객체지향 속성 문법을 이용한 XML 문서 처리기 생성기

(An XML Document Processor Generator using Object-oriented Attribute Grammar)

최 종 명 † 유 재 우 ††
(Jong-Myung Choi) (Chae-Woo Yoo)

요 약 XML 문서 처리기는 XML 문서를 문서의 목적과 의미에 맞게 처리할 수 있어야 한다. XML의 DTD는 문서의 구조적인 정보만 제공하고 의미 정보는 제공하지 않기 때문에 문서 처리기를 자동적으로 생성하기 어렵다. 본 논문에서는 객체지향 속성 문법을 이용해서 XML 문서의 의미 정보를 기술하는 방법과 XML 문서 처리기를 자동적으로 생성할 수 있는 생성기를 소개한다. 문서 처리기 생성기는 문서 처리기를 작성해야 하는 개발자의 시간과 노력을 감소시켜줄 것이다.

키워드 : XML 문서 처리기 생성기, 속성문법

Abstract An XML document processor should process the XML documents according to their purposes and semantics. It is very hard to automatically generate an XML document processor with DTD, because it does not provide the semantic information. In this paper, we introduce an XML document processor generator and a method for specifying semantics using the object-oriented attribute grammar. The XML document processor generator will reduce costs and efforts in developing XML document processors.

Key words : XML document processor generator, attribute grammar

1. 서 론

XML은 문서와 데이타를 구조적으로 표현하기 위해서 사용되는 표준화된 메타언어이다[1]. XML은 플랫폼 독립성, 호환성, 자동화된 문서 처리 등의 장점 때문에 일반 문서, 웹 문서, 그래픽 정보[2,3], 메타 정보[4], 프로토콜[5] 등의 분야에서 널리 사용된다. XML은 활용 분야와 목적에 따라 문서의 문법적인 사항을 DTD (Document Type Definition)를 이용해서 정의하고, 문서가 유효한(valid) XML 문서가 되기 위해서는 반드시 DTD에 맞게 작성되어야 한다. XML 파서는 DTD를 이용해서 XML 문서의 문법적인 사항은 자동적으로 검사할 수 있는 기능을 제공하기 때문에 문서의 유효성은 자동적으로 파악될 수 있다.

· 본 연구는 숭실대학교 교내 연구비 지원으로 이루어졌음
† 비 회 원 : 숭실대학교 컴퓨터학과
jmchoi@comp.ssu.ac.kr
†† 종신회원 : 숭실대학교 컴퓨터학부 교수
cwyoo@comp.ssu.ac.kr
논문접수 : 2003년 4월 18일
심사완료 : 2003년 8월 2일

XML 문서의 데이타는 사용되는 목적에 따라 다양한 형태로 처리되어야 한다. 어떤 경우에는 XML 문서를 다른 형태의 문서로 변환하기도 하고, 또 다른 경우에는 XML 문서의 내용을 해석해서 결과를 출력해야 한다. 예를 들어, 프로그램을 XML로 표현하는 경우[6]에 문서 처리기는 프로그램의 내용에 맞게 기계어 코드를 생성하거나 프로그램을 해석해서 실행시킬 수 있을 것이다. 이렇게 XML 문서의 내용을 의미에 맞게 처리해야 할 필요성이 있는 경우에 일반적으로 SAX[7]와 DOM[8]과 같이 표준화된 API를 이용해서 문서 처리기를 작성하게 된다. 이처럼 문서 처리기를 API를 이용해서 작성하는 것은 많은 시간과 노력을 필요로 한다. 또한 XML 문서 구조가 작고, 복잡하지 않은 경우에는 이러한 API를 이용해서 프로그램을 작성하는 것이 허용되겠지만, 문서 구조가 복잡한 경우에는 API를 이용한 프로그램 작성이 어려워진다. 따라서 XML 문서 처리기를 자동적으로 생성할 수 있는 방법과 문서 처리기를 자동적으로 생성하기 위해서 필요한 의미 정보를 기술할 수 있는 방법이 필요해진다. XML의 DTD는 문서의 문법적인 사항들은 제공하지만, XML 문서가 어떤 의미를

가지고 있으며, 어떻게 XML 문서를 처리해야 하는지에 대한 정보들은 제공하지 못한다. 따라서 XML에도 의미 정보를 기술할 수 있는 방법이 필요하다.

프로그래밍 언어 분야에서는 언어의 문법을 BNF를 이용해서 기술하고, 언어의 의미는 속성 문법(attribute grammar)을 이용하는 방법이 널리 사용되고 있다. SGML과 XML에서도 컴파일러 기법과 속성 문법을 적용해서 구조화된 문서를 자동적으로 변환하기 위한 노력들이 있어왔다. 그러나 지금까지 SGML과 XML 문서를 처리하기 위해서 사용된 속성 문법은 Knuth[9]에 의해 제안된 기본적인 형태이며, 보다 진보된 속성 문법을 사용하기 위한 시도는 상대적으로 적었다. 현재 많은 응용프로그램들이 객체지향 언어를 이용해서 개발되고 있기 때문에 XML 문서를 객체지향 언어에서 쉽게 접근하고, 해석할 수 있기 위해서는 XML 문서 처리에서 객체지향 속성 문법[10,11]을 사용해야 할 필요성이 있다. 본 논문에서는 객체지향 속성 문법을 이용해서 XML 문서의 의미를 기술하는 방법과 XML 문서 처리기를 자동적으로 생성할 수 있는 생성기를 소개한다. 객체지향 속성 문법은 객체지향 프로그래밍에 적합하며, 모듈성이 있는 장점을 가지고 있다. XML 문서 처리기 생성기는 의미 정보를 통해서 문서 처리기를 자동적으로 생성할 수 있고, 생성된 문서 처리기는 주어진 의미에 맞게 XML 문서를 변환하거나 실행할 수 있기 때문에 프로그래머의 시간과 노력을 감소시켜줄 것이다.

논문은 2장에서는 기존에 연구된 관련 연구들을 소개하고, 3장에서 XML에 객체지향 속성 문법을 적용해서 의미 정보를 기술하는 방법을 제시한다. 4장에서 의미 정보를 이용해서 XML 문서 처리기를 생성할 수 있는 생성기 시스템을 소개하고, 5장에서 결론과 향후 연구 과제를 밝힌다.

