정보학연구 제6권 제4호 2003년 12월(pp.113-123)
The Journal of Information Technology, Vol.6, No.4

비 환형 속성 문법을 이용한 코드 생성 시스템

김상훈*

요 약

본 논문은 사이클이 없는 속성문법을 사용한 코드 생성에 관한 연구이다. 이 연구를 수행하는 목적은 비 순환 속성문법을 기술하고 속성 문법을 표현하는 메타 언어인 속성 기술 언어를 설계하는 것이다. 여기서, 비 환형 속성 문법으로 L-attributed grammar를 사용한다. 본 시스템은 속성 문법으로부터 정보를 얻기 위한 분석기와 속성 평가기로 구성되어 있다. 속성 문법 분석기는 속성 기술 언어를 취하고 속성 평가기에 의해 요구되는 유용한 정보를 생성한다. 속성 평가기는 이 정보를 사용하여 속성 값을 계산한다.

I. 서론

정보화 사회로의 급속한 발전은 여러 응용 소프트웨어의 필요성을 낳았다. 다양한 응용 소프트웨어를 제작하기 위해서는 그의 운영 환경에 따라 적절한 프로그래밍 언어를 사용해야 한다. 또한 하드웨어의 발전에 따라 새로운 기능을 충분히 수용할 수 있도록 컴파일러는 지속적으로 개선되고 유지 보수되어야 한다. 이러한 요구에 따라 신뢰성 있고 효율적으로 프로그래밍을 할 수 있는 프로그래밍 언어의 설계와 구현은 매우 중요한 문제로 등장하였다. 컴파일러-컴파일러의 필요성이 오늘날 더욱 절실히 요구되고 있다. 컴파일러를 구현하는데 있어서 드는 개발비용을 최소화하면서 신뢰성 있는 컴파일러를 만들 수 있는 소프트웨어 도구를 개발하려는 것이다.

컴파일러의 어휘 분석(lexical analysis)과 구문 분석(syntax analysis)을 자동화하려는 노력은 형식 언어 이론의 확립에 힘입어 많은 부분 자동화되어 있다. 예를 들어, 어휘분석기 생성기인 Lex, flex, ... 그리고 구문분석기 생성기인 Yacc, bison, PGS, ...과 같은 자동화 도구는 이미 일반화되어 있고 많은 응용 분야에서 사용되고 있다. 이에 비해 의미 분석(semantic analysis) 단계를 위한 소프트웨어 도구에 대한 연구는 아직 부족하여 또한 많은 부분에서 수동으로 이루어지고 있다.

본 연구에서는 의미 분석 단계를 자동화하려는 노력으로, 프로그래밍 언어의 의미를 정형하게 표현할 수 있는 메타 언어(metalanguage)를 개발하고 이러한 표현으로부터 특정 언어의 코드 생성기를 자동 생성할 수 있는 코드 생성 시스템을 개발하려는 것이 본 연구의 근본 목적이다.

프로그래밍 언어의 의미를 정형하게 기술하기 위해 가장 잘 알려진 속성 문법(attribute grammar)을 기반으로 코드 생성 시스템을 자동 구성하여 보고자 한다. 그러나 속성 문법은 의미 규칙을 평가하기 위해 종속 그래프를 구성하고 위상 정렬(topological sort)에 의해 그의 평가 순서를 결정하게 된다. 그의 평가 순서를 결정하는 단

* 세명대학교 소프트웨어학과 조교수

계에서 속성 문법의 형태에 따라 무한 반복에 빠진다. 무한 반복 문제는 위상 정렬 당시에 발견할 수 있다. 사이클을 내포한 속성 문법을 사용하여 구성된 컴파일러는 원시 프로그램의 번역과정에서 속성 문법의 자료 종속 사이클로 인하여 컴파일이 불가능해지는 문제점을 가진다. 속성 문법의 기술 단계에서 사이클이 내포되지 않는 속성 문법의 기술로 제약을 가함으로서 이를 해결할 수 있다. 이러한 속성 문법으로 본 논문에서는 L-attributed grammar를 사용하고자 한다. 본 논문의 실험을 위해 Pascal의 축소형인 Mini-Pascal을 이용하였으며 목적 코드로는 스택 가상 기계에 근거를 두고 있는 U-code를 사용하였다.

