타내는 문법 부분 안에 속성문법을 사용하여 <expr>을 경우에 따라서 int형으로 혹은 real형으로 혹은 논리형으로 결정 지어주기는 상황이다. 그래서 두 속성을 사용하는 속성 문법을 많이 시용한다.

Ⅲ. 본 논문에서의 수식 문법 구성 방법

수식같은 구문을 하나의 속성만으로 속성문법을 작성하기 위해서 본 논문에서는 문법 구성 방법을 통하여 시도하였다. 즉, 수식을 위한 문법 안의 넌터미널들과 각 문법규칙들에 의미를 내포시켜서 해결해 보였다.

수식은, 수식의 종류가 세 종류이므로 그들을 각각 따로 따로 생각할 수 있다. 그러나 서로 간의 포함 관계로 볼 수도 있다. 예를 들어 논리식은 논리 연산자와 논리 변수만으로 구성 될 수도 있으나 많은 경우 관계식을 포함시켜서 구성되어 있다. 그리고 관계식은 피연산자로 하나의 변수나 상수가 올 수도 있으나 여러 변수로 이루어진 산술식이 오는 경우가 더 많다. 따라서 산술식은 관계식의 일부가 될 수 있고 또한 관계식은 논리식의 일부가 될 수 있다. 따라서 이와 같은 개념으로 본 논문에서는, 수식이 삼분법적으로 세 가지로 이루어져 있다고 할 수도 있겠지만, 어차피 구문 분석, 파싱이 목적이라면 수식 종류들을 포함 관계로 해석하였다. 즉 논리식은 관계식을 포함하고 관계식은 산술식을 포함한다는 것이다. 그래서 본 논문에서의 수식을 위한 문법 구성 방식을 논리식을 출발점으로 하였다. 모든 수식은 논리식으로 놓고 그 논리식에서 관계식이 생성될 수 있고 또한 그 관계식에서 산술식이 생성될 수 있게 하는 개념이다. 즉, 논리식을 관계식보다 상위개념으로 보았고 관계식을 산술식의 상위개념으로 보았다. 그래서 논리식은 관계식의 상위개념이기에 논리식 안의 관계식은 논리식 안의 또 다른 논리형 변수 수준으로 본다. 산술식과 관계식의 관계도 동일하다.

위와 같은 개념에 근거하여 각 수식 종류들을 위한 넌터미널들이 각각 서로 “계층적”인 관계를 갖도록 하였다. “계층적”의 의미는 논리식을 위한 넌터미널들에서 관계식을 위한 넌터미널들을 생성할 수는 있으나 그 반대의 경우는 안 되고 또한 관계식을 위한 넌터미널들에서는 산술식을 위한 넌터미널들이 생성될 수는 있으나 그 반대는 생성하지 않게 하는 것이다. 또한 각 종류의 수식에는 각각을 대표하며 나타내는 넌터미널을 두었다. 그래서 상위 계층의 규칙에서 현 단계의 수식 종류를 생성한다면 반드시 그 대표하는 넌터미널을 만들도록 하였다. 이러한 방식은 파싱 과정에서 어느 한 쪽으로 넌터미널이 생성되면 그 이전 단계로 다시 환원될 수 없음을 의미한다. 일단 관계식에서 산술식을 위한 넌터미널로 바뀌면 이것은 다시 관계식으로 환원이 될 수 없음을 의미하는데 이와 같은 방식으로 문법 구성을 하면 넌터미널들과 규칙들의 각 단계에서의 의미가 분명하여진다. 즉 어떠한 수식의 파싱 과정에서 어느 단계까지 파싱이 되었다면, 예를 들어 산술식 부분까지 파싱이 되었다면, 그 부분은 파싱이 종료된 것으로 바로 그 부분에 필요하면 술어함수를 넣으면 산술식인지의 확인이 되는 것이다. 관계식 부분 규칙까지 파싱이 되었다면 그 부분에 필요한 술어함수를 넣는 방식이다. 여기서 주의할 것은 관계 연산자가 사용되는 규칙이다. 이 부분은 그 결과가 논리형으로 변형되는 부분이다. 따라서 해당 넌터미널에 속성값 변화를 확정적으로 논리형이라고 주어야 한다.