2. 관련 연구

속성 문법은 프로그래밍 언어의 의미(semantics)를 기술하기 위한 방법으로 Knuth에 의해 소개되었다. 프로그래밍 언어에서 속성 문법을 이용해서 의미를 기술하는 것과 유사하게 XML 문서를 처리하기 위해서도 속성 문법을 사용하려는 시도가 있어 왔다. Warmer[13]는 SGML에서 YACC 스타일로 DTD에 C 코드를 추가하는 방법을 이용해서 SGML 문서를 처리하는 문서 처리기를 자동적으로 생성하는 연구를 수행하였다. Warmer는 DTD에서 원소 선언을 프로시저로 변환하고, SGML의 특정 문서 파서에 의미 정보를 추가하는 방법을 사용한다. Giuseppe[14]는 XML에서 속성 문법을 이용해서 의미를 기술하기 위한 방법으로 별도의 XML 파일에 의미 정보를 사용하는 방법을 사용한다.

XML 문서에 속성 문법을 사용하는 기존 연구들은 본 논문에서 제안하는 방법과는 기본적으로 다음과 같은 차이점을 가지고 있다. 첫째로 기존의 SGML과 XML 문서에 속성 문법을 이용해서 자동적으로 처리하기 위한 연구들은 주로 SGML 혹은 XML 문서를 다른 형태의 문서로 변환하기 위한 목적으로 수행되었다. 이것은 관련 연구들이 수행될 당시에는 SGML 혹은 XML 문서가 응용프로그램을 위한 일반적인 데이타 혹은 보다 광범위한 용도로 사용되지 않았고, 주로 사람이 읽기 위한 데이타를 주로 다루었기 때문이다. 따라서 당시의 주된 관심은 SGML/XML 문서를 TeX 혹은 다른 문서 형태로 변환하는 것이었다. 그러나 현재 XML에서 문서 형태를 변환하는 것은 속성 문법 보다는 훨씬 간단한 형태의 XSLT를 사용함으로써 해결할 수 있다. 그러나 XML 문서가 응용프로그램의 데이타 혹은 프로그램 형태로 사용되는 경우에는 XSLT를 이용해서는 원하는 작업을 수행할 수 없기 때문에 문서에 포함된 데이타를 획득, 변환 혹은 실행할 수 있는 방법이 필요하다.

둘째로 기존에 XML과 SGML에서 사용되는 속성 문법은 Knuth에 제안된 기본적인 형태가 주로 사용되었기 때문에 객체지향 특성을 지원할 수 없다. 현재 많은 응용프로그램들이 객체지향 언어를 이용해서 작성되기 때문에 이러한 응용프로그램에서 XML 문서의 데이타를 사용하기 위해서는 객체지향 기술이 지원되어야 한다.

셋째로 객체지향 속성 문법을 사용하는 경우에 재사용성과 확장성이 높아지는 장점을 가지고 있다. 본 논문에서 소개하는 XML 문서 처리기는 객체지향 프로그래밍의 visitor 패턴을 사용하기 때문에 비교적 간단한 방법으로 XML 문서를 다른 형태로 처리할 수 있는 확장성을 제공한다.

객체지향 속성 문법과 유사하게 XML 문서의 구조를 클래스로 표현하기 위한 방법들이 연구되고 있다. 이렇게 XML 문서 구조를 클래스로 표현하는 것을 XML 바인딩(binding)이라고 한다. 대표적인 것으로는 JAXB[15]가 있으며, JAXB는 XML Schema[16]를 이용해서 XML 원소들을 자바 클래스로 변환한다. JAXB는 XML 문서로 표현된 데이타를 단순히 자바의 엔티티(entity) 클래스로 표현하는 것이고, 객체로 표현된 데이타들이 어떻게 처리되어야하는지는 기술하지 못한다. 또한 클래스를 표현할 때도 상속 관계는 표현하지 못하고, 결합 관계만 표현할 수 있다는 단점을 가지고 있다. 따라서 JAXB는 객체지향 프로그래밍의 장점의 하나인 상속을 통한 재사용은 활용할 수 없다.

객체지향 속성 문법은 문법 구조에 따라 상속과 결합(composition)의 관계를 갖는 클래스로 표현하고, 의미 속성과 의미 속성 규칙들을 클래스의 멤버 필드와 메소

드로 추가할 수 있는 기능을 가지고 있다. 따라서 객체지향 속성 문법을 사용하는 경우에 XML 문서를 자동적으로 처리할 수 있는 문서 처리기를 자동적으로 생성할 수 있는 장점을 가지고 있다.

3. XML의 의미 처리

3.1 XML 문서 처리

XML은 DTD를 이용해서 문서의 논리적인 구조와 어휘들을 정의한다. 이것은 프로그래밍 언어가 BNF를 이용해서 정의되는 것과 유사하다. 일반적으로 프로그래밍 언어는 컴파일러에 의해서 파싱 과정에서 문법적인 오류를 체크하고, 의미 처리 과정에서 입력 프로그램의 의미에 해당되는 내용을 수행한 다음에 결과를 출력하게 된다. 이러한 처리 과정은 그림 1과 같이 표현될 수 있다.

그림 1 프로그램 처리 과정

프로그램이 컴파일러에서 처리되는 것과 마찬가지로 XML 문서도 문서 처리기에 의해서 처리된다. 문서 처리기에서 XML 문서의 파싱은 XML 파서에 의해서 이루어지고, 파서는 문서의 문법적인 오류를 파악하게 된다. XML 파서에 의해서 올바르다고 파악된 XML 문서의 내용은 의미 처리 과정에서 문서의 의미에 맞게 처리된다. 이러한 처리 과정은 그림 2와 같이 표현될 수 있다.

그림 2 XML 문서 처리 과정

프로그래밍 언어에서 문법은 프로그램의 구조적인 특성만 기술하고, 의미적인 사항은 속성 문법을 사용해서 기술하게 된다. 유사하게 DTD는 문서의 구조적인 관계만 정의하고, 문서의 의미는 정의하지 않는다. XML 문서 처리기도 프로그래밍 언어 처리기처럼 자동적으로 생성되기 위해서는 XML 문서 구조에 의미 정보를 기술할 수 있는 방법이 필요하다. XML 문서에서 의미 정보가 필요한 경우를 예를 들어 보기 위해서 사칙 연산을 표현하는 XML 문서가 있다고 가정해보자. XML에서 사칙 연산을 표현하기 위한 수식의 문법은 표 1의 DTD와 같이 기술할 수 있다. exp 원소는 add, sub, mu, div, v 자식 원소들 중에서 하나를 사용할 수 있고, 덧셈 연산을 위한 add 원소는 두개의 exp 원소로 구성

표 1 expression.dtd

```
<!ELEMENT exp (add|sub|mul|div|v)>
<!ELEMENT add (exp, exp)>
<!ELEMENT sub (exp, exp)>
<!ELEMENT mul (exp, exp)>
<!ELEMENT div (exp, exp)>
<!ELEMENT v (#PCDATA)>
```

표 2 exp.xml

```
<?xml version="1.0" encoding="euc-kr"?>
<!DOCTYPE exp SYSTEM "expression.dtd">
<exp>
    <mul>
        <exp>
            <add>
                <exp><v>3</v></exp>
                <exp><v>4</v></exp>
            </add>
        </exp>
        <exp><v>5</v></exp>
    </mul>
</expr>
```

된다. v 원소는 숫자 상수를 표현하기 위해서 사용된다.

