시멘틱 트리를 이용한 2단계 코드 생성
(2-Level Code Generation using Semantic Tree)

손윤식 · 오세만
동국대학교 컴퓨터공학과 컴퓨터공학전공
sonbug@dongguk.edu smoh@dongguk.edu

요 약

최근 컴파일러 이론은 컴파일의 구조를 여러 단계로 분리하여 구성함으로 컴파일러의 부분적인 기능이 개선 가능하고 이식성이 높으며, 모듈별 자동화가 용이한 형태를 추구하고 있다. 그러나 이러한 컴파일러 이론의 정립과는 달리 실제 컴파일러의 구현에는 실질적인 제약이 따른다.

본 논문에서는 이러한 컴파일러 이론 중 의미 분석과 코드 생성 측면을 일반화하여 자동화할 수 있도록 2단계 코드 생성 기법을 제안하고, 이를 위해 구문 분석과 의미 분석 및 코드 생성을 위한 중간 표현으로 AST와 시멘틱 트리를 사용한다. 또한, 이러한 기법을 실제적인 컴파일러 구현에 적용하고 생성된 코드를 검증하여 2단계 코드 생성의 타당성을 확인한다. 2단계 코드 생성 기법은 의미 분석과 코드 생성 과정을 용이하게 만들며, 각 단계의 자동화를 가능하게 해준다.

1. 서 론

최근 컴파일러는 컴파일의 각 과정을 여러 단계로 분리하고, 각 단계를 모듈화하여 구성한다. 이러한 구성은 기능에 따라 여러 단계로 구성되어 있기 때문에 컴파일러의 부분적인 기능을 개선하고 다른 기종으로 이전하기 쉬우며, 상대적으로 작은 기억공간을 사용하고, 모듈별 자동화가 용이하다. 이러한 컴파일러의 구성에 대한 연구가 각 단계별로 진행되었으며, 정형화된 이론으로 자리 잡혔지만, 실제 컴파일러의 구현에는 많은 부분 제약이 따르고 있으며, 일관적인 컴파일러 구성이 이루어지기 힘든 단점이 있다.[1][10]

본 논문에서는 이러한 단점을 보완하기 위하여 여러 컴파일러에 대한 이론을 일반화하여, 의미 분석과 코드 생성에 용이한 2단계 코드 생성 기법을 제안하고 이를 실제 컴파일러 구현에 적용하여 그 결과를 검증하도록 한다. 컴파일러 구성에 있어 중요하게 고려된 부분은 의미 분석과 코드 생성, 그리고 코드의 검증이다[11].

의미 분석은 파서의 구문 분석 결과인 AST와 심벌 테이블을 이용하여 분석 과정을 수행하며, 그 결과로 시멘틱 트리를 생성한다. 시멘틱 트리는 프로그램 구문구조 정보와 의미 정보가 동시에 반영된 자료구조로써 효율적인 의미 분석과 코드 생성이 용이한 형태이다. 코드 생성은 시멘틱 트리

와 패턴 매칭 코드 생성기법을 이용하여 효율적으로 타겟 코드를 생성하게 된다. 타겟 코드는 디컴파일러를 이용하여 소스 프로그램으로 변환되며, 이 과정을 이용하여 생성된 코드에 대한 검증을 한다.

본 논문은 다음과 같은 순서로 2단계 코드 생성에 대한 기법에 대해 논한다. 우선 2장에서 2단계 코드 생성과 의미 분석 과정을 효율적으로 처리할 수 있는 시멘틱 트리의 특징을 분석하고 패턴 매칭 코드 생성 기법에 대해 그 장단점을 파악 한다. 이어 3장에서는 앞서의 연구를 바탕으로 AST에서 시멘틱 트리를 생성하는 과정으로 트리 변환 모델을 제시한 다음 트리 변환 메소드를 설명한다. 또한 시멘틱 트리를 이용하여 타겟 코드를 생성하는 기법에 대해 논한다. 4장에서는 실제 프로그래밍 언어를 대상으로 AST와 시멘틱 트리를 설계하고 트리 변환 메소드와 코드 생성기를 구현하여 그 과정을 보이고, 디컴파일러를 이용하여 실험 결과에 대한 검증을 한다. 마지막으로 5장에서는 연구의 결과와 앞으로의 연구 방향에 대해 제시한다.

