클래스 기반의 의미수행코드 명세를 이용한 시각언어 컴파일러 자동 생성

김 경 아[†]

요 약

의미 수행 코드를 이용한 문법-지시적 변환 방법은 컴파일러 설계자가 원시 언어의 구문 구조에 따라 직접 컴파일러의 후단부를 표현할 수 있는 효과적인 방법으로 텍스트 프로그래밍 언어에서는 컴파일러 구축 방법으로 널리 사용되고 있다. 그럼에도 불구하고 시각언어의 경우에는 통합된 파스 트리 노드 표현 방법의 부족과 구문 구조의 모델링 방법의 결여로 인하여, 의미 수행 코드를 이용한 문법 지시적 변환 방법에 기반을 둔 컴파일러 구축에 어려움이 있다. 본 연구에서는 Pictorial Class Grammar의 시각언어 구문 정의를 캡슐화하는 방법과 And-Or-Waiting Graph를 이용한 구문 분석 방법을 기반으로 하여, 구문 요소를 하나의 클래스 형태로 구성함으로써, 구문 요소의 표현에 사용되는 의미와 구문의미(syntax-semantics)를 분리할 수 있는 방법을 제시한다. 이 방법에 기초하여 기존 연구들의 문법-지시적 시각언어 컴파일러 구축의 문제점을 극복하고, 구문 명세와 분리된 의미 수행 코드 명세 방법을 제시하여, 유지보수성을 보다 향상시킨 문법-지시적 변환 방법을 이용한 시각언어 컴파일러 자동 생성 방법을 제공한다.

Automatic Compiler Generator for Visual Languages using Semantic Actions based on Classes

Kyung-Ah Kim[†]

ABSTRACT

The syntax-directed translation using semantic actions is frequently used in construction of compiler for text programming languages. It is very useful for the language designers to develop compiler back-end using a syntax structure of a source programming language. Due to the lack of the integrated representation method for a parse tree node and modeling method of syntax structures, it is very hard to construct compiler using syntax-directed translation in visual languages. In this paper, we propose a visual language compiler generation method for constructing a visual languages compiler automatically, using syntax-directed translation. Our method uses the Picture Layout Grammar as a underlying grammar formalism. This grammar allows our approach to generate parser efficiently using And-Or-Waiting Graph and encapsulating syntax definition as one unit. Unlike other systems, we suggest separating the specification and the generation of semantic actions. Because of this, it provides a very efficient method for modification.

Key words: 시각언어 컴파일러, 컴파일러 자동화 도구, 문법-지시적 변환 방법, 의미 수행 코드, 클래스-의존 관계 등

1. 서 론

시각언어는 1차원 형태의 텍스트가 아닌 2차원 형태의 시각 구문을 다루는데 있어서의 어려움으로 인해, 컴파일러나 인터프리터와 같은 언어 처리 도구의 개발 과정이 텍스트 프로그래밍 언어 보다 복잡하여 더 많은 노력과 시간을 요구한다. 그 동안 많은 연구에서 이러한 어려움을 해결하는 방법의 하나로 컴파

접수일 : 2002년 12월 26일, 완료일 : 2003년 3월 13일
[†] 정회원, 명지전문대학 컴퓨터정보과 교수

일러 자동화 도구를 이용하여 시각언어를 구현하고자 하였다[3,5,7,9,10]. 다양한 시각언어의 개발과 활용을 위해서는 텍스트 프로그래밍 언어에서 사용되고 있는 *lex*[1]와 *yacc*[1]과 같이 실질적인 언어 처리 도구의 개발 과정에 편리하게 사용할 수 있는 시각언어를 위한 자동화 도구가 필요하다. 그러나 기존 연구들은 기본적으로 개발과정에서 요구되는 노력을 감소시키는 결과는 얻을 수 있었으나 시각언어 개발자가 실질적인 개발 과정에 효과적으로 사용하는데는 각각 한계점을 내포하고 있다.