수식을 위한 expression.dtd가 주어지면, 다양한 형태의 4칙 연산을 XML을 이용해서 표현할 수 있다. 예를 들어, (3 + 4) * 5를 표현하기 위한 XML 문서 exp.xml은 표 2와 같이 작성할 수 있다.

프로그램 코드가 파싱 과정을 통해서 파스 트리로 표현되는 것과 유사하게 XML 문서는 XML 파서를 이용해서 파싱하는 경우에 DOM 트리로 표현될 수 있다. DOM 트리는 XML문서의 원소, 속성, 텍스트 등을 노드로 표현한 트리이다. DOM 트리에서 원소는 Element라는 클래스로 표현되고, 속성은 Attr 클래스, 텍스트는 TEXT 클래스로 표현된다. exp.xml 문서는 XML 파서를 통해서 DOM 트리로 표현되는 경우에 그림 3과 같은 형태로 표현될 수 있다. DOM 트리는 문서 구조에 관한 정보를 가지고 있다는 점에서 프로그래밍 언어의 파스 트리와 유사하다.

수식을 위한 DTD인 expression.dtd와 exp.xml 문서는 의미 정보를 전혀 가지고 있지 않기 때문에 XML 파서는 exp.xml 문서의 문법적인 사항을 체크하고, DOM 트리를 생성하지만 더 이상의 작업은 처리할 수 없다.

3.2 XML에 객체지향 속성 문법의 적용

객체지향 속성 문법은 속성 문법에 객체지향 개념을 추가한 것이다[10,11]. 객체지향 속성 문법을 사용하는

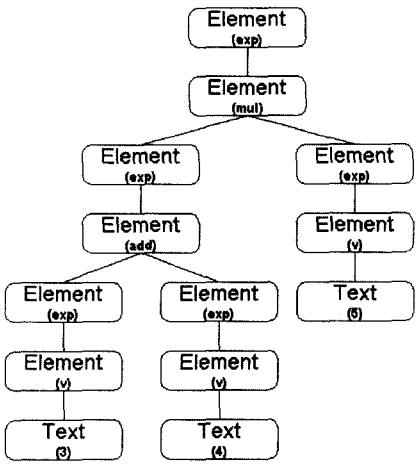

그림 3 exp.xml의 DOM 트리

경우에 언어의 문법에서 넌터미널(nonterminal)은 클래스로 표현된다. 따라서 언어의 정적인 명세는 클래스들의 상속과 결합 관계로 표현되고, 입력 프로그램을 파싱한 결과인 파스 트리에서 각 노드는 클래스들의 인스턴스들로 표현된다. 이때 각 노드 객체들은 의미 속성들을 가질 수 있기 때문에 객체지향 속성 문법에서 클래스의 인스턴스는 문법적인 특성과 의미적인 특성을 동시에 갖는다. 객체지향 속성 문법은 제한된 형태의 문맥 자유 문법(CFG, Context Free Grammar)에만 적용될 수 있지만, 일반 CFG를 객체지향 속성 문법이 적용될 수 있는 형태로 변환할 수 있다[11].

DTD는 확장된 문맥 자유 문법(ECFG, Extended Context Free Grammar)을 사용해서 XML 문서의 구조를 정의하고, ECFG는 CFG로 변환될 수 있다. 따라서 XML 문서에 객체지향 속성 문법을 적용할 수 있으며, 객체지향 속성 문법을 따르는 경우에 XML의 각 원소는 클래스로 표현될 수 있다. XML에 객체지향 속성 문법을 적용하기 위해서는 DTD를 적절한 형태로 변환하여야 한다. DTD를 변환하기 위해서 DTD의 구성원들을 사용되는 목적에 따라서 몇 개의 집합으로 분류할 수 있다.

정의 1. DTD 구성원들의 집합

E는 DTD에서 정의된 모든 원소들의 집합이고, Att(e)는 원소 e의 모든 속성들 집합이다. S는 원소의 옵션 혹은 반복 등을 기술하기 위해 사용되는 구성물들로서, S = {'?', '*', '+'}로 정의한다. □

DTD에서 원소는 OR(|) 혹은 SEQ(,)를 이용해서 자식 원소들을 기술할 수 있다. 원소가 자식 원소들을 OR 관계를 이용해서 기술하고, 원소가 속성을 갖지 않는 경우에 이 원소를 추상 원소(abstract element)라고 한다. 객체지향 속성 문법을 이용하면, 추상 원소와 자식 원소들은 상속 관계를 이용해서 표현할 수 있다.

정의 2. 추상 원소

원소 A는 다음과 같은 형태로 정의되는 경우에 추상 원소라고 한다.

<!ELEMENT A (B$_1$O$_1$ | B$_2$O$_2$ | ... | B$_n$O$_n$)>

단, $1 \le i \le n$일 때 $B_i \in E$이고, $O_i \in \{\varepsilon, '?'\}$이며, $Att(A) = \varnothing$이다. □

추상 원소 A와 자식 원소들은 상속 관계로 표현되고, 다음과 같은 UML로 표현될 수 있다.

원소 A가 하나의 자식 원소를 갖는 경우에도 원소 A는 추상 원소이다. 따라서 <!ELEMENT A (B$_1$)>인 원소 정의가 있는 경우에 B$_1$은 A로부터 상속받는 서브 클래스로 표현될 수 있다.

어떤 원소는 연속적으로 기술된 자식 원소들을 갖는다. 이렇게 자식 원소들을 SEQ를 이용해서 기술하는 경우에 이 원소를 구조적 원소(structured element)라고 한다.