본 논문은 2장에서 속성 문법과 본 시스템을 구축하기 위한 기본 이론의 설명으로 시작한다. 3장에서는 속성 평가기의 구축 방안, 코드 생성을 위한 의미 표현을 위해 고안된 의미 표현 언어(SDL; Semantic Description Language)와 속성 문법으로 기술된 표현을 처리하기 위한 속성 문법 분석기(AGA; Attribute Grammar Analyzer)에 대해 설명한다. 그리고 마지막 4장 평가 및 결론에서는 본 시스템의 출력 결과를 확인하고, 연구로 인하여 기대되는 결과에 대해 알아보도록 하겠다.

Ⅱ. 속성 문법

속성 문법은 문맥 자유 문법(context free grammar)에 프로그래밍 언어의 의미를 정형하게 기술한 방법으로 1986년 D. Knuth에 의해 처음 제안되었다[11].

[정의 2.1] 속성 문법은 문맥 자유 문법 부분과, 상속 속성(inherited attribute)과 합성 속성(synthesized attribute)으로 언어의 의미를 표현한 의미 함수(semantic function) 부분으로 다음과 같이 구성된다[1,12,13].

형식: 생성규칙$_i$: 의미규칙$_1$, ... 의미규칙$_n$
생성규칙$_{i+1}$: 의미규칙$_1$, ... 의미규칙$_n$
...
생성규칙$_k$: 의미규칙$_1$, ... 의미규칙$_n$

속성은 상속 속성과 합성 속성으로 분류된다. 합성 속성은 파스 트리의 단말노드(leaf node)로부터 루트노드(root node)를 향하여 정보가 전달되는 반면 상속 속성은 루트노드로부터 단말노드를 향하여 아래로 정보가 전달된다.

시작 심벌은 부 노드(parent node)를 갖지 않으므로 합성 속성을 갖지 않고 터미널 심벌은 그 심벌이 가지고 있는 정보를 전달할 자 노드(child node)를 갖지 않으므로 상속 속성을 갖지 않는다. 임의의 심벌 X에 대한 상속 속성의 집합을 I(X)로 표기하며 합성 속성의 집합을 S(X)로 나타낸다. 또한 I(X)와 S(X)의 합집합을 A(X)로 표기한다. 모든 터미널 심벌의 합성 속성의 값은 컴파일러의 어휘 분석기에 의하여 공급된다(예 identifier, number ...). 생성 규칙 p: X_0 -> X_1 X_2 ... X_{np}에 대해, $A(X_k)$에 a가 속한다면 생성 규칙 p는 속성 오커런스(attribute occurrence) (a, k)를 갖는다. 여기서, k는 0,1,..., np이다.

의미 규칙이란 각 생성 규칙(production rule)과 관계된 함수들로서, 생성 규칙의 RHS(Right Hand Side)에 있는 심벌의 상속 속성과 생성 규칙의 LHS(Left Hand Side)에 있는 합성 속성으로 사상되는 함수이다. 따라서, 본 논문에서는

의미 규칙을 문장에 따라서 의미 함수로서도 표현될 것이다. 즉, 각 생성 규칙 p에 대해 의미 함수의 집합이 존재하는데, k = 0인 합성 속성 (a, k)과 k = 1,2,...np 인 모든 상속 속성 (a, k)에 대해, 이 생성 규칙 p의 다른 속성 오커런스로부터 (a, k)로 사상되는 의미 함수f(p(a, k))를 말한다. 의미 함수의 종속 집합(dependency set) D(p(a, k))는 결과 속성(result attribute) (a, k)를 계산하는 데 사용되는 속성 오커런스들의 집합을 나타낸다. 따라서 결과 속성인 (a, k)를 계산하기 위해서는 D(p(a,k)) 내의 오커런스가 모두 계산 된 후에만 가능하다.