표 1은 이와 같은 개념을 도입한 수식을 위한 문법이다. 우선 순위는 여러 컴파일러 교재나 언어론 교재에서 사용하는 방식을 도입하였다. 즉 우선순위가 낮은 것을 먼저 생성시키고 우선순위가 높은 것은 나중에 생성시키는 방식이다.

규칙(1)~규칙(6)은 논리식을 위한 규칙들이고 그중 규칙(1)~규칙(3) 규칙들은 논리합(or)과 논리곱(and)을 나타내기 위한 규칙들인데 논리곱(and)은 논리합(or)보다 우선순위가 높기에 나중에 생성시킨다. 이것은 산술식 규칙들에도 동일하다. 규칙(5)에서 관계식을 위한 <r-expr>을 생성하고 규칙(7)~규칙(9)는 관계식을 위한 규칙들이다. 규칙(7)과 규칙(8)에서 <arith-expr>이 생성되고 규칙(10)~규칙(15)는 산술식을 위한 규칙들이

다. 각 수식 종류 관련 규칙들의 마지막 규칙 혹은 바로 그 이전 규칙에서 다음 단계 수식을 위한 넌터미널이 생성된다. 각 넌터미널들과 규칙 안의 여러 개는 그 의미가 내포되어 있다. 예를 들어 규칙(7) 혹은 (8)의 <arith-expr>은 그 부분에 나타날 산술식을 전체를 나타내고 어떠한 수식의 파싱이 규칙 10까지 파싱이 되었다면 산술식 부분이 종료된 것이다.

표 1. 수식을 포함 관계로 구성한 문법

```
<expr>[1]   →  <expr>[2]'or'<l-term>… (1)
<expr>       → <l-term>… (2)
<l-term>[1]→ <l-term>[2] 'and'
               <l-factor> … (3)
<l-term>     →   <l-factor>… (4)
<l-factor>   → <r-expr> … (5)
<l-factor>   → (<expr>) … (6)
<r-expr>[1] → <r-expr>[2] ('>'|'>='|'<'|
      '<=' | '==' | '!=') <arith-expr> … (7)
<r-expr>    →    <arith-expr> … (8)
<r-expr>[1] →   (<r-expr>[2])  … (9)
<arith-expr>[1] → <arith-expr>[2] '+'
              <arith-term>  …  (10)
<arith-expr>→<arith-term> … (11)
<arith-term>[1]→<arith-term>[2]'*'
<arith-factor>  …       (12)
<arith-term>[1]→<arith-factor>              … (13)
<arith-factor> → id … (14)
<arith-factor> → (<arith-expr>) … (15)
```

표2는 표1을 위한 속성문법이다.

표 2. 표1에 대한 속성 문법

```
규칙1: <expr>[1].type ← <l-term>.type
규칙2: <expr>[1].type ← <l-term>.type
규칙3: <l-term>[1].type← <l-factor>.type
규칙4: <l-term>[1].type ← <l-factor>.type
규칙5: <l-factor>.type ← <r-expr>.type
규칙6: <l-factor>.type ← <expr>.type
규칙7: <r-expr>[1].type  ← 3
   (술어함수)   (<arith-expr>.type   <   3)   and
(<r-expr>[2].type < 3 )
규칙8: <r-expr>[1].type  ← <arith-expr>.type
규칙9: <r-expr>[1].type  ← <r-expr>.type
규칙10:<arith-expr>[1].type                     ←
max(<arith-expr>[2].type, <arith-term>.type)
   (술어함수)  (<arith-expr>[2].type   <3     and
<arith-term>.type < 3)
규칙11:<arith-expr>.type ← <arith-term>.type  규칙
12:<arith-term>[1].type ← max((<arith-term>[2].type,
<arith-factor>.type)
   (술어함수)(<arith-term>[2].type<3)          and
(<arith-factor>.type <3)
규칙13:<arith-term>.type  ←     <arith-factor>.type

규칙14:<arith-factor>.type←look_up(id)          규칙
15:<arith-factor>.type←<arith-expr>.type
```