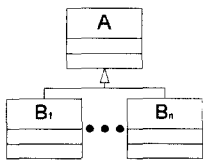

그림 4 추상 원소의 클래스 관계

정의 3. 구조적 원소

원소 A는 다음과 같은 형태로 정의되는 경우에 구조적 원소라고 한다.

<!ELEMENT A (B$_1$O$_1$, B$_2$O$_2$, ..., B$_n$O$_n$)>

단, $1 \le i \le n$일 때 $B_i \in E$, $O_i \in S \cup \{\varepsilon\}$이다.□

구조적 원소는 객체지향 속성 문법을 사용하는 경우에 자식 원소들과 결합 관계로 연결된다. 이때 자식 원소들을 컴포넌트(component) 클래스라고 한다. 구조적 원소는 그림 5와 같은 클래스 관계를 갖는다.

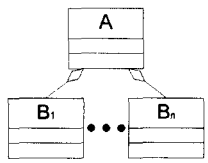

그림 5 구조적 원소의 클래스 관계

정의 4. 반복 연산자

원소 정의에서 자식 원소의 반복 형태를 지정하기 위한 기호인 '*'와 '+'를 반복 연산자라고 한다. □

단일 자식 원소를 갖는 원소 A가 반복 연산자를 이용해서 <!ELEMENT A (B_1)X>, X ∈ {'*', '+'} 형태로 정의된 경우에 원소 A는 B_1이라는 자식 원소를 0개 이상(혹은 1개 이상) 갖기 때문에 구조적 원소이며, 그림 6과 같은 형태의 클래스 관계를 갖는다. 만약 원소 A가 '+' 반복 연산자를 사용하는 경우에는 그림 6에서 B_1은 1..n개의 다중성(multiplicity)을 갖는다.

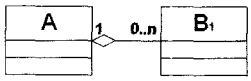

그림 6 반복 연산자를 갖는 경우 클래스 관계

원소가 자식 원소들을 OR 관계를 이용해서 기술하고, 반복 연산자를 사용하는 경우에는 새로운 원소를 도입해서 DTD를 변경하여야 한다. 반복은 여러 개의 내용을 갖는다는 의미이기 때문에 자식 원소들을 결합(composition) 관계를 통해서 표현하고, OR는 자식 원소들을 부모로부터 상속받는 클래스로 표현된다. 이러한 경우에 결합과 상속은 동시에 표현할 수 없기 때문에 새로운 원소를 도입해야 한다.

정리 1. 반복 연산자가 사용된 OR 관계

자식 원소가 반복이 있는 OR 관계로 표현되는 경우에는 OR 관계 부분을 새로운 원소를 도입해서 치환한다. □

예를 들어, <!ELEMENT A (B_1 | B_2 | ... | B_n)*> 형태의 원소 A를 정의하는 경우에 OR 관계로 묶인 자식 원소들을 위해 새로운 원소 B를 도입해서 다음과 같이 변경할 수 있다.

<!ELEMENT A (B)*>
<!ELEMENT B (B_1 | B_2 | ... | B_n)>

어떤 경우에는 원소의 자식 노드가 OR와 SEQ 관계를 모두 갖는 경우도 있다. 이러한 원소는 복합 원소(compound element)라고 한다.

정의 5. 복합 원소

원소 A가 다음과 같은 형태로 정의되는 경우에 A를 복합 원소라고 한다.

<!ELEMENT A (δ)>

δ는 문법 G에 의해서 생성되는 언어 L(G)의 문자열이고, 문법 G는 G = ({K, O, R}, E ∪ {'(', ')', '|', ','}, P, K)로 정의된다. P는 다음과 같은 생성 규칙들을 갖는다.

K -> K O K | '('K')' R | e R (단, e ∈ E)
O -> '|' | ','
R -> ε | '?' | '*' | '+'

문자열 δ는 '|' ∈ δ이고, ',' ∈ δ이거나, '(' ∈ δ이고, ')' ∈ δ이다. □

객체지향 속성 문법은 결합과 상속을 동시에 지원하지 못하기 때문에 XML에서 복합 원소는 새로운 원소를 도입해서 추상 원소와 구조적 원소의 형태로 변경하여야 한다. 예를 들어, <!ELEMENT A ((B_1, B_2) | D | F)>는 다음과 같이 변경될 수 있다.

<!ELEMENT A (B | D | F)>
<!ELEMENT B (B_1, B_2)>

XML에서 원소의 속성은 원소의 메타 정보들을 표현하기 위해서 주로 사용된다. 속성은 자식 원소로 변경해서 기술할 수 있으며, 어떤 정보를 속성으로 기술할 것인지 혹은 자식 원소로 기술할 것인지는 DTD 설계자의 취향에 따라 결정될 수 있다. 따라서 객체지향 속성 문법을 사용하는 경우에 원소의 속성은 원소의 자식 원소와 동일하게 취급할 수 있다.

정리 2. 속성

원소의 속성은 원소의 자식 원소로 취급하고, 자식 원소들과 SEQ 관계를 갖는다. □

<!ELEMENT X (B_1 | B_2 | ... | B_n)>이고, Att(X) = {a_1, a_2, ..., a_m}인 경우에 원소 X는 다음과 같은 복합 원소로 변경할 수 있다.

<!ELEMENT X ((a_1, a_2, ..., a_m), (B_1 | B_2 | ... | B_n))>

객체지향 속성 문법의 정의를 따르는 경우에 expression.dtd의 원소 선언들은 객체지향 문맥 자유 문법에 의해서 그림 7과 같은 클래스 관계를 갖는다. 그림 7에서 원소들을 클래스로 표현하기 위해서 원소 이름의 첫 글자를 대문자로 기술하였다. 그림 7에서 볼 수 있듯이 Add, Sub, Mul, Div, V는 모두 Exp라는 클래스로부터 상속받으며, V를 제외한 원소들은 컴포넌트 클래스로 Exp를 포함한다.

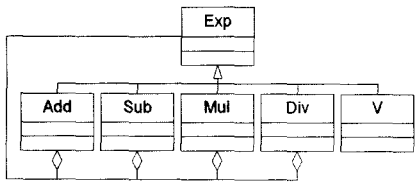

그림 7 expression.dtd의 객체지향 표현

컴파일러에서 파스 트리는 파싱 과정의 결과로 나타나지만, 의미 분석과 코드 생성을 위해서 사용되기에는 부적합하기 때문에 파스 트리는 AST(Abstract Syntax Tree)로 변환해서 사용하게 된다. XML 문서에서도 DOM 트리는 많은 정보를 가지고 있지만, 문서를 효과적으로 처리하기 위해서는 보다 간략한 형태로 표현하여야 한다. 본 논문에서는 XML 객체 트리를 사용한다.