SPARGEN[7]은 시각언어 컴파일러 생성기로서 action routine을 사용하는 형태를 제공하고 있다. 그러나 전반적으로 정의된 문법을 위한 그래픽 전단부가 부족하다는 제한점으로 인해 파서는 생성할 수 있으나 컴파일러의 다른 부분을 포함하는 시각언어의 프로그래밍 환경을 개발하는 과정에 사용하기에는 어려움이 있다. 뿐만 아니라 기존 클래스와 독립된 완벽한 의미의 새로운 클래스를 정의할 수 없고, 파서의 효율성이 떨어진다. PROGRES[9]는 파싱 알고리즘이 매우 복잡하여 자동 생성 과정의 복잡성을 유발시킨다. 또한 언어의 의미를 처리하는 방법에서 시스템 자체의 한 부분인 간단한 텍스트 프로그래밍 언어를 사용함으로써 대상 시각언어의 기능을 제한하는 문제점을 내포한다. VLCC[5]는 임의의 시각언어의 구현을 지원하는 문법 지원 시스템으로 구문 분석 단계에서 우선 순위를 적용하여 시각 객체의 2차원 결합을 텍스트 심볼의 일차원 결합으로 변환한다. 이는 시각 언어의 공간적인 특성을 충분히 반영하지 못함으로써 일반적인 시각 프로그래밍에 쉽게 적용하는데 어려움이 있다. 또한 새로운 그래픽 객체를 생성하는데 있어 원하는 객체 생성이 주어진 정보에 제한을 받고 이를 추가하거나 변경함에 있어 언어설계자의 의도를 반영하는데 어려움이 있다. VisPro[10]는 어휘 분석 단계와 구문 분석 단계를 구별하여 처리할 수 있는 방법과 범용 프로그래밍 언어를 사용한 의미 표현 방법을 제공한다는 측면에서는 앞에서 기술한 기존 방법보다 향상된 방법을 제공한다. 그러나, 이 도구가 기반으로 하는 Reserved Graph Grammar는 Graph Grammar의 부분집합으로 제한된 형태의 시각언어에만 적용 가능하다는 한계점으로 인해 개발한 도구의 일반성에 문제점이 남아 있다.

문법-지시적 변환방법은 전단부 컴파일러 자동화 도구의 효용성을 증대시키는 유용한 방법으로 의미 수행 코드를 이용한 컴파일러 후단부를 입력 명세로 표현하여 전단부 컴파일러 구축에 결합하여 자동화 도구의 생성 능력을 향상시킨다. 전단부 컴파일러 자동화 도구의 높은 활용성은 이미 텍스트 프로그래밍 언어에서 입증된 것으로 시각언어의 경우는 아직 후단부에 대한 형식적인 기술 방법이 거의 없으므로 의미 수행 코드를 결합한 문법-지시적 변환 방법에 의한 컴파일러의 자동 생성은 보다 그 효과 및 활용 가능성이 높은 방법이다[6].

기존 시각언어의 컴파일러에 대한 연구 및 방법들도 이러한 장점을 적용하여 자동화 도구를 개발하고자 하였으나, *yacc*과 같은 도구 개발에 관심을 가지고 연구된 VisPro도 동일한 시각언어에 대한 다양한 도구 개발에 적용할 수 있는 활용성과 유지보수 측면에서 볼 때, 새로운 내용의 추가 확장시 수정에 어려움이 있다는 문제점을 내포하고 있다.

본 연구에서는 Pictorial Class Grammar(PCG)[2,3,8]의 시각언어 구문 정의을 캡슐화 하는 방법과 And-Or-Waiting(AOW) Graph[2,3,8]를 이용한 구문 분석 방법을 기반으로 하여, 구문 요소를 하나의 클래스 형태로 구성함으로써, 구문 요소의 표현에 사용되는 의미와 구문의미(syntax-semantics)를 분리할 수 있는 방법을 제시한다. 이 방법에 기초하여 기존 연구들의 문법-지시적 시각언어 컴파일러 구축의 문제점을 극복하고, 구문 명세와 분리된 의미 수행 코드 명세 방법을 제시하여, 유지보수성을 보다 향상시킨 문법-지시적 변환 방법을 이용한 시각언어 컴파일러 자동화 도구인 *VisCC*를 개발하였다.