2. 관련 연구

2.1 2단계 코드 생성

2단계 코드 생성은 구문 분석 과정과 의미 분석 과정을 별도의 중간 표현을 사용하여 각 과정의 독립성을 높이고 효율적인 분석을 가능하게 하기위한 기법이다.

구문 분석 결과와 의미 분석 결과를 표현하기 위한 자료구조로 각각 AST와 시멘틱 트리를 사용하며, 기존의 컴파일러 이론과 자동화 도구로 작성이 수월한 구분 분석 과정과, 분석 과정이 복잡한 의미 분석 과정을 분리하여 각 과정을 효율적으로 처리할 수 있는 구조이다. 또한, AST로 표현이 힘든 속성 정보와 코드 생성 정보를 시멘틱 트리를 이용하여 표현함으로써 의미 분석 과정과 코드 생성 과정이 용이해지며, 의미 분석과정에 대한 자동화가 가능하다.

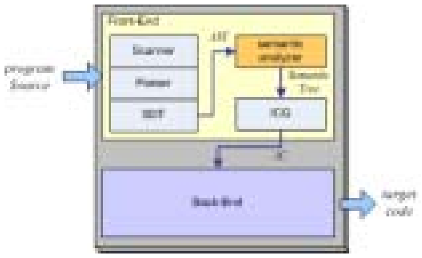

그림 1. 2단계 코드 생성 컴파일러

2.2 시멘틱 트리

시멘틱 트리는 의미 분석을 위한 이진 트리 형태의 자료구조로 프로그램의 구문 구조 정보인 AST에 의미 정보가 반영된 형태다. 구문 구조 정보인 AST는 의미 정보의 효율적인 표현이 힘들고 코드 생성시 복잡도가 높아지는 반면 시멘틱 트리는 의미 분석 정보가 포함되어 있으며, 코드 생성에도 용이한 형태이다[2]. 또한 시멘틱 트리는 프로그램의 구문 구조 정보를 효율적으로 표현하는 AST를 기반으로 정의하였기 때문에, AST가 가지는 구문 구조 정보를 모두 포함한다.

시멘틱 트리는 프로그램의 의미정보를 구조적으로 표현하며, 시멘틱 노드를 기본 단위로 한다. 각각의 심벌에 대한 의미 정보는 시멘틱 트리의 시멘틱 노드가 관리하고 프로그램의 구조적인 표현은 시멘틱 노드로 구성된 트리의 형태와 AST의 프로그램 구조 노드의 결합으로 이루어진다. 시멘

틱 트리는 의미 분석 과정에서 사용하는 속성을 시멘틱 노드의 종류로 구분 짓고, 각각 역참조와 형 변환, 형 부가 연산자로 노드로 구별한다.

시멘틱 트리는 의미 분석과정에서 속성 정보 수집 외에 코드 생성을 위한 추가 정보를 수집하며, 정보는 각각의 시멘틱 노드가 보유하게 된다. 또한 이 과정에서 소스 언어의 복잡한 연산을 단순화하여 표현함으로 코드 생성이 용이해진다. 이러한 의미 정보와 코드 생성 정보는 AST를 이용하여 의미 분석하는 과정에서 트리 변환 메소드에 의해 얻어진다.

시멘틱 트리는 의미 정보 및 구문 구조를 표현함과 동시에 시멘틱 트리의 구조와 시멘틱 노드가 유일성을 가지는데, 이는 트리 변환 메소드을 이용하여 의미 분석을 하기 위한 전제 조건으로, 시멘틱 트리의 구조와 시멘틱 노드가 각각의 프로그램 구조에 따라 유일성을 가지게 되므로 트리 변환 메소드를 일관되게 적용시킬 수 있다.