정의 6. XML 객체 트리

XML 객체 트리는 XML 문서의 인스턴스를 표현하는 트리이고, 다음과 같이 정의된다.

1. 루트 노드는 XML 문서의 문서 타입(document type) 클래스 혹은 문서 타입 클래스로부터 상속받는 클래스의 인스턴스이다.
2. 원소 A가 추상 원소이고, A_1이 A로부터 상속받는 구조적 원소이고, XML 문서에서 A 원소 내용으로 A_1 원소가 사용될 때, A 원소 내용은 A_1 클래스의 인스턴스로 표현된다.
3. 노드 X가 b_1, b_2, ..., b_n인 자식 노드를 갖고, b_1, b_2, .., b_n이 각각 B_1, B_2, ..., B_n 클래스의 인스턴스일 때 <!ELEMENT X (B_1, B_2, ..., B_n)> 혹은 <!ELEMENT X (B_{i+1}, ..., B_n)>, Att(X) = { b_1:B_1, b_2:B_2, ..., b_i:B_i}인 원소 선언이 존재한다.
4. 단말 노드 t는 #PCDATA, EMPTY로 선언된 원소 혹은 속성이다.

표 2의 exp.xml 문서는 XML 객체 트리를 이용해서 표현하는 경우에 다음 그림 8과 같은 형태의 트리로 나타낼 수 있다. exp.xml 문서에서 exp 원소는 추상 원소이기 때문에 exp 원소의 내용은 exp로부터 상속받는 클래스들이 대신 사용된다.

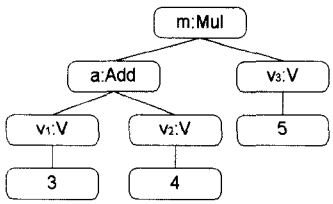

그림 8 exp.xml 문서의 XML 객체 트리

객체지향 속성 문법을 사용하는 경우에 문서의 의미 정보들을 클래스의 멤버 필드로 추가할 수 있다. 예를 들어, expression.dtd에서 exp 원소에 value라는 의미 속성을 추가하기 위해서는 Exp 클래스에 멤버 필드를 추가하는 것으로 표현할 수 있다. 따라서 Exp 클래스로부터 상속받는 다른 원소들은 자동적으로 value라는 의미 속성을 갖게 된다. 만약 Exp 클래스에서 의미 속성을 평가하기 위해서는 evaluate() 메소드를 정의하고, Add 클래스에서 value 속성 값을 다른 방식으로 계산해야 한다면, Add 클래스는 evaluate() 메소드를 재정의할 수 있다. 그림 9는 의미 속성을 클래스의 멤버 필드로 추가한 클래스들의 모습이다.

XML 객체 트리를 사용하는 경우에 그림 3의 DOM 트리는 그림 10과 같은 보다 간단한 형태의 내부 구조로 표현될 수 있다. 그림에서 각 노드는 둥근 사각형으로 표현되고, 점선을 이용해서 내부 구조를 보여준다. 예를 들어, Mul 클래스는 Exp 클래스로부터 상속받기 때문에 Mul 클래스 타입의 인스턴스인 m은 내부에 Exp 클래스에 대한 내용을 포함하고 있다. 그림에서 m:Mul 노드의 점선으로 그려진 상단은 부모 클래스인 Exp 클래스의 내용을 의미한다. Exp 클래스는 의미 속성을 위해 value라는 멤버 필드와 evaluate() 메소드를 갖고 있고, Mul 클래스는 Exp 클래스로부터 상속받기 때문에 value 멤버 필드를 가지고 있으며, 곱셈에 대해 value 멤버 필드 값을 계산하기 위한 evaluate() 메소드를 재정의한다. 그림 10에서 루트 노드의 Expr과 Mul 부분은 이러한 내용을 표시한다. Exp의 .value는 Exp 클래스의 멤버 필드 value를 의미하고, Mul 부분의 "= $1.value * $2.value"는 Mul에서 재정의한 evaluate() 메소드 내용을 의미한다.

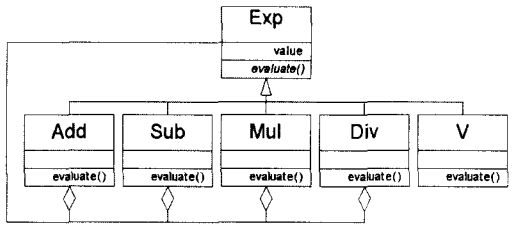

그림 9 의미 속성이 추가된 클래스

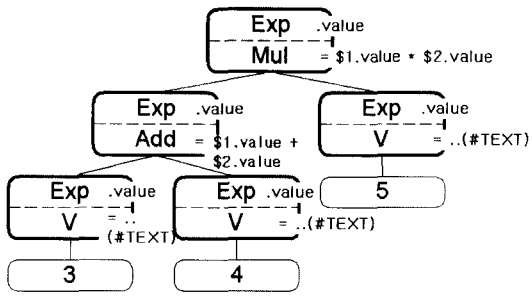

그림 10 exp.xml을 위한 객체지향 속성 트리

3.3 의미 기술

XML 문서를 자동적으로 처리하기 위해서는 XML 문서의 구조를 위한 DTD 이외에 문서 처리를 위한 의미 규칙을 기술할 수 있는 방법이 필요하다. YACC[12]과 같은 컴파일러 생성 도구는 문법과 의미 정보를 같이 기술한다. 그러나 본 논문에서는 XML 문서의 의미 정보를 별도의 XML 파일에 기술하는 방법을 사용한다. 이것은 여러 가지 장점을 가지고 있다. 첫 번째로 XML 문서의 문법은 DTD라는 표준화된 방법을 사용하기 때문에 DTD에 의미 정보를 추가하는 것은 표준에 어긋나는 문제점이 있다. 두 번째는 XML 문서가 여러 가지

의미를 가질 수 있다는 점이다. 컴파일러에서는 프로그래밍 언어의 문법이 결정되면 프로그램 실행이라는 오직 하나의 의미를 갖게 된다. 그러나 XML 문서는 처리되는 방식에 따라 많은 의미들이 존재할 수 있다. 예를 들어, 앞에서 다룬 exp.xml 문서는 실행될 수 있으며, 어떤 경우에는 데이타베이스에 저장될 수도 있다. 혹은 프로그래밍 언어의 수식으로 변환될 수도 있다. 이처럼 XML 문서는 사용되는 목적에 따라 의미 정보가 달라질 수 있다. 문법과 속성 평가 규칙을 분리하는 경우에는 DTD는 변경하지 않고, 의미 정보만 새로 작성할 수 있는 장점이 있다.