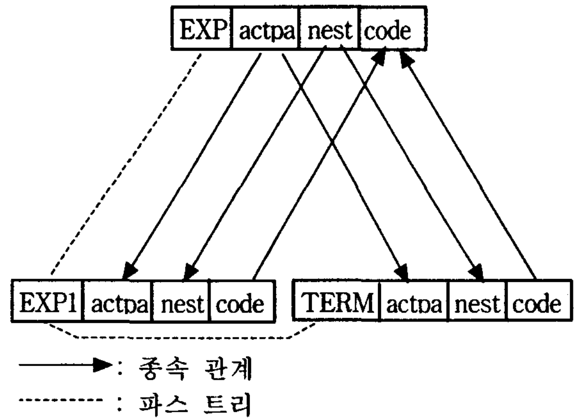
```
61: EXP -> EXP1 '+' TERM
 => EXP1.actpa     := EXP.actpa
    TERM.actpa     := EXP.actpa
    EXP1.nest      := EXP.nest
    TERM.nest      := EXP.nest
    EXP.code       := concat(EXP1.code,
                       TERM.code,emit_a('add'))
```

(그림 1) 식에 대한 속성 문법

(그림 1)에서 각 문법 심벌에 대한 속성들의 특성을 알아보면 다음과 같다.

I(EXP) = {actpa, nest}, S(EXP) = {code}
I(TERM) = {actpa, nest}, S(TERM) = {code}
A(EXP) = {actpa, nest, code}
A(TERM) = {actpa, nest, code}

이전에 설명한 바와 같이 속성 들 간에는 의존 관계가 발생한다. 이러한 속성 들 간의 의존 관계는 정보의 필요성에 따라 야기되는데, 이러한 종속 관계를 나타낸 그래프를 종속 그래프(dependency graph)라 한다. 종속 그래프에서 의미 함수 f(p(a, k))의 종속 집합(dependency set) D(p(a, k))로의 지시선(directed edge)이 존재한다. 생성 규칙 61에서 문법 심벌 EXP의 속성 code에 대한 종속 집합 D(61(code,0)) = {(code,1),(code,2)}이다. 그러므로 (code,1)에서 (code,0)로 그리고 (code,2)에서 (code,0)로의 지시선을 연결한다. 이와 같은 방법으로 그림 1의 생성 규칙 61에서의 모든 의미 함수들에 대해 종속 집합을 구하여 지시선을 연결한 종속 그래프의 형태가 그림 2이다.

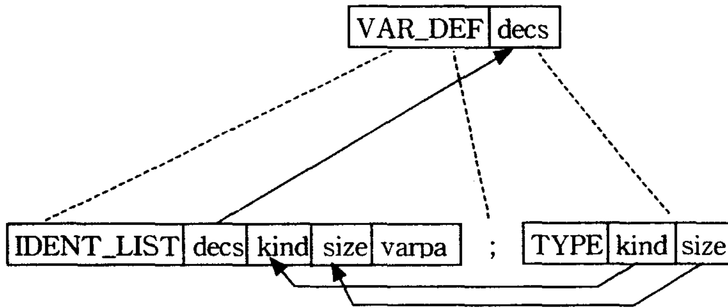


(그림 2) 연산식의 종속 그래프

이러한 의존관계는 형제 노드(sibling node)들 간에도 발생할 수 있다. 그에 대한 예는 변수의 선언에서 찾아 볼 수 있다. 생성 규칙 VAR_DEF -> IDENT_LIST ':' TYPE에 대한 종속 그래프는 (그림 3)과 같다.