프로그램의 구조를 표현하기 위한 시멘틱 트리의 구조는 AST는 기반으로 한다. AST의 구성은 일반적으로 SDT를 이용하기 때문에 언어의 문법 설계와 동시에 AST의 구조를 정의하게 되고, 시멘틱 트리는 AST의 구문 구조 정보를 반영하여 그 구조를 정의한다.

2.3 패턴 매칭 코드 생성 기법

패턴 매칭 코드 생성 기법은 코드 생성에 필요한 정보와 패턴을 미리 결정하고 해당 패턴에 따라 생성될 코드를 정의하여, 입력을 패턴에 매칭하면서 코드를 생성하는 기법이다.

패턴 매칭 코드 생성의 입력에 사용되는 자료구조는 리스트나 트리등의 다양한 구조가 가능하며, 코드 생성이 빠르며, 정의된 패턴과 타겟 코드 정보를 기준으로 코드를 생성하기 때문에 가상 기계와 레지스터 기계를 대상으로 코드 생성이 용이한 방법이다. 또한 구현이 비교적 간단하며 자동화가 수월하다[1][10].

반면 패턴의 종류와 그 형태에 따라 패턴의 검색에 대한 복잡도가 높아지기 때문에 패턴의 정의와 패턴의 형태에 대한 설계가 중요하다.

3. 타겟 코드 생성

3.1 시멘틱 트리 변환

AST를 시멘틱 트리로 변환하기 위해서는 각각의 AST 노드에 해당하는 트리 변환 메소드를 정의해야한다. 트리 변환 메소드는 AST를 시멘틱 트리로 변환하면서 의미 분석을 하고, 시멘틱 트리에 의미 분석 결과를 반영한다. <그림 2>는 트리 변환 모델의 전체적인 모습이다.

트리 변환 모델에서 AST는 프로그램의 구문 구조 정보를 표현하고 각 심벌에 대한 정보는 심벌 테이블에 표현한다. 이러한 정보는 AST의 노드별로 정의되어 있는 트리 변환 메소드를 통하여 수집되고 의미 분석 과정을 거친 다음 시멘틱 트리로 변환된다.

의미 분석을 위한 트리 변환 메소드는 크게 심벌의 속성 정보를 계산하는 메소드와 타입변환 메소드, 노드 변환 메소드, 그리고 프로그램의 제어 흐름을 분석하는 제어 흐름 메소드로 구성된다. 우선 심벌의 속성 정보를 계산하는 메소드는 다음과 같은 형식으로 이루어진다.

의미 분석을 위한 심벌 정보는 심벌 테이블에 저장이 되어 있으며, 계산된 속성

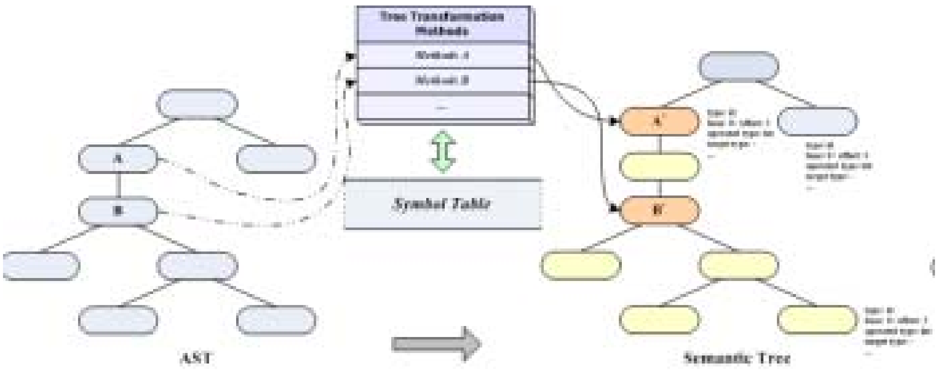

그림 2. 트리 변환 모델

정보는 시멘틱 노드에 저장된다. 이는 시멘틱 노드를 대부분 구문 구조 표현을 위한 연산 위주로 분류한 데에 기인한다. 저장되는 정보로는 일반적인 의미 분석에 사용되는 심벌 속성과 타입 정보, 그리고 코드 생성을 위한 base/offset 등이다.