의미 정보를 정의하는 XML 문서의 구조는 표 3과 같은 형태로 구성된다. 문서의 가장 상위에는 semantics 원소가 있고, semantics는 code, rules, util이라는 자식 원소를 포함한다.

표 3 의미 속성을 기술하기 위한 문서의 DTD

```
<!ELEMENT semantics (code?, rules, util?)>
<!ELEMENT code    (#PCDATA)>
<!ELEMENT rules   (tag)+ >
<!ELEMENT tag     (node|rule)+ >
<!ATTLIST  tag
           name   NMTOKEN #REQUIRED>
<!ELEMENT node    (attribute)+ >
<!ELEMENT attribute EMPTY>
<!ATTLIST  attribute
           name   NMTOKEN #REQUIRED
           type   NMTOKEN #REQUIRED>
<!ELEMENT rule    (#PCDATA)>
<!ATTLIST  rule
           name   NMTOKEN #REQUIRED>
<!ELEMENT util    (#PCDATA)>
```

code 원소에는 생성되는 XML 문서 처리기에서 필요로 하는 라이브러리의 사용여부에 관련된 사항들을 기술한다. 예를 들어, 생성되는 프로그램들이 소속되는 패키지에 관련된 정보를 기술하거나, 필요로 하는 라이브러리를 임포트하기 위한 내용들을 기술한다. code 원소의 내용은 생성되는 클래스들의 소스에 그대로 복사된다.

rules 원소에는 문서에서 필요로 하는 의미 속성들과 의미 속성들을 계산하기 위한 규칙들을 기술하게 된다. rules는 여러 개의 tag 원소들로 구성된다. tag에는 의미 속성이나 속성을 평가하기 위한 속성 평가 규칙들을 기술하게 된다. tag는 의미 속성이 적용되기 위한 XML 문서의 원소 이름을 기술하기 위해서 name 속성을 갖는다. tag의 node 자식 노드는 의미 속성을 기술하기 위해서 사용된다. attribute의 name 속성은 의미 속성의 이름을 기술하고, type은 의미 속성의 자료형을 기술한다. rule 원소는 의미 속성의 이름을 기술하기 위한 name 속성을 가지고 있고, 의미 속성을 계산하기 위한 내용을 PCDATA 형식으로 기술한다. tag 원소에서 node 자식 원소는 한번만 나타날 수 있고, rule은 여러 번 사용될 수 있다.

util 원소에는 속성을 계산하기 위해서 필요로 하는 여러 가지 형태의 유틸리티용 함수들을 정의하기 위해서 사용된다. util 원소에 있는 내용은 생성되는 클래스에 그대로 복사된다.

표 1과 표 2의 사칙 연산 수식을 위한 의미 XML 문서는 표 4와 같이 작성할 수 있다. code 원소에는 필요한 라이브러리를 임포트하기 위한 import 문장을 기술한다. rules 원소에는 의미 속성들과 의미 속성을 계산하기 위한 내용들을 기술한다. name 속성 값이 exp인 tag에는 Number 타입의 value 의미 속성을 정의한다. name 속성 값이 add인 tag는 exp 원소로부터 value 의미 속성을 상속받고, evaluate() 메소드를 재정의해야 하기 때문에 rule 부분만 기술한다.

표 4 수식을 위한 의미

```
<semantics>
    <code>
        import java.lang.*; ...
    <rules>
        <tag name='exp'>
            <node>
                <attribute name='value' type='Number'/>
            </node>
        </tag>
        <tag name='add'>
            <rule name='value'>
            $$ = NumberUtil.add($1.value, $2.value);
            </rule>
        </tag>
        <tag name='v'>
            <rule name='value'>
            $$ = NumberUtil.s2n(#TEXT);
...
```

의미 규칙에서 $문자와 같이 사용되는 숫자는 객체지향 파스 트리에서 현재 노드의 자식 노드의 인덱스 번호를 의미한다. 예를 들어, $1.value는 첫 번째 자식 노드의 value 속성 값을 의미한다. $$는 rule의 name에 기술된 의미 속성을 의미하고, #TEXT는 원소의 PCDATA 타입의 텍스트를 의미한다. XML 원소의 속성은 @ 문자와 속성 이름을 이용해서 접근한다.

4. 시스템 구성

의미 규칙을 사용하는 시스템은 그림 11과 같은 형태로 구성된다. 문서의 DTD와 의미 정보를 갖고 있는

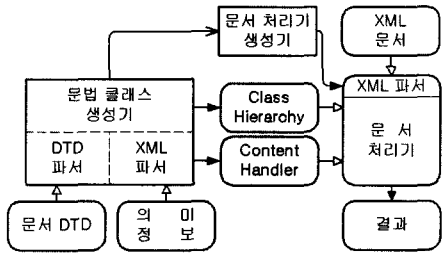

그림 11 시스템 구성과 XML 문서 처리 과정

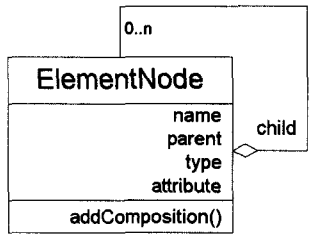

그림 12 ElementNode 클래스

XML 파일은 문법 클래스 생성기의 입력으로 사용된다.

문법 클래스 생성기는 DTD 파서를 통해서 XML 문서에서 원소들 간의 관계를 파악한다. DTD 파서는 문서 DTD를 입력으로 받고, DTD를 파싱해서 DTD 내용들을 그래프 형태로 표현해서 리턴하게 된다. DTD그래프에서 원소를 표현하기 위한 노드는 ElementNode 클래스를 이용해서 표현된다. ElementNode는 그림 12와 같은 정보들을 가지고 있다. parent는 원소의 부모 클래스를 표현하기 위해서 사용된다. type 멤버 필드는 원소의 구조가 어떤 형태인지를 알려준다. child 멤버 필드는 원소의 자식 원소들에 대한 정보를 가지고 있고, attribute는 원소의 속성들을 표현한다. addComposition (ElementNode e) 메소드는 원소 e를 결합 관계로 묶는 것을 의미한다.