이와 같이 상속 속성과 합성 속성이 모두 존재하는 경우, 의존 관계를 나타내는 종속 그래프가 환형(circularity)을 이룰 수 있다. 이러한 사이클을 이루는 경우, 원시 프로그램의 번역 당시에 사이클 만들어졌음을 확인하게 된다. 이는 올바른 프로그램임에도 불구하고 속성 문법의 사이클을 내포한 기술로 인하여 번역을 하지 못하는 문제점을 가진다. 이러한 문제점은 프로

```
15: VAR_DEF -> IDENT_LIST ';' TYPE
 => VAR_DEF.decs        := IDENT_LIST.decs
  IDENT_LIST.kind       := TYPE.kind
  IDENT_LIST.size := TYPE.size
  IDENT_LIST.varpa      := 0
```



(그림 3) 선언에 대한 속성 문법과 종속 그래프

그램의 번역 당시에 종속 그래프 내의 사이클을 발견하기 때문에 기인하는 문제이다. 이는 속성 문법의 기술 당시부터 어떤 제약을 가하여 사이클을 가진 종속 그래프를 근본적으로 생성하지 못하게 함으로써 해결할 수 있다. 이러한 제약을 가진 속성 문법에는 합성 속성만을 허락하는 S-attributed grammar과 합성 속성과 생성 규칙 내 평가하고자 하는 속성의 왼쪽에서 발생하는 문법 심벌의 속성에만 의존할 수 있다는 L-attributed grammar가 있다.

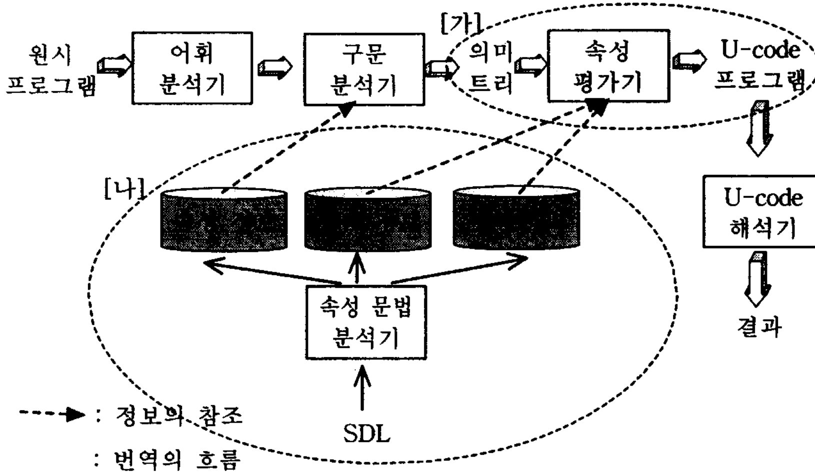
[정의 2.2] 각 합성 속성 a_j와 생성 규칙 $X_0 \rightarrow X_1X_2 \ldots X_n$ 에 대해, a_j와 연관된 모든 의미 함수의 형태가 $X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, X_1.a_1, \ldots, X_1.a_k, \ldots, X_{i-1}.a_1, \ldots, X_{i-1}.a_k)$이라면 이러한 속성 문법을 L-attributed grammar라 한다[13].

원래 L-attributed grammar는 파싱과 함께 속성을 평가하기 위해 속성 문법에 어느 정도의 제약이 가해진 문법이다. 이는 좌에서 우로 한번의 진행으로 모든 속성을 평가할 수 있으며, 문법이 LL(1) 형태의 CFG를 사용한 다면 LL 파싱 동안 모든 속성을 평가할 수 있다. 이러한 개념을 바탕으로 구성된 속성 평가기를 LL(1) L-attributed Evaluator라 한다[1,13]. 그러나 LR 파싱의 경우 모든 속성의 정보 흐름은 단말 노드로부터 부 노드로 제한된다. 따라서 상속 속성은 사용할 수 가 없고 합성 속성만을 평가할 수 있다. 이에 적합하도록 모든 논 터미널이 합성 속성만을 허락하는 속성 문법을 S-attributed grammar라 한다[12,13]. S-attributed grammar는 합성 속성만을 허락한다는 너무 강한 제약을 가지고 있다. 그러나 본 연구에서 구축하고자 하는 시스템은 트리 운행을 기본으로 하기 때문에 좀더 자유로운 L-attribute grammar을 사용하고자 한다.

Ⅲ. 코드 생성 시스템의 구축

3.1. 시스템의 구성 개요

코드 생성 시스템의 구성 형태를 그림으로 도시하여 보면 (그림 4)와 같다.