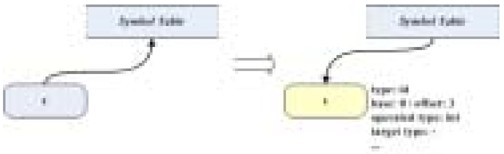

그림 3. 시멘틱 노드의 속성 부여

다음으로 타입 변환 메소드는 타입 변환이 가지는 특성(transparency)을 이용하여 AST의 노드를 타입 변환 시멘틱 노드로 변환한다. 연산에 따른 타입 변환은 synthesized attributes를 이용하여 해당 메소드를 적용하는데 자식 노드의 타입 속성을 이용하여 미리 정의된 변환 규칙에 따라 트리를 확장한다[5].

일반적인 노드 변환 메소드는 다음 세가지 규칙을 따른다. 첫째, AST의 기본 노드는 전부 시멘틱 트리의 타입부가 노드로 변환한다. 둘째, AST 복합 연산 노드의 경우는 의미 분석과 코드 생성의 용이를 위하여 시멘틱 트리의 기본 연산 노드로 확장한다. 셋째, AST의 상수 연산 노드에 대해서는 연산결과를 반영하여 상수 노드로 대치한다.

<그림 4>는 배열 연산에 대한 AST와 배열 연산을 기본 연산으로 분해한 시멘틱 트리의 모습이다.

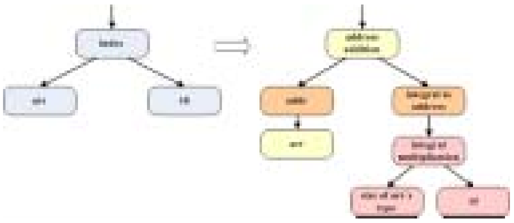

그림 4. 배열 연산(arr[10])에 대한 AST와 시멘틱 트리

마지막으로 프로그램의 제어 흐름을 분석하기 위해서 프로그램 상의 가능한 분기점을 추적하여 그 흐름을 트리에 기술한다. 제어 흐름을 분석하기 위해서는 우선 프로그래밍 언어별로 정의된 분기문에 해당하는 AST 노드를 선택하여 이를 기준으로 기본블록(basic block)을 구분하고 각 분기에 해당하는 노드간을 서로 연결함으로써 제어흐름을 표현한다. <그림 5>의 시멘틱 트리는 <표 1>의 if-else 분기에 대한 제어 흐름이 어떠한 식으로 표현되는 가를 보여준다.

표 1. if-else 제어 흐름 예제 프로그램

```
…
if ( i > 100) {
    return i / 100;
} else if (i > 10) {
    return i / 10;
}
else {    // return i % 10;}
…
```

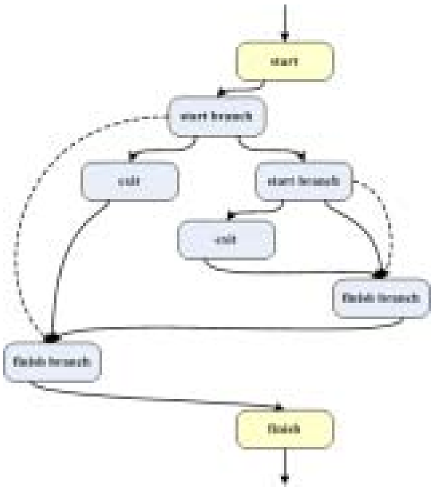

그림 5. 제어흐름이 표현된 시멘틱 트리

제어 흐름 노드는 프로그래밍 언어의 분기문과 의미적으로 연결되어 각 분기 구문에 따라 가능한 분기 경로를 확인하고 이를 표현한다. 제어 흐름이 표현된 시멘틱 트리에서 실선으로 표시된 부분은 실제 제어 흐름의 경로이며 점선으로 표시된 부분은 각 분기가 발생할 경우 그 분기와 대를 이루는 분기의 끝을 나타낸다.