본 논문에서는 하나의 속성만 필요하므로 단순히 그 속성을 type이라 하고 그리고 이 type은 언어가 int, real, 논리형(bool)만을 사용하는 것으로 하였다. 또한 type 결정의 편리함을 위하여 본 논문에서는 각 자료형 int, real, 논리형들을 수치적 값 1, 2, 3으로 설정하고 계산하도록 하였고 자동형변형을 허락하는 것으로 하였다. 규칙(10)과 (12)에서 자동형변형을 허락하므로 max() 함수를 사용한다. 이 함수는 두 인자 중 큰 값을 반환하는 함수이다. 따라서 int, real이면 max(1,2)가 되어 2를 반환한다. 규칙(14)의 look-up() 함수는 심볼 테이블에서 해당 id의 자료형을 찾아 1 혹은 2 혹은 3 중 어느 한 값을 배정하는 함수이다. 규칙(7)에는 수식 자체의 속성값 변화이다. 산술식이 관계연산자를 사용 그 결과는 논리값이 되므로 수식 자체의 논리값은 3으로 하여야 한다. 그리고 술어함수들은 변수들이 같이 나오는 경우에 작성한다.

그림1은 배정문 규칙에 본 논문의 방식을 적용하여 본 파스 트리와 속성값의 흐름도이다. 참고문헌[6]에서 제안한 대로 이 if문에 술어함수를 넣는다.

배정문 규칙과 술어함수는 다음과 같이 하였다.

규칙: <assign> → <var> '=' <expr>

술어함수: <var>.type=<expr>.type

그림 1을 보면 x1, x2의 실질 자료형이 심볼 테이블에서 찾아져서 상향식으로 type 값들이 전달된다. 그리고 '+'를 파싱하여야 하는 단계에서(구문 규칙(10)) 자동형변형이 되는지 계산한다. 계속적으로 속성값이 전달되어 <expr>의 속성값이 결정된다. 그래서 배정문의 술어함

수에서 배정문의 속성이 옳은지를 확인한다.

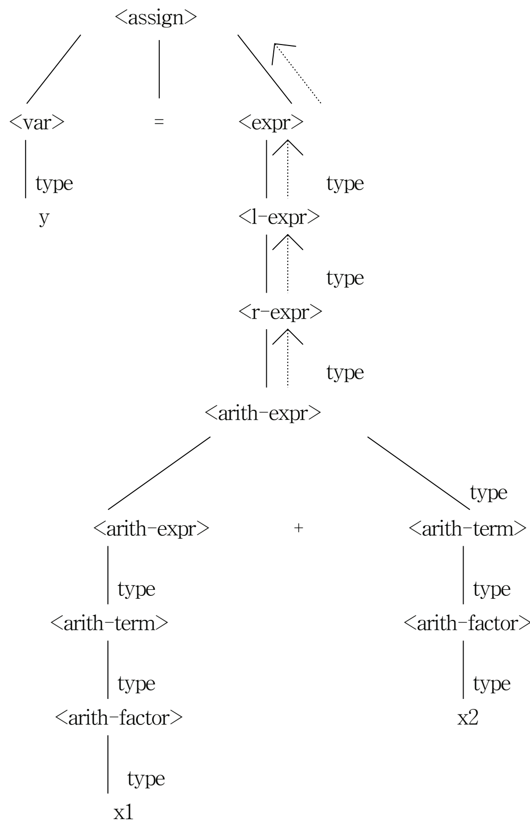

그림 1. y = x1 + x2의 속성의 흐름

if문의 경우이다. if문의 규칙과 술어 함수는 다음과 같이 한다.