(그림 4) 전체 시스템의 구성 및 흐름

시스템은 크게 두 개의 부분으로 되어있다. 첫 번째는 원시 코드를 번역하여 목적 코드를 생성하는 부분이며, 두 번째는 의미 기술 언어(SDL; Semantic Description Language)를 입력으로 받아 구문 분석 및 속성 평가에 필요한 정보인 속성 정보, 종속 정보, 해석 정보를 생성하는 속성 문법 분석기(AGA; Attribute Grammar Analyzer) 부분으로 나뉘어진다. 첫 번째 부분에서 어휘 분석기는 일반적인 어휘 분석기와 동등하며 구문 분석기는 구문 분석의 결과로서 파스 트리 대신 속성 정보를 이용하여 의미트리를 생성한다는 것을 제외하면 기존 방법과 동등하다. 따라서, 번역 부분에서의 핵심 모듈인 속성 평가기(attribute evaluator)[가]와 번역에 필요한 정보를 생성하기 위한 SDL, AGA[나]를 차례로 다루도록 하겠다.

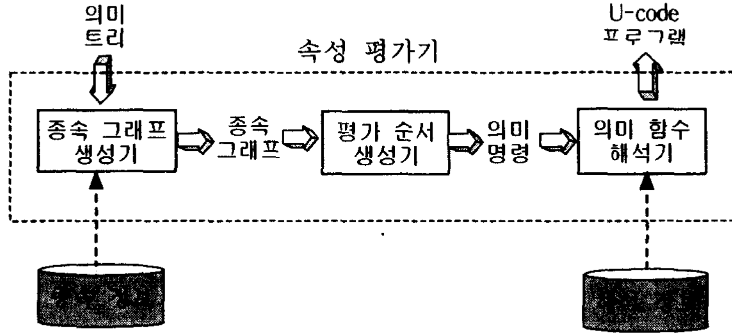
3.2. 속성 평가기(Attribute Evaluator)

의미 트리의 구성은 구문 분석 시 파스 트리를 구성하고 이 파스 트리의 각 노드에 AGA에서 얻어진 속성에 관한 정보를 이용 각 문법 심벌에 해당하는 속성을 붙인다. 이 단계는 구문 분석과 병행하여 처리된다.

의미 트리를 입력으로 받아 처리하게 되는 속성 평가기는 종속 그래프의 구성, 평가 순서 생성과 이를 입력으로 받아 U-code를 생성하는 의미 함수 해석기로 크게 구별된다. 이 트리 평가기의 구성 형태는 그림 5와 같다.

종속 그래프 생성기에서는 의미 트리를 탐색하면서 속성 문법 분석기(AGA)로부터 출력된 종속에 관한 정보를 이용하여 각 속성 노드들을 지시선으로 연결하고 속성 노드에 의미 함수에 관한 정보를 넣는다. 종속 그래프를 구성하기 위해 필요한 AGA에서 출력되는 종속에 관한

정보의 형태는 다음과 같다.



(그림 5) 속성 평가기의 구성

생성 규칙 p: X_0 -> X_1 X_2 X_3 ... X_{np}에 대한 종속 테이블의 형태는 다음과 같다.

(R# S#) k1 a1, k2 a2, k3 a3, ...

위 형태의 각 항에 대해 알아보면 다음과 같다.

- -R# : 생성 규칙의 번호
- -S# : 의미 함수의 번호
- -k1 a1 : 출력 속성 오커런스 (a1,k1)
- -k2 a2, k3 a3, ... : 입력 속성 오커런스 (a2,k2), (a3,k3), ...

여기서, k1, k2, k3, ...는 생성 규칙 p의 ki번째 문법 심벌을 나타내기 위한 인덱스이다. 즉, 0 ≤ ki ≤ np 이다. 또한 ai는 생성 규칙 p의 ki번째 심벌의 속성 들 중에 하나를 나타낸다. 다시 말해서 ai는 $A(X_{ki})$ 내의 임의의 한 원소이다. 또한 위 테이블에서는 속성 오커런스 (a1,k1)의 종속 집합 D(p(a1,k1))는 {(a2,k2), (a3,k3), ... }을 나타내고 있다. 이에 대한 실제 예를 생성 규칙 65를 이용하여 보여준 것이 그림 6 이다.