3.2 타겟 코드 생성기

의미 분석과정을 거쳐 생성된 시멘틱 트리는 패턴 매칭 코드 생성기를 통하여 타겟 코드를 생성한다. <그림 6>은 패턴 매칭 코드 생성기의 모델이다.

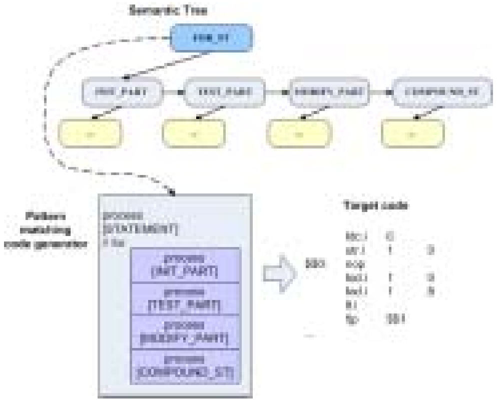

그림 6. 패턴 매칭 코드 생성기 모델

프로그램의 구문 구조를 표현하는 AST와 시멘틱 트리의 형태를 분석하여 시멘틱 트리의 각 하위 트리의 노드를 패턴으로 구성하여 해당 패턴과 일치하는 코드를 생성하는 구조를 가지고 있다. 이미 구문적/의미적으로 타당한 시멘틱 트리를 운행하며 코드를 생성하기 때문에 효과적인 코드 생성이 가능하다. 시멘틱 트리의 각 노드는 이미 코드 생성을 위한 정보가 추가되어 있고, 코드 생성에 용이한 형태로 단순화되어 있으므로 패턴 매칭 코드 생성기는 그 구성의 복잡도를 줄일 수 있다.

4. 구 현

4.1 시멘틱 트리의 구현

다음은 제안된 트리변환 기법을 이용하여, ANSI C 프로그래밍 언어를 실제 시멘틱 트리화한 결과이다. 총 122개의 AST 노드와 C언어의 의미정보를 기반으로 245개의 의미 정보 노드를 정의하였다.

<표 2>는 정의된 AST를 기준으로 각 연산에 해당하는 시멘틱 트리를 나열한 것

이다. AST의 각 연산 노드는 기본적으로 ANSI C 언어가 가지는 타입에 따라 시멘틱 트리로 확장되며, 연산에 대해 언어가 가지는 의미에 따라 시멘틱 트리 노드에서 불필요한 타입을 제거하였다.

표 2. 시멘틱 트리 노드

AST Node	Semantic Tree Node
ADD / SUB	ADD / SUB (I, U, L, P, F, D)
MUL / DIV	MUL / DIV (I, U, L, F, D)
MOD	MOD(I, U, L)
NEG	NEG(I, L, F, D)
EQ / NE / GE / GT / LE / LT	EQ / NE / GE / GT / LE / LT (I, U, L, F, D)
LOGICAL_AND / LOGICAL_OR / LOGICAL_NOT / BITWISE_AND / BITWISE_OR / BITWISE_XOR / LEFT_SHIFT / COMP /	AND / OR / NOT / BAND / BOR / XOR / SHL / BCOM (I, L)
RIGHT_SHIFT	[U]SHR(C, S, I, L)

타입 변환을 위한 시멘틱 노드는 <표 3>과 같다. 우선 타입 변환 노드를 결정하기 위하여, ANSI C 언어의 타입과 변환 특징을 반영하여 <그림 7>과 같은 그래프를 작성하고 이를 기반으로 타입 변환 노드를 결정하였다. 타입 변환 노드는 ANSI C가 가지는 타입이 모두 변환이 가능하도록 노드를 결정하였고, 타입 변환에 따르는 의미적인 부분은 타입 변환 그래프와 트리 변환 메소드를 이용하여 분석한다.