2. 시각언어 컴파일러-컴파일러: VisCC의 개요

본 논문에서 제시하는 시각언어 컴파일러 자동화 도구 ***VisCC***(***Visual*** Language ***C***ompiler-***C***omplier)는 시각언어의 그래픽 객체와 구문, 의미 수행 코드에 대한 문법 정의로부터 객체지향 프로그래밍 언어로 작성된 컴파일러 프로그램을 자동 생성하는 문법 기반 시각언어 자동 생성 환경이다.

언어 설계자는 언어의 그래픽 객체, 구문, 의미를 정의하는데 편리한 PCG에 기반을 둔 명세언어와 의미 수행 코드를 이용하여 컴파일러를 기술하고, *VisCC*내의 각 구성요소를 이용하여 임의의 시각언

어를 구현할 수 있다. 이러한 *VisCC*는 크게 윈도우 기반 사용자 인터페이스와 PCG를 기반으로 하는 컴파일러 자동 생성기로 구성된다.

*VisCC*의 가장 핵심적인 부분은 컴파일러 생성기이다. 이는 PCG에 기반을 둔 고급 명세언어로 작성된 대상 시각언어의 입력 명세를 받아들여 어휘 분석기를 생성하는 어휘분석기 생성기(Lexical Analyzer Generator)와 구문 분석기를 생성하는 파서 생성기(Parser Generator), 의미 수행 코드를 생성하는 의미 수행 코드 생성기(Action Generator)로 구성된다. *VisCC*의 개략적 구성도는 그림 1과 같다.

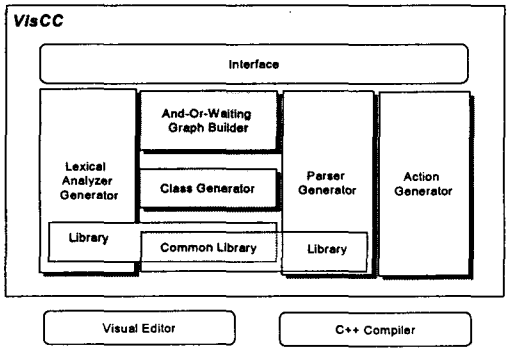


그림 1. *VisCC*와 주변 요소의 개략적 구성도

*VisCC*는 기존 시스템과 비교하여 몇 가지 중요한 특성을 가진다. PCG를 기반으로 한 명세 방법을 사용함으로써 시각언어의 구문 표현과 정적 의미 형태로 표현되는 구문 정의 요소를 캡슐화하여 단일 표현으로 정의하는 방법을 제시한다. 이를 통해 사용자 정의 그래픽 객체 정의를 이용한 어휘 분석 단계의 활용 효과를 얻는다. 또한 이를 기반으로 유도되는 AOW Graph를 이용한 구문 분석 방법은 언어 의존적인 구문 분석 정보와 독립적인 부분의 분리를 통해 체계적인 구문 분석 방법을 제공한다. 형식적 명세에서 자동으로 생성되는 과정이 이 모델링 구조를 기반으로 함으로써, 직접적으로 파서를 생성하는 방법에 비해 보다 효과적으로 처리된다. 이러한 클래스 개념에 기반을 둔 문법과 구문 분석 방법은 의미 수행 코드와의 결합을 통한 문법-지시적 변환 방법의 기초를 제공하여 효과적인 의미 분석 단계의 처리 방법을 제시한다.