종속 그래프 구성은 파스 트리를 운행하면서 구성할 수 있는 데 방문 순서는 중요하지 않다. 파스 트리의 한 노드가 방문되었을 때 그 노드와 관련된 생성 규칙 pi와 관련된 종속에 관한 정보를 이용하여 각 속성 노드를 연결하고 방문된 속성 노드에서 수행되어야할 의미 명령을 저장한다. 이 때 속성 노드의 연결은 종속 집합의 각각의 원소에 대응되는 속성 노드로부터 결과가 되는 속성 노드 (a1,k1)에 대한 의미 명령을 저장하고 있는 속성 노드로 지시선을 연결한다.

의미 명령은 종속 집합 D(p(a,k))을 입력으로 하여 의미 함수 f(p(a,k))를 계산하기 위한 루틴으로 다음과 같은 구조를 갖는다.

(OPCODE, OUTPUT, INPUT)

의미 명령 각 항의 의미는 다음과 같다.

- -OPCODE : 의미 함수의 종류
- -OUTPUT : 결과 속성 오커런스 (a,k)에 대응되는 속성 노드로의 포인터
- -INPUT : D(p(a,k)) 내 속성 오커런스들에 대응되는 attribute node로의 포인터

여기서 실제로 속성 노드에 저장될 때는 OPCODE가 저장되어있는 속성 노드가 결과 노드가 된다. 또한 INPUT가 지시하는 속성 노드

```
[속성 문법]
65: TERM -> TERM1 '*' FACTOR
 => TERM1.actpa := TERM.actpa;
  FACTOR.actpa := TERM.actpa;
  TERM1.nest        := TERM.nest;
  FACTOR.nest := TERM.nest;
  TERM.code         := concat(TERM1.code, FACTOR.code,emit('mult'))

[종속 테이블]
 (65 1) 1 actpa, 0 actpa
 (65 2) 2 actpa, 0 actpa
 (65 3) 1 nest, 0 nest
 (65 4) 2 nest, 0 nest
 (65 5) 0 code, 1 code, 2 code
```

(그림 6) 속성 문법과 종속 테이블

에는 현재 속성 노드의 의미 명령이 시행되기 전에 먼저 시행되어야할 의미 명령이 저장되어 있다.

(그림 6)에 있는 생성 규칙의 5번째 의미 함수에 대한 의미 명령을 작성하면 다음과 같다.

OPCODE : (65, 5)
OUTPUT : TERM의 속성 code로의 포인터
INPUT : TERM1과 FACTOR 속성 code로의 포인터

종속 그래프가 완성되면 평가 순서 생성기에서 위상 정렬을 하여 평가 순서를 결정하고, 결정된 평가 순서에 의하여 종속 그래프의 각 속성 노드를 운행하며 의미 명령의 열을 구성한다. 위상 정렬에 의하여 생성된 의미 명령의 열은 AGA에 의해 생성된 해석 정보를 이용하여 의미 함수 해석기를 수행시킴으로써 평가된다. 평가가 끝난 후 의미 트리의 루트 노드의 합성 속성인 code의 값이 원시 프로그램에 대한 U-code의 열이다.

3.3. 속성 문법 분석기(AGA)

본 절에서는 속성 평가기에 필요한 각종 정보를 생성하는 속성 문법 분석기에 대해 알아보도록 하자. 속성 문법 분석기는 그의 입력으로서 의미 기술 언어(SDL; Semantic Description Language)로 기술된 특정 프로그래밍 언어에 대한 속성 문법을 입력으로 받는다. SDL은 프로그래밍 언어의 의미를 속성 문법으로 기술하기 위하여 설계된 메타 언어로서 SDL로 쓰여진 프로그램은 속성 문법 분석기에 의해 처리되어 속성 평가기에 필요한 정보로 변환된다. 간단한 속성 문법만으로는 평가에 필요한 정보를 생성하고 일반적인 프로그래밍 언어의 의미를 기술하기에 어려운 점이 있다. 예를 들어 각 속성의 형 또는 좀 더 복잡한 제어 구조를 갖는 문장 구조를 표현하기에 불충분하다. 그리하여 더욱 강력하고 효율적인 표현을 가지는 SDL 언어를 설계하게 되었다. SDL의 전체 구조는 다음과 같다.

```
agprog ;
  declar
 /* 각 심벌에 대한 속성들의 형을 정의. */
  enddcl
 /* 각 생성규칙에 대한 의미 함수를 정의. */
  endag
```

생성 규칙에 대한 의미 함수의 정의 부분은 다음과 같다.