규칙: <if> → ‘if’ ′(′ <expr> ′)′ <stmt>

술어함수 : <expr>.type = 3

첫 번째 예로 하는 if문은 if(x1 + x2) y= ...;으로 논리식이 있어야 할 곳에 산술식만 있다. 이 경우 <expr>을 위한 속성 흐름도는 그림1의 <expr> 부분까지는 동일하다. 그러나 그 부분까지의 <expr>의 값은 1 아니면 2이다. 속성문법 규칙(7)을 지나지 않은 까닭이다. 그래서 if문의 술어함수에서 에러가 된다.

두 번째 경우는 옳게 단일 변수로 if문이 작성된 경우이다.

if(x) ....; // x는 boolean

와 같은 경우 x는 논리 자료형으로 선언된 변수이다. 이 경우 규칙(14)에서 속성값으로 3이 찾아진다. 그리고 이것은 규칙(13), 규칙(11), 규칙(8), 규칙(5), 규칙(3), 규칙(10을 통하여 <expr>까지 전달된다. 이 값은 if문의 술어함수에서 true로 결정된다.

Ⅳ. 결정적 LL의 경우

결정적 LL의 경우는 하향식으로 전달만 하므로 기본적으로 두 개의 속성이 필요하지 않지만 본 논문의 수식문법 개념을, 계층적 구성 방법을, 결정적 LL 파싱의 경우에도 적용하여 보았다. 이것은 Yacc가 LR의 경우와 같이 결정적 LL의 경우도 같은 방식으로 작동하다는 전제로 구성하였다. 그리고 이 경우에는 논리연산자는 없고 관계 연산자와 산술 연산자만 있는 경우로 하였다.

표 3. 관계, 산술 연산자만을 사용하는 수식의 결정적 LL 문법 ({}안은 lookahead)

<r-expr>→<r-term><r-expr-p>,{(,id,$}··· (1)
<r-expr-p> → (‘>’|‘>=’|‘<’|‘<=’|‘==’|‘!=’)
<r-term>,{>,>=,|,<,,<=,,==,!=} ··· (2)
<r-expr-p> → ε, {),$} ··· (3)
<r-term> → (<r-expr>), {(} ··· (4)
<r-term> → <arith-exp>, {id, (} ··· (5)
<arith-exp> → <arith-term><arith-exp-p>,
{id, (} ··· (6)
<arith-exp-p>→′+′<arith-term><arith-exp-p>,
{+} ··· (7)
<arith-exp-p>→ε,{),$,>,>=,|,<,,<=,,==,!=}···(8)
<arith-term>→<arith-factor><arith-term-p>,
{id, (} ··· (9)
<arith-term-p>→
′*′<arith-factor><arith-term-p>,{*} ··· (10)
<arith-term-p>→ ε,
{),$,>,>=,|,<,,<=,,==,!=}···(11)
<arith-factor> → id, {id} ··· (12)
<arith-factor> → (<arith-exp>), {(} ··· (13)

표 4. 표3에 대한 속성 문법

```
규칙1: <r-term>.type ← <r-expr>.type
       <r-expr-p>  ← <r-expr>.type
규칙2: <r-term>.type ← 2
규칙3:
규칙4: <r-expr>.type ← <r-term>.type
규칙5: <arith-exp>.type ← <r-term>.type
규칙6: <arith-term>.type←<arith-exp>.type
       <arith-exp-p>.type←<arith-exp>.type
규칙7: <arith-term>.type←<arith-exp-p>.type
<arith-exp-p>.type←<arith-exp-p>.type
규칙8:
규칙9:<arith-factor>.type←<arith-term>.type
    <arith-term-p>.type←<arith-term>.type
규칙10:
  <arith-factor>.type←<arith-term-p>.type
  <arith-term-p>.type←<arith-term-p>.type
규칙11:
규칙12:(술어함수)
 switch(<arith-factor>.type){
  1: look_up(id)) = 1; break;
  2: look_up(id)) <= 2; break;
  3: look_up(id) = 3
   }
규칙13:<arith-expr>.type←<arith-factor>.type
```

표2의 경우와 같이 look-up()은 심볼 테이블에서 속성값을 찾는 것을 의미하고 역시 int, real, 논리형을 각각 1, 2, 3으로 하였다.