3. 시각언어 컴파일러-컴파일러의 자동 생성 방법

컴파일러 자동 생성기 개발시 반드시 필요한 두 가지 요소는 자동 생성 가능한 파싱 알고리즘과 대상 시각언어를 형식적으로 정의할 수 있는 명세 방법이다. 그러나 이 두 가지 요소를 충족하는 것만으로는 시각언어 컴파일러 후단부의 자동화 문제를 해결하기 위한 의미 수행 코드 결합을 통한 문법-지시적 컴파일러 개발 환경을 자동 생성하는데는 어려움이 있다. 앞에서 언급한 바와 같이 시각언어의 구문표현에 있어서 사용되는 의미 표현이 구문요소와 결합되어 하나의 구문을 표현하는 시각언어에서는 이들을 하나의 구문단위로 표현할 수 있는 방법이 형식 명세에서도 필요하고, 이를 기반으로 한 코드의 자동 생성 및 의미 수행 코드와의 결합 방식에서도 필요하다. 본 연구에서 사용하는 형식적인 명세 방법인 PCG는 구문을 단위 요소인 클래스로 정의하는 방법을 제시하고 있다.

```
<Expr_Tree> ::= <lhs:Variable><rhs:Child>
{  print(lhs.value);
   when     touches_R(lhs,rhs);      }
<Node>:<Rectangle>::=<rect:Rectangle><str:Text>
{   Syn value;
    Node.value = 0;
    when    contains(rect,str);      }
<Subtree>:<Node> ::= <plus:Plusnode>
{                                    }
<Subtree>:<Node> ::= <minus:Minusnode>
{                                    }
<Const>:<Node> := Case
{  Const.value = stoi(str.string)
   when     is_integer(str.sval)     }
<Binop>:<Node>:=<left:Child><op:Node><right:Child>
{  Syn opcode : char;
   Binop.opcode = op.sval;
   when    touches_R(left,op) && touches_R(right,op); }
<Plusnode>:<Binop> := Case
{  Plusnode.value = left.value + right.value;
   when     strcmp(op.sval, "+");    }
          .....
```

그림 2. ETL의 문법 정의

PCG는 속성 다중집합 문법과 객체지향 패러다임에 기반을 둔 형식적 명세 방법이다. 이 문법은 시각 구문과 정적 의미로 표현되는 하나의 시각 구문 정의를 캡슐화시켜 하나의 단위로 표현할 수 있는 명세 방법을 제공한다. 그 결과 기존의 시각언어 문법 및 파싱 방법과는 달리 시각언어 구문 분석 관계를 개념적으로 모델링 할 수 있는 구조를 정의할 수 있다. 모델링 구조인 AOW Graph는 자신의 인스턴스로 표현되는 구문 분석 트리를 유도함으로써 문법-지시적 변환 방법을 이용한 컴파일러 구축 과정의 기반 기술

을 제공한다. 그림 2는 Expression Tree Languages (ETL)라는 시각언어의 구문을 PCG로 정의한 예의 일부분이다.

PCG에 기초하여 설계되는 구문 분석기는 대상 시각언어에 독립적인 파싱 드라이버 루틴과 대상 시각언어에 따라 다르게 구축하는 AOW Graph로 구성된다. 이러한 특성을 가지는 구문 분석기는 AOW Graph를 대상 시각언어를 정의한 문법으로부터 자동 생성하고, 파싱 드라이버 루틴과 결합하여 자동 생성할 수 있다. 또한 구문 분석과정에서 결합되어 수행될 의미 수행 코드를 의미 수행 코드 정의로부터 생성하여 관련된 구문 분석 과정에서 수행할 수 있도록 한다.

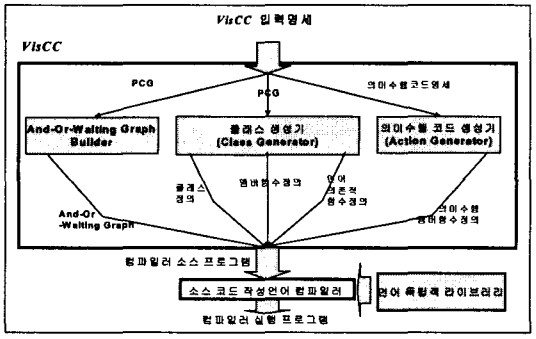


그림 3. *VisCC*의 구성요소 및 입출력 관계

그림 3은 입력명세를 받아들여 컴파일러의 각 요소들을 자동 생성하는 생성기의 주요 요소들과 의미수행 코드의 결합 관계를 각 부분의 입출력 관점에서 도식화 한 것이다.