생성 규칙 번호 : 생성 규칙

=> $Semantic_function_1$

$Semantic_function_2$

.

.

$Semantic_function_n$ (단, $0 < n$이다.)

위에서, 심벌 =>는 생성 규칙과 의미 함수를 구별하는 기호로서 사용하였다. 문법 심벌과 그 심벌에 대한 속성을 표기하는데 다음과 같이 점(dot)을 구분자로 하여 표기하였다.

grammar_symbol . attribute

또한 의미 함수의 표기는 다음 방법으로 기술된다.

A.a := B.b 에서

심벌 := 는 일반적인 프로그래밍 언어의 배정과 같은 의미를 가지며, 또한 B.b가 계산되기 전에는 A.a가 계산될 수 없다. 즉, "A.a는 B.b에 의존한다."는 의미를 내포하고 있다. AGA는 SDL을 입력으로 받아 각 논 터미널에 관련된 속성과 종속에 관한 정보, 그리고 의미 명령을 수행시키는데 필요한 인터프리터에 관한 정보를 생성하는 도구이다. AGA는 Lex와 Yacc을 사용하여 구성하였다.

Ⅳ. 평가 및 결론

본 시스템은 UNIX 계열의 Redhat 7.1에서 구현하고 실험하여 보았으며, 그의 수행 여부를 확인하기 위하여 미니 파스칼로 작성된 1부터 n 사이 정수의 합, 버블 정렬, 완전수 등 다양한 프로그램을 입력으로 하여 그의 실행 여부를 확인하여 보았다. 그 중 완전수 프로그램(그림 7)과 그를 입력으로 하여 생성된 U-코드는 (그림 8)에서 보여주고 있다.

```
program perfect;
const max = 100;
var i,j,k,r,sum: integer;
begin
 i:=2;
 while i<=max do
 begin
 sum:=0;
 k:=i div 2;
 j:=1;
 while j<= k do
  begin
  r:= i mod j;
  if r = 0 then sum:= sum+j;
  j:=j+1
  end;
 if i=sum then write(i);
 i:=i+1
 end
end.
```

(그림 7) 완전수를 구하는 프로그램

본 연구는 비 환경 속성 문법을 이용하여 컴파일러 구성을 빠르고 쉽게 구성하기 위한 시도이다. 또한 정형화된 의미 표현을 사용하여 컴파일러를 구성하게되므로 컴파일러의 의미 분석

```
[완전수에 대한 U-code]
  sym 1 1 0
  sym 2 1 0
  sym 3 1 0
  sym 4 1 0
  sym 5 1 0
  bgn 5
  ldc 2
  str 1 1
$$1 nop
  lod 1 1
  ldc 100
  le
  fjp $$2
  ldc 0
  str 1 5
  lod 1 1
  ldc 2
  divop
  str 1 3
  ldc 1
  str 1 2
$$3 nop
  lod  1 2
  lod  1 3
  le
  fjp   $$4
  lod  1 1
  lod  1 2
  modop
  str  1 4
  lod  1 4
  ldc  0
  eq
  fjp   $$5
  lod  1 5
  lod  1 2
  add
  str  1 5
$$5 nop
  lod  1 2
  ldc  1
  add
  str  1 2
  ujp   $$3
$$4 nop
  lod  1 1
  lod  1 5
  eq
  fjp   $$6
  ldp
  lod  1 1
  cal  write
$$6 nop
  lod  1 1
  ldc  1
  add
  str  1 1
  ujp   $$1
$$2 nop
  endop

[실행 결과]
***** execution start *****
 6 28
***** execution end *****
```