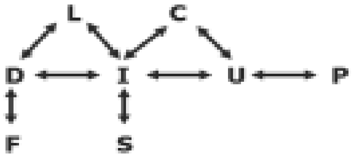

그림 7. C 언어의 타입 변환 그래프

표 3. 타입 변환 노드

Convert to	Semantic Tree Node
char	CV(S, I, U, L, F, D) _C
short	CV(C, I, U, L, F, D) _S
int	CV(C, S, U, L, F, D) _I
unsigned	CV(C, S, I, L, F, D) _U
long	CV(C, S, I, U, F, D) _L
float	CV(C, S, I, U, L, D) _F
double	CV(C, S, I, U, L, F) _D

역참조를 위한 노드는 다음과 같다. 변수에 대한 l-value와 r-value를 결정하는 노드로 <표 4>와 같이 두 가지가 존재한다.

표 4. 참조, 역 참조 노드

Semantic Tree Node
ADDR, VALUE

시멘틱 트리의 시멘틱 노드는 위에서 언급한 노드 외에 추가로 총 245개의 노드로 정의되었으며, AST에서 시멘틱 트리로 변환하면서 의미 분석을 하는 트리 변환 메소드는 다음과 같이 그룹화 하였다.

표 5. 트리 변환 메소드

의미 정보 추가	type determine address calculation
연산 노드 변환	basic node manipulation complex node manipulation check member operation check index operation constant folding
제어 흐름 추가	check control flow analysis

각 심벌의 속성 및 AST의 노드의 타입과 주소 정보를 계산하는 트리 변환 메소드는 의미정보 추가 그룹에 해당하며, 연산 노드의 변환과 프로그램의 연산 식 반영, 상수 최적화 등은 연산 노드 변환 그룹에서 처리한다. 마지막으로 프로그램의 제어

흐름 분석은 제어 흐름 추가 그룹에서 이루어진다.

4.2 타겟 코드 생성

다음은 구현된 시멘틱 트리와 트리 변환 메소드를 이용하여 프로그램을 AST, 시멘틱 트리 타겟 코드로 변환한다. 타겟 코드로는 EVM의 가상 기계 코드인 SIL 코드를 사용한다[9]. 소스 프로그램은 ANSI C언어의 대부분의 특징을 표현하는 프로그램으로 선택하였으며 일부분을 발췌하였다.

표 6. 소스 프로그램(perfectNumber.c)

```
#include <stdio.h>
void main(){
    int n, i, k;

    for (i=1; i<=500; i++) {
        k =0;
        for (n=1; n <= i/2; n++) {
            if (i%n == 0) {
                k += n;
            }
        }
        if (i == k){
            printf("%d ", i);
        }
    }
}
```

<표 7>은 <표 6>의 소스 프로그램을 구문 분석한 결과인 AST의 일부분이다.

표 7. 구문 분석 결과(perfectNumber.ast)

```
...
    Nonterminal: FOR_ST
         Nonterminal: INIT_PART
             Nonterminal: ASSIGN_OP
                 Terminal( Type:id / Value:n )
                 Terminal( Type:int / Value:1 )
        Nonterminal: TEST_PART
             Nonterminal: LE
                 Terminal( Type:id / Value:n )
                 Nonterminal: DIV
                     Terminal( Type:id / Value:i )
                     Terminal( Type:int / Value:2 )
...
```

다음은 위 AST를 입력으로 각각의 AST 노드에 해당하는 트리 변환 메소드를 실행하고 그 결과로 구성된 시멘틱 트리이다.

표 8. 의미 분석 결과(perfectNumber.st)

```
...
// for (n=1; n <= i/2; n++) {
Nonterminal: FOR_ST
    Nonterminal: INIT_PART
        Nonterminal: ASSIGN_OP / operatedType: 3
            Terminal( Type:id / Value:n / operatedType:3
                        / targetType:-1 / qualifier:0
                        / (b:1, o:0, w:4))
            Terminal( Type:int / Value:1
                       / operatedType:3)
    Nonterminal: TEST_PART
        Nonterminal: LEI / operatedType: 3
            Terminal( Type:id / Value:n / operatedType:3
                        / targetType:-1 / qualifier:0
                        / (b:1, o:0, w:4) / Tag:0 / Dim:0)
            Nonterminal: DIVI / operatedType: 3
                Terminal( Type:id / Value:i
                           / operatedType:3
                            / targetType:-1 / qualifier:0
                            / (b:1, o:4, w:4))
                Terminal( Type:int / Value:2
                           / operatedType:3)
...
```

의미 분석결과로 각 노드의 심벌 속성이 추가되고, 타입 변환 등의 특징이 트리에 반영되어 AST가 시멘틱 트리로 확장된 결과를 확인할 수 있다.