3.1 And-Or-Waiting Graph 자동 생성 방법

AOW Graph는 시각언어의 구문을 정의한 문법상의 클래스간의 의존관계를 나타낸 구문 분석 모델링 구조로 본 연구에서 제안하는 파싱 방법의 핵심 요소이다. 이러한 AOW Graph는 PCG로 정의한 대상 시각언어의 구문 정의로부터 자동 생성할 수 있다.

3.1.1 And-Or-Waiting Graph의 표현

AOW Graph는 각 클래스를 나타내는 노드와 클래스간의 구문 분석 의존 관계를 나타내는 에지로 구성된다. AOW Graph를 표현하기 위해서는 그래프상의 모든 노드와 에지를 표현하는 방법이 필요하다. 본 연구에서는 AOW Graph를 클래스와 그래프 상의 에지가 나타내는 자신의 propagation class를 이용하여 정의한다. 먼저, 하나의 클래스 관점에서 하나의 에지는 자신과 자신의 propagation class와의 의존관계이다. 하나의 클래스와 자신의 propagation class로 구성된 서브 그래프는 인접 리스트 형태로 표현한다. 그래프의 모든 클래스는 하나의 배열로 구성하고, 에지는 각 클래스와 자신의 propagation class들로 구성된 리스트로 표현한다. 그림 4는 ETL의 AOW Graph의 일부분을 표현한 예이다.

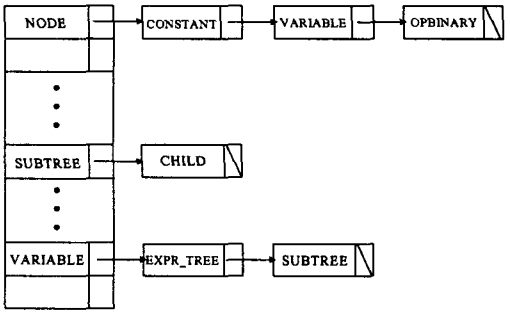


그림 4. ETL의 And-Or-Waiting Graph를 표현한 인접 리스트의 일부분

이 인접 리스트의 표현 형태는 AOW Graph의 의존관계를 나타내는 에지를 효과적으로 표현한다. 그러나 실질적인 구문 분석을 위해서는 이들 각 클래스의 인스턴스 생성을 유도하는 과정과 그래프의 각 노드를 표현하는 방법이 필요하다. 각 노드는 복합적인 구문 정보의 통합체이다. 이들은 클래스 정의를 통해 구현된다. 이에 대한 자세한 설명은 3.2절에서 기술한다.

3.1.2 And-Or-Waiting Graph의 자동 생성 방법

AOW Graph의 자동 생성은 입력된 문법으로부터 자동 생성에 필요한 정보를 추출하여 클래스간의 의존 관계를 형성하는 것이다. 따라서, AOW Graph를 자동 생성하는 과정은 생성규칙의 rewrite rule의 클래스 정보로부터 propagation class를 찾아 자신의 propagation class의 리스트에 추가하는 과정을 반복하여 정의한다.

각 클래스의 propagation class는 클래스의 종류에 따라 다르다. 문법에서 클래스 <Node>에 대한 생성규칙의 rewrite rule이 다음과 같으면, 클래스 <Node>는 구조클래스이다.

<Node>:<Rectangle>::=<rect:Rectangle><str:Text>

이 생성규칙으로부터 문법상의 클래스 의존 관계에 따라 그림 5와 같은 그래프의 일부분을 자동 생성하기 위해서는 rewrite rule의 모든 클래스간의 관계를 사용한다. 즉 클래스 <Rectangle>과 클래스 <Text>의 propagation class의 집합에는 클래스 <Node>를 포함시킨다.

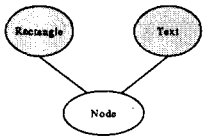


그림 5. And-Or-Waiting Graph 예의 일부분(1)

사례클래스인 클래스 <Constant>에 대한 생성규칙의 rewrite rule은 다음과 같이 정의된다.