DTD 그래프는 알고리즘 1을 통해서 객체지향 속성 문법의 규칙에 맞게 클래스들의 상속과 결합 관계를 이용해서 표현한다. 알고리즘 1은 DTD에서 문서 타입 원소를 시작으로 문서 전체 구조를 클래스의 상속과 결합 관계를 표현하는 그래프를 생성하게 된다.

문법 클래스 생성기는 XML 파서를 이용해서 의미 정보를 읽고, 각 원소별로 속성과 의미 정보를 추출해서 해시 테이블로 관리한다. 이때 의미 정보는 그림 13의 SemanticNode 클래스로 표현된다. SemanticNode는 의미 속성을 위한 GAttribute 클래스로 구성되어 있으며, GAttribute는 의미 속성의 이름(name 멤버 필드)과 의미 속성을 평가하기 위한 코드(rule 멤버 필드)를 갖는다.

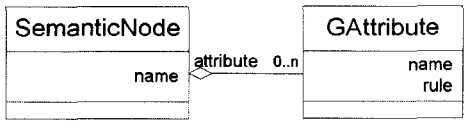

그림 13 SemanticNode 클래스

알고리즘 1. 클래스_관계_만들기

```
매개 변수: ElementNode e
begin
    if (e.attribute.length == 0) then
        if (e.type == 추상원소) then
            foreach c in e.child
                c.parent = e
                클래스_관계_만들기(c)
            end-foreach
        else if (e.type == 구조원소) then
            foreach c in e.child
                e.addComposition(c)
                클래스_관계_만들기(c)
            end-foreach
        else
            error("에러: 복합 원소를 추상 원소와 구조
            원소로 변경해야함.")
        end-if
    else
        error("에러: 속성을 원소로 변경해야함.")
    end-if
end
```

알고리즘 2. 클래스_생성하기

```
매개변수: ElementNode e
매개변수: SemanticNodeHash s
매개변수: 원소_그래프 G
begin
    if (e.parent != NULL) then
        클래스 이름이 e.name이고, 부모 클래스 이름이
        e.parent인 클래스를 작성한다.
    else
        클래스 이름이 e.name인 클래스를 작성한다.
    end-if
    SemanticNode node = s.find(e.name)
    foreach a in node.attribute
        클래스의 멤버 필드로 a.name을 추가한다.
        클래스의 메소드로 a.rule의 내용을 추가한다.
    end-foreach
    if (e.child != NULL) then
        foreach c in e.child
            클래스의 멤버 필드로 c를 추가한다.
            클래스_생성하기(c)
        end-foreach
    end-if
    foreach c in G
        if (e.name == c.parent) then
            클래스_생성하기(c)
        end-if
    end-foreach
end
```

문법 클래스 생성기는 알고리즘 1에서 만들어진 클래스 그래프와 SemanticNode로 구성된 해시 테이블을 이용해서 알고리즘 2에 따라서 자바 클래스 코드를 생성한다.

문법 클래스 생성기는 DTD 원소들을 위한 클래스를 생성하는 동시에 의미 정보들을 효과적으로 처리하기 위해서 visitor 패턴에 따라 순회하기 위해서 필요한 의미 처리기 클래스를 생성한다. 의미 처리기는 비지터 패턴(visitor pattern)[17]을 이용해서 XML 객체 트리를 순회하면서 XML의 내용을 의미에 맞게 처리한다.

문서 처리기 생성기는 문법 클래스 생성기에서 의미 정보를 전달받고, 문서 처리기와 SAX 컨텐트 핸들러를 생성한다. 문서 처리기는 XML 문서를 입력받아서 유효성을 체크하고, 컨텐트 핸들러 클래스를 이용해서 XML 객체 트리를 생성한다. 생성된 XML 객체 트리는 의미 처리기에 의해서 의미에 맞게 처리될 수 있다.

표 1의 DTD 정보와 표 4의 의미 정보를 이용하는 경우에 문법 클래스 생성기는 add 원소를 위해 표 5와 같은 Add 클래스를 생성한다. Add 클래스는 Exp 클래스로부터 상속받고, Exp 클래스 타입의 _exp0와 _exp1이라는 2개의 컴포넌트 클래스를 갖는다. 또한 의미 속성을 평가하기 위해서 evaluate() 메소드를 갖는다.

표 5 생성된 Add 클래스

```
public class Add extends Exp implements IVisitable {
    protected Exp  _exp0;
    protected Exp  _exp1;
    protected IVisitor visitor;
.....
    public Object evaluate(IVisitor visitor) {
        this.visitor = visitor;
        return this.visitor.visit(this);
.....
```

XML 문서의 의미를 처리하기 위한 의미 처리기는 비지터 패턴 형태로 구현된다. 수식의 XML 문서를 처리하기 위한 의미 처리기는 SemVisitor 클래스이고, SemVisitor 클래스는 표 6과 같은 형태로 문법 클래스 생성기를 통해서 생성된다. SemVisitor의 visit() 메소드는 XML 객체 트리에서 각 노드의 타입에 따라 적절한 메소드를 호출하게 된다. 예를 들어, add 원소가 사용된 경우에는 visitAdd() 메소드를 호출한다. visitAdd() 메소드는 자신의 컴포넌트 클래스로 갖는 _exp0와 _exp1의 value 의미 속성 값을 얻고, 2개의 값을 더해서 리턴한다.

문서 처리기 생성기는 표 7과 같은 문서 처리기를 자동적으로 생성한다. 문서 처리기는 SAX 파서를 생성해서 XML 문서의 유효성을 파악하고, SAX의 컨텐트 핸들러 클래스를 이용해서 XML 문서를 처리하면서 XML 객체 트리를 생성한다. 생성된 XML 객체 트리는 의미 처리기인 SemVisitor 클래스의 accept() 메소드의 매개 변수로 전달된다. SemVisitor는 트리를 순회하면서 XML 문서의 의미를 처리하고, 결과를 Object 타입으로 리턴한다.