<표 9>는 시멘틱 트리를 패턴 매칭 코드 생성기의 입력으로 하여 타겟 코드인 SIL 코드를 생성한 결과이다.

표 9. 타겟 코드(perfectNumber.sil)

```
...
%Line 136: for (n=1; n <= i/2; n++) {
             ldc.i 1
             str.i 1 0
$$3:         nop
             lod.i 1 0
             lod.i 1 4
             ldc.i 2
             div.i
             le.i
             fjp $$4
```

```
  %Line 137: if (i%n == 0) {
             lod.i 1 4
             lod.i 1 0
             mod.i
             ldc.i 0
             eq.i
             fjp $$6
  ...
```

4.3 코드 검증

생성된 코드가 소스 프로그램을 올바르게 변환한 것인지 검증하는 타겟 코드 별로 여러 가지 방법이 있다. 타겟 코드인 SIL 코드는 가상기계 코드이며 이를 검증하기 위해서는 SIL 코드를 가상기계에서 실행하는 방법과 SIL 코드를 디컴파일러를 통하여 소스 언어의 프로그램으로 변환한 후, 이를 기존의 컴파일러를 이용하여 실행 파일로 변환하고 그 결과를 비교하는 방법이 있다[11]. 본 논문에서는 타겟 코드의 검증과정에서 디컴파일러를 사용하며 그 모델은 <그림 8>과 같다.

<표 10>은 디컴파일러를 이용하여 타겟 코드를 소스 언어의 프로그램으로 변환한 결과이다.

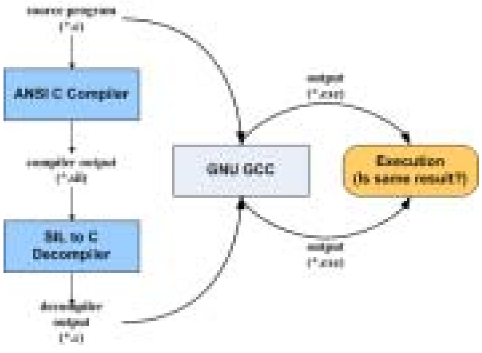

그림 8. 타켓 코드 검증 모델

표 10. 디컴파일 결과(perfectNumber.sil.c)

```
#include "perfectNumber.sil.h"
// C SourceFile : perfectNumber.c
/* global type dcl */
/* global sym dcl */

void main()
{
/* local type dcl */
/* local sym dcl */
int silSym_0;
int silSym_1;
int silSym_2;

   // C SourceLine(134) : for (i=1; i<=500; i++) {
   silSym_1=1;
$$0:    {/* label not operation */}
   if(!(silSym_1<=500))    goto $$1;
   // C SourceLine(135) : k =0;
   silSym_2=0;
   // C SourceLine(136) : for (n=1; n <= i/2; n++) {
   silSym_0=1;
$$3:    {/* label not operation */}
   if(!(silSym_0<=(silSym_1/2)))    goto $$4;
   // C SourceLine(137) : if (i%n == 0) {
   if(!((silSym_1%silSym_0)==0))    goto $$6;
   // C SourceLine(138) : k += n;
   silSym_2=silSym_2+silSym_0;
$$6:    {/* label not operation */}
$$5:    {/* label not operation */}
   silSym_0=silSym_0+1;
   goto $$3;
$$4:    {/* label not operation */}
   // C SourceLine(141) : if (i == k){
   if(!(silSym_1==silSym_2))    goto $$7;
   // C SourceLine(142) : printf("%d ", i);
   printf("%d ", silSym_1);
$$7:    {/* label not operation */}
$$2:    {/* label not operation */}
   silSym_1=silSym_1+1;
   goto $$0;
$$1:    {/* label not operation */}
}
```

소스 프로그램과 디컴파일된 프로그램을 컴파일하여 실행한 결과는 <표 11>과 같이 동일하였으며, 생성된 코드가 이상이 없음을 검증하였다.