<Constant>:<Node>::= Case

이 생성규칙으로부터 문법의 클래스 의존 관계에 따라 그림 6과 같은 그래프의 일부분을 자동 생성하기 위해서는 클래스와 상위클래스의 관계를 사용한다. 이는 사례클래스인 경우는 자신의 구문 구조를 상위클래스로부터 상속받아 사용하기 때문이다. 즉, LHS인 클래스 <Constant>를 클래스 <Node>의 propagation class 리스트에 추가하여 두 노드 클래스 <Constant>와 클래스 <Node>의 의존 관계를 생성한다. 그림 7은 AOW Graph의 자동 생성 알고리즘이다.

3.2 클래스 자동 생성 방법

AOW Graph의 각 노드는 대상 시각언어의 구문 구조를 정의하는 중간 상태를 표현한다. 따라서 하나의 노드는 자신의 구문 구조를 정의하기 위해서 사용하는 하위 구문 구조와 이 들간의 관계를 표현하는 제약조건 처리 부분을 하나로 캡슐화하여 클래스로 표현한다. 구문 분석기를 자동 생성하기 위해서는

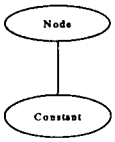


그림 6. And-Or-Waiting Graph 예의 일부분(2)

```
Algorithm And-Or-Waiting Graph Construction
 begin
  repeat
   get a production;
   repeat
    get a token;
    case (a kind of token)
     lhs:make new node for lhs;
   rhs:case (a kind of rhs)
         class lists:repeat
                     get a token;
                      case (a kind of token)
                       rhs:link lhs's node to rhs's propagation list;
                      end case;
                     until class lists;
           CASE : find its super class;
                  link lhs's node to super class node's propagation list;
         end case;
    end case;
   until End of a production;
  until End of file;
 end.
```

그림 7. And-Or-Waiting Graph 자동 생성 알고리즘

AOW Graph의 각 노드를 나타내는 클래스 정의를 입력 명세의 생성규칙 정의로부터 자동 생성해야만 한다.

3.2.1 클래스

3.2.1.1 클래스 정의

AOW Graph는 구문 분석을 위한 정적인 의존 관계에서 각 노드가 가지는 정보와 구문 분석 행동을 하나의 클래스로서 정의한다. 하나의 클래스 정의는 기본적으로 클래스의 구문 분석 정보를 표현하는 멤버 변수와 구문 분석 행동을 정의하는 메소드로 구성된다. 각 노드를 정의하기 위한 클래스 정의는 자신의 waiting class를 위한 멤버 변수와 working object, complete object, 파싱 아이템을 생성하는 일련의 구문 분석 과정을 수행하기 위한 메소드로 구성한다. 이 절에서는 멤버 변수에 대해 설명하고, 메소드에 대해서는 다음절에서 기술한다.

클래스의 멤버 변수는 크게 구문 구조 멤버 변수, 상태 정보 멤버 변수, 속성 정보 멤버 변수의 3가지 유형으로 나누어진다. 첫 번째 유형인 구문 구조 멤버 변수는 자신의 구문 구조 생성을 위해 사용되는 waiting class들을 멤버 변수로 선언함으로써 클래스의 구문 구조를 생성하는 하위 구조를 표현한다. 각 클래스에서 하위 구조는 자신의 waiting class들의 인스턴스 그래프이다. 따라서 모든 클래스는 자신의 waiting class 유형의 객체들을 멤버 변수로 가진다.

클래스의 waiting class들은 클래스의 유형에 따

라 성격을 달리한다. 먼저, And Class인 경우는 자신의 waiting class들이 자신의 구문 구조 생성을 위해 필수 불가결한 요소로 이 들 각각을 멤버 변수로 정의한다. 예를 들어 위의 클래스 <Node>의 경우는 RECTANGELCls *rect와 TEXTCls *str의 두 멤버 변수를 가진다. Or Class인 경우는 And Class보다는 복잡한 자료구조를 가진다. Or Class의 waiting class들은 클래스의 구문 구조 생성을 위해 그 들 중 하나만 자신의 인스턴스를 가지면 된다. 따라서 본 연구에서 Or class는 자신의 멤버 변수로 자신의 waiting class들을 필드(field)로 가지는 공용체(union structure)를 이용하여 표현한다. 예를 들어 Or Class인 <SUBTREE> 클래스의 waiting class들은 <CONSTANT>를 비롯한 6개의 클래스들이다. 이를 처리할 수 있는 멤버 변수는 다음과 같은 공용체로 표현되고, 클래스 <SUBTREE>는 SUBTREE_or 유형의 구문 구조 멤버 변수 SUB를 가진다.

```
union SUBTREE_or {
        CONSTANTCls *cons;
        VARIABLECls *var;
        PLUSNODECls *plus;
        MINUSNODECls *minus;
        TIMESNODECls *times;
        DIVIDENODECls *divide;
};
```