(그림 8) 속성 문법을 이용한 코드 생성 시스템에 의해 출력된 U-code

코드를 수동으로 작성함에 비해 오류의 소지를 줄이면서 또한 높은 생산성을 유도할 수 있는 방법이다. 속성 문법을 이용한 속성 평가기를 구현함에 있어 L-attributed grammar를 사용함으로서 언어 기술의 제약사항을 가지나 컴파일 시간에 발견되는 사이클 발견 문제를 컴파일러 구성시의 문제로 이전시킴으로서 반자동으로 구성된 컴파일러의 활용도를 높였다. 현재는 비순환을 이루기 위해 L-attributed grammar를 사용하며, L-attributed grammar의 정당성은 개발자의 책임이다. 앞으로 L-attributed grammar 보다 완화된 속성 문법으로 언어의 기술이 가능하며, 이러한 확장으로 인하여 발생될 수 있는 사이클 문제를 자동으로 탐지해 개발자에게 알려 줄 수 있는 지원도구가 필요하리라 여겨진다. 본 연구의 결과는 완전한 범용 컴파일러의 제작을 지원하기보다는 짧은 생명 주기를 갖는 시스템 또는 간단한 프로토타입 형태의 언어 등의 작고 단기간 사용되는 언어의 개발에 적합한 기술이라 사료된다.

참고문헌

[1] Aho A. V. and R. Sethi and J. D. Ullman, *Compilers principles, techniques and Tools,* Addison Wesley, 1986.

[2] Bochmann G. V, Semantic evaluation from left to right, *CACM, Vol.19,* No.2, 1976, pp. 55-62

[3] Bochmann G. V. and P. Ward, Compiler writing system for attribute grammars, *Computer Journal, Vol.21,* No.2, 1976, pp. 144-148,

[4] Dick Grune, Henri E. Bal, Ceriel J.H., *Modern compiler design,* Wiley Press, 2000.

[5] Engelfriet J., Attribute grammars: Attribute evaluation methods, in *Methods and tools for compiler construction,* Cambridge Univ. Press, 1984, pp. 103-138.

[6] Horowitz E. and S. Sahni, *Fundamental of data structures in pascal,* Computer Science Press, 1984.

[7] Farrow R., Yet another translator writing system based on attribute grammar, *Proceedings of the SIGPLAN 82 SYMPOSIUM on Compiler Construction,* 1982.

[8] Fisher C. N. and R. J. LeBlanc, Jr, *Crafting a compiler with C,* Benjamin/ Cummings Pub, 1991.

[9] Frank G. P. and Formal, *Specification of programming language*, Prentice-Hall, INC, 1981.

[10] Kennedy K. and J. Ramanathan, A deterministic attribute grammar evaluator based on dynamic sequencing, *ACM TOPLAS, Vol.1,* No.1, 1979, pp.142-160.

[11] Knuth D. E, Semantics of context-free languages, *Math. System theory, Vol.2,* 1968, pp.127-146.

[12] Lemone K. A., *Design of compiler techniques of programming language translation,* CRC Press, 1992.

[13] Louden K. C., *Compiler construction principles and practice,* PWS press, 1997.

[14] Waite W. M. and G. Goos, *Compiler Construction, springer-verlag,* 1984, pp. 183-252,

[15] Watt D. A. and O. L. Madsen, Extended attribute grammars, *The Computer Journal, Vol.26,* No.2, 1983, pp. 142-153,

The Code Generating System using Noncircular Attribute Grammar

Sang-Hoon Kim*

Abstract

This paper is the study for code generation using attribute grammar without cycle. For the purpose of this study, we designed metalanguage SDL which describes an attribute grammar and, represented attribute grammar to SDL. This attribute grammar is based on L-attributed grammar without cycle. This system consists of attributed-grammar analyzer and attribute evaluator. Attributed grammar analyzer takes a semantic description language and then generates useful informations that are needed by attribute evaluator. The evaluator calculates an attribute values using these information.

* Dept. of software Semyung Univ.