표 11. 실행 결과

```
6 28 496
```

5. 결론 및 향후 연구

본 논문에서는 컴파일러를 구조화하고 여러 단계로 분리하는 기존 이론 연구를 바탕으로 AST와 시멘틱 트리를 이용하여, 의미 분석과 코드 생성이 효율적인 2단계 코드 생성 컴파일러를 구현하였다. 또 각 단계별로 그 결과를 확인하고, 생성된 타겟 코드에 대해서는 디컴파일러를 통하여 검증을 하였다. 구현된 ANSI C 컴파일러는 기존 컴파일러에서 처리하기 힘든 과정을 2단계로 분리함으로써 번역 과정을 구조화하고 보다 단순화시킬 수 있다. 또한 각 단계가 명확하게 구분됨으로써 유지 보수 및 타 기종으로의 이식성이 향상되고, 컴파일러의 각 단계에 대한 자동화 가능하다.

향후 과제로는 이러한 기법을 C++이나 C# 컴파일러의 구현에 적용하여 그 가능성을 확인하고, 적립된 컴파일러 이론에 대한 일반화 과정이 필요하다. 또한 AST에서 시멘틱 트리를 생성하는 과정을 보다 정형화하여 AST 노드와 시멘틱 노드, 트리 변환 메소드의 매핑을 통한 의미 분석기 자동화 도구에 대한 연구가 필요하다.

참고문헌

[1] A. V. Aho, R. Sethi, J. D. Ullman. Compilers: Principles, Techniques, and Tools, Addison Wesley, 1988.

[2] B. M. Brosgol, “TCOLAda and the Middle End of the PQCC Ada Compiler,” Proceedings of the ACM−SIGPLAN Symp. on The ADA programming lanugage, pp.101−112, 1980.

[3] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, Koen G. Langendoen, Modern Compiler Design, John Wiley & Sons, 2000.

[4] D. E. Knuth, “The Genesis of Attribute Grammars,” ACM Proceedings of the international conference on Attribute grammars and their applications, pp.1−12, 1990.

[5] J. C. Mithell, “Coercion and Type Interface,” 11th ACM Symp. on Principles of Programming languages, pp.175−185, 1984.

[6] Kai Koskimies, “A specification language for one−pass semantic analysis,” Proceedings of the 1984 SIGPLAN symposium on Compiler construction, pp.179−189, 1984.

[7] Mark S. Sherman, Martha S. Borkan, “A flexible semantic analyzer for Ada,” ACM SIGPLAN Notices, Proceeding of the ACM−SIGPLAN symposium on Ada programming language, Volume 15 No. 11, pp.62−71, 1980.

[8] S. S. Muchnick, Advanced Compiler Design Implementation, Morgan Kaufmann Press, 1997.

[9] 남동근, 가상 기계를 위한 어셈블리 언어의 설계, 동국대학교 석사학위 논문, 2003.

[10] 오세만, 컴파일러 입문, 개정판, 정익사, 2004.

[11] 이양선, 오세만, 김영근, 권혁주, 손윤식, 박성환, 임베디드 시스템을 위한 ANSI C 컴파일러 개발, 서경대학교 산학협력단, 2004.

손 윤 식

97 ~ 04 동국대학교
컴퓨터공학과(학사)
04~현재 동국대학교
컴퓨터공학과 석사 과정
관심분야 : 프로그래밍 언어, 컴파일러, 모바일 컴퓨팅

오 세 만

85.03 ~ 현재 동국대학교
컴퓨터공학과 교수
93.03 ~ 99.02 동국대학교
컴퓨터 공학과 대학원
학과장
01.11 ~ 03.11 한국정보과학회 프로그래밍언어연구회
위원장
04.06 ~ 현재 한국정보처리학회 게임연구회 위원장
관심분야 : 프로그래밍 언어, 컴파일러, 모바일 컴퓨팅