두 번째 유형은 상태 정보 멤버 변수로 클래스로부터 사례화된 객체로 표현되는 working object, complete object, 파싱 아이템을 생성하는 일련의 구문 분석 단계를 수행하는 과정에서 객체의 상태를 저장하기 위해서 사용한다.

마지막 유형은 속성 정보 멤버 변수로 생성규칙 정의시 사용되는 속성을 선언한다. 명세언어는 대상 시각언어의 구문을 정의하는 생성규칙이 자신의 속성을 정의하여 사용하는 것을 허용한다. 이는 일반적으로 관련 클래스의 의미 함수나 제약 조건 등에서 사용되고, 필요에 따라 정적 의미 표현을 위한 값을 저장하기 위해서 사용된다. 속성 정보 멤버 변수는 이를 실질적인 클래스 정의에 표현하는 방법이다.

예를 들어, 클래스 <CONSTANT>는 정적 의미로 입력되는 상수 값을 구문 분석 과정에서 유지하고자 한다면 생성규칙 정의시 int 유형의 속성 value를 정의하여 값을 저장할 수 있다. 이 때 이 속성은 생성된 파서에서는 멤버 변수 int value;로 선언되어 사용된다.

3.2.1.2 클래스 메소드 선언

텍스트 프로그래밍 언어의 구문 분석과 비교하여 기술하면, AOW Graph는 파싱 테이블처럼 구문 분석시 입력 정보의 종류와 현재의 상태에 따라 결정하는 다음 상태의 정적인 관계를 포함하는 구조로 효과적인 구문 분석기의 설계를 유도한다. 그러나 파싱 테이블과는 달리 입력과 현재 상태에 따른 구문 분석 행동은 그래프 자체에 포함되어 있지 않다. 왜냐하면, 동일한 정적 의존 관계에서 발생하는 동적인 구문 분석 행동이 하나 이상이고, 실질적인 시각 프로그램에 따라 다른 동적 행동이 결정되기 때문이다. 따라서 AOW Graph의 각 노드에서 발생하는 구문 분석 행동에 대한 정의가 필요하다. 이러한 구문 분석 행동은 각 노드에 해당하는 생성규칙의 클래스로부터 생성되는 메소드로서 클래스 정의부분에 기술한다.

3.2.1.3 클래스 메소드 정의

구문 분석기의 각 클래스에서 구문 분석 행동을 표현하기 위해 정의되는 클래스 메소드는 크게 객체 생성 메소드, backtracking 메소드, 제약조건 메소드, 의미함수 메소드의 4가지로 구분한다.

객체 생성 메소드는 각 클래스의 인스턴스인 프로그램 인스턴스 그래프 생성을 위한 가장 핵심적인 메소드이다. 객체 생성 메소드는 자신의 waiting class의 인스턴스를 생성하고, 이 인스턴스가 파싱 아이템이면 waiting class로부터 이 파싱 아이템을 전달받아 자신의 인스턴스를 생성한다. 즉, 자신의 구문 구조를 유도한다. 이러한 객체 생성 메소드는 두 가지 요소를 인자로 정의한다. 하나는 자신의 waiting class의 객체이고, 또 다른 하나는 생성된 객체를 하위 구조로 사용하는 해당 클래스의 working object이다. 따라서 객체 생성 메소드는 자신의 waiting class의 수와 동일한 수만큼 정의된다. 예를 들어, 클래스 <Node>의 waiting class는 클래스 <Rectangle>과 클래스 <Text>이다. 따라서 클래스 <Node> 정의는 두 개의 객체 생성 메소드 정의를 유도한다.