표 6 생성된 SemVisitor 클래스

```
public class SemVisitor implements IVisitor {
    protected Object visitAdd(Add e ) {
        Object o1 = e.get_exp0();
        Object oa1 = visit((IVisitable)o1);
        Attribute _a1 = (Attribute) oa1;
.....
        value = NumberUtil.add(_a1.value, _a2.value);
        return new Attribute(value);
    }
.....
    public Object visit(IVisitable e) {
        if(e instanceof Add) {
            return visitAdd((Add) e);
        } else if(e instanceof Sub) {
.....
```

표 7 생성된 문서 처리기 클래스

```
.....
public class DocProcessor {
    public static void main(String args[]) {
        try {
            Handler ha = new Handler();
            XMLReader parser = new SAXParser();
.....
            parser.setContentHandler(ha);
            parser.parse(args[0]);
            IVisitable root = ha.getRoot();
            SemVisitor visitor = new SemVisitor();
            Object o = root.accept(visitor);
.....
```

문서 처리기 생성기는 문서 처리기와 함께 SAX의 컨텐트 핸들러 클래스를 생성한다. 표 8은 생성된 컨텐트 핸들러 클래스의 예이다. 컨텐트 핸들러는 SAX 파서가 XML 문서를 처리하면서 발생하는 이벤트에 따라 XML 객체 트리의 노드를 위한 객체들을 생성하고, 트리를 생성한다.

표 8 SAX 컨텐트 핸들러 클래스

```
.....
public class Handler extends DefaultHandler {
.....
    public void startElement(String uri, ...) throws
    SAXException {
        try {
            if(localName.equals("add")) {
                _o = Class.forName("Add").newInstance();
                stack.push(_o);
            }
.....
```

5. 결 론

XML의 DTD는 문서의 문법적인 규칙만 정의하고, 의미는 기술하지 않는다. 따라서 XML 문서를 올바르게 처리하기 위해서는 SAX, DOM과 같은 API를 이용해서 문서 처리기 프로그램을 작성해야 하는 어려움이 있다. API를 이용한 문서 처리기의 작성은 많은 시간과 노력을 필요로 한다. 이러한 문제를 해결하기 위해서 속성 문법을 사용해서 문서 처리를 자동적으로 이루어지기 위한 연구들이 있었지만, 이러한 연구에서 개발된 시스템들은 주로 C 언어를 기반으로 기본적인 속성 문법만을 사용하도록 작성되었기 때문에 현재의 객체지향 언어에서 사용하기 부적합하다. 또한 클래스 개념을 제공하지 못하고, 객체지향 프로그래밍의 장점인 재사용성의 장점을 얻을 수 없었다.

본 논문에서는 객체지향 속성 문법을 이용해서 XML 문서의 의미를 기술하는 방법과 문서 처리기 생성기를 이용해서 문서 처리기를 자동적으로 생성할 수 있는 방법을 소개하였다. 객체지향 속성 문법을 적용하는 경우에 XML의 원소는 클래스로 표현되고, 의미 속성은 클래스의 멤버 필드로 표현된다. 객체지향 속성 문법은 객체지향 프로그래밍에 적합하고, 배우고, 사용하기 쉬운 장점을 가지고 있다. XML 문서의 의미 정보는 별도의 XML 문서에 표현되며, 의미 정보는 XML 문서를 의미에 맞게 처리할 수 있는 문서 처리기를 자동적으로 생성할 수 있도록 한다. 본 논문에서 제안하는 문서 처리기 생성기는 XML 문서를 처리하는데 드는 많은 시간과 노력을 감소시켜줄 것이다. 객체지향 속성 문법의 단점은 원소의 형태를 객체지향 속성 문법이 적용될 수 있는 형태로 변경되어야 한다는 점이다. 즉, 직관적으로 만들어진 XML의 DTD를 객체지향 속성 문법을 적용할 수 있는 형태로 변경해야 한다. 이러한 단점을 해결하기 위한 방법은 추후에 연구될 것이다.

참 고 문 헌

[1] Extensible Markup Language (XML), http://www.w3.org/XML/.

[2] Vector Markup Language (VML), http://www.w3.org/TR/NOTE-VML.

[3] Scalable Vector Graphics (SVG), http://www.w3.org/Graphics/SVG/.

[4] Resource Description Framework (RDF), http://www.w3.org/RDF/.

[5] XML Protocol Activity, http://www.w3.org/2000/xp/.

[6] Greg K. Badros, "JavaML: A Markup Language for Java Source Code," In *9th WWW Conf.*, 2000, available at http://www.cs.washington.edu/homes/gjb/JavaML/.

[7] The Simple API for XML (SAX), http://www.saxproject.org/.

[8] Document Object Model (DOM), http://www.w3.org/DOM/.

[9] Donald E. Knuth, "Semantics of context-free languages," In *Math. Systems Theory*, pp. 127~145, 1968.

[10] Jukka Paakki, "Attribute Grammar Paradigms -- A High-level Methodology in Language Implementation," In *ACM Computing Surveys*, Vol. 27, No. 2, pp. 197~255, 1995.

[11] Kai Koskimies, "Object-Orientation in Attribute Grammars," In *Attribute Grammars, Applications and Systems*, LNCS 545, Springer-Verlag, pp. 297~329, 1991.

[12] Stephen C. Johnson, "Yacc: Yet Another Compiler-Compiler", In *Computing Science Technical Report 32*, AT&T Bell Labs, Murray Hill (NJ), 1975.

[13] Jos Warmer and Hans Van Vliet, Processing SGML Documents, In *Electronic Publishing*, Vol. 4, No. 1, pp. 3~26, 1991.

[14] Giuseppe Psaila, Stefano Crespi-Reghizzi, "Adding Semantics to XML," In *WAGA99*, 1999, available at www-rocq.inria.fr/oscar/waga99.html.

[15] Sun, "Java Architecture for XML Binding", available at http://java.sun.com/xml/jaxb/.

[16] XML Schema, http://www.w3.org/XML/Schema.

[17] Erich Gamma, et al, *Design Patterns*, Addison-Wesley Pub., 1995.

최 종 명

1992년 숭실대학교 전자계산학과 학사 1996년 숭실대학교 전자계산학과 석사 2003년 숭실대학교 컴퓨터학과 박사. 관심분야는 시각 프로그래밍, 멀티패러다임 시스템, XML, 유비쿼터스 컴퓨팅

유 재 우

1976년 학사, 숭실대학교 전자계산학과 1985년 박사, 한국과학기술원 전산학과 1983년~현재, 숭실대학교 컴퓨터학부 교수. 1986년~1987년, 1996년~1997년 코넬대학교, 피츠버그대학교 객원교수. 1999년~2000년 한국정보과학회 프로그래밍언어 연구회 위원장. 관심분야는 프로그래밍언어, 컴파일러, 인간과 컴퓨터 상호작용