3절에서의 예와 같은 배정문 y=x1+x2의 경우, 규칙 적용 순서는 1, 5, 6, 9, 12, 7, 9, 12, 11, 3이고 이 규칙 순서에서 규칙 12의 첫 번째 적용은 x1을 위한 속성확인이다. 가령 y가 int 이라면 switch 문의 case 1인 경우에 해당되므로 1(int)가 아니면 에러이다. 가령 y가 real이고 자동형변형을 허락하면 이 부분에서 case 2와 같은 조건이 된다.

if(x1 > x2) ... 의 경우 x1에 대해서는 위와 동일하게 1, 5, 6, 9, 12이며 그 뒤는 8, 2, 5, 6, 9, 12, 11이다. 이 규칙 순서에서 속성규칙 2를 지나며 if문에서 요구했던 3(논리형)이 2로 변형된다. 따라서 속성문법 12에서 x1과 x2가 각각 산술형임을 확인한다.

결정적 LL의 경우에도 속성값을 단순히 내리면 되므로 속성문법 작성이 보다 단순해졌다.

V. 결론

본 논문에서는 수식을 위한 속성문법 구성을 보다 단순하게 할 수 있는 수식 문법 구성 방법에 관하여 연구하였다. 이것은 문법 구성을 계층적으로 하여서 가능하였다. 수식 구문 문법 구성을 세 종류의 수식 종류에 따라서 서로 분리되어 계층적으로 작성하므로 각 넌터미널들은 내재된 의미를 갖게 되어 파싱되는 과정 과정이 의미를 지니게 할 수 있었고 그래서 하나의 속성만 사용하여 속성문법을 작성하므로 속성문법 작성이 보다 쉬워졌다. 그러나 이 방식의 단점은 두 가지 속성을 모두 사용하는 경우보다 에러 탐색의 세밀함은 떨어진다. 각 종류의 수식을 대표하는 넌터미널에 대하여 속성확인을 하므로 특정 변수를 찾지는 못한다. 그러나 최근의 번역 장치들이 수식에 관해서는 비교적 간단히 에러 차리를 하는 경향인 것을 고려하면 나름대로 의미가 있다고 판단된다. 또한 본 논문에서의 예는 간단한 언어의 경우의 예이지만 언어가 다양한 기능을 갖추었을 때, 예를 들면 C는 if(x1=x2)도 가능한데 이러한 기능들, 이러한 때도 본 논문의 문법 구성 방법이 가능할 지는 더 연구를 하여야 한다.

참고 문헌

[1] Knuth, D. E., "Semantics of Context-Free Languages", Mathematical Systems Theory, Vol. 2, pp. 127-146, 1968

[2] Lewis, P. M., D. J. Rosenkrantz and R. E. Stearns, “Compiler Design Theory" Addison-Wesley 1976

[3] Alfred V. Aho, etc., Compilers, Principles, Techniques and Tools, Addison-wesley, 1986

[4] Meyer, B. Introduction to the Theory of programming Languages, Prentice-Hall, Englewood Cliffs, NJ, 1990

[5] Farrow, R. "Linguist 86: Yet Another Translator Writing System Based on Attribute Grammar", ACM SIGPLAN Notices, Vol. 17, No.6, pp.160-171, 1982

[6] N. Correa, R. C. Berwick etc. "Empty Categories, Chain Binding and Parsing", pp.83-121, Kluwer

Academic Publisher, New York, 1992
[7] Robert W. Sebesta, Concepts of Programming Languages, 7th ed., Pesrson Education, 2006

저자 소개

정 용 주(정회원)

- 1977년-1981년 : 서울대학교 자연대 계산통계학과졸
- 1981년-1983년 : KAIST 전산학과졸
- 1983년-1984년 : 단국대학교 통계문제연구소 연구원
- 1984년 : 단국대학교 공학대 컴퓨터학부 교수
- 1995년-1998년 : RPI 박사과정, 신병으로 휴학후 다시 단국대 복직.