Backtracking 메소드는 클래스 인스턴스 생성과

정에서 사용된 하위 구조가 최종 구문 분석 과정에서 파싱 아이템 생성에 실패한 경우, 사용된 하위 구조의 재사용을 위한 메소드이다. 시각언어에서는 기본적으로 구문 분석시 입력된 정보에 대한 순서의 개념이 제거됨으로서 한 번 읽혀진 토큰이 현재 진행 중인 구문 분석 과정에서 complete object 생성에 사용되었지만 최종적인 파싱 아이템 생성에서 사용되지 않을 수 있다. 이 때 사용된 waiting class의 인스턴스는 다시 다른 부분에서 다른 인스턴스에 사용하여 최종적인 구문 분석까지 적용해야 한다. 따라서 인스턴스 생성과정에서 현재까지 진행된 유도과정에 생성규칙이 잘못 적용되었다면, 다른 생성규칙의 적용을 통한 유도를 수행하기 위해 사용된 입력 정보의 상태를 다른 상황에서 반복 사용할 수 있도록 해야한다. 본 연구에서는 전체적인 backtracking 방법이 아닌 자신의 상태 정보 멤버 변수를 이용한 이 전 단계로의 부분적 backtracking 방법을 사용한다. 이를 위해 각 클래스 정의는 backtracking을 처리할 자신의 메소드를 정의한다. 예를 들어, 클래스 <Node>는 클래스 <Rectangle>과 클래스 <Text>의 인스턴스를 하위 구조로 하는 인스턴스를 생성해야한다. 그러나 이 두 클래스의 인스턴스가 생성된 상태는 complete object의 상태이다. 따라서 이 complete object가 파싱 아이템인가를 검사하기 위해 제약조건인 포함관계(*contains*)를 검사한다. 검사 결과, 이 객체의 상태가 파싱 아이템이 될 수 없다면, 사용하였던 클래스 <Rectangle>과 클래스 <Text>의 각 인스턴스는 다른 인스턴스와 결합할 수 있도록 backtracking 한다.

제약조건 메소드는 클래스의 인스턴스로 생성된 complete object를 입력으로 받아 제약조건을 검사하고, 결과에 따라 파싱 아이템을 생성할지 backtracking을 수행할 지를 결정하게 된다. 의미함수 메소드는 멤버 변수로 정의된 속성 값을 정의하기 위한 메소드이다.

3.2.2 클래스 자동 생성 방법

클래스를 자동 생성하는 과정은 크게 두 단계로 구분한다. 첫 번째 단계는 생성규칙의 rewrite rule의 클래스 정보를 이용하여 AOW Graph의 각 노드인 생성규칙의 클래스에 해당하는 클래스 정의를 자동 생성하는 것이고, 두 번째 단계는 클래스가 포함하는 각 노드 상태의 구문 분석 메소드를 자동 생성하는 것이다.

클래스 정의는 다음과 같은 단계로 자동 생성한다. 먼저, 문법으로부터 하나의 생성규칙을 가져온다. 생성규칙의 rewrite rule에서 LHS의 이름에 해당하는 클래스 정의 헤딩(heading)을 구성한다. 이 때, LHS의 상위 클래스는 생성되는 클래스 정의에서 상위클래스로 정의한다. 다음 LHS가 And Class이면, RHS의 모든 클래스를 구문 구조 멤버 변수로 만드는 클래스 선언문을 구성한다. 이 때 RHS의 클래스 정의에서 사용된 태그는 멤버 변수의 이름으로 사용한다. 만약 LHS가 Or Class이면, 3.2.1.1절에서 기술한 바와 같이 모든 waiting class들을 구성요소로 하는 공용체를 이용하여 구문 구조 멤버 변수 선언문을 구성한다. 상태 정보 멤버 변수는 문법으로부터의 정보를 이용하여 생성하는 것이 아니므로, 필요한 변수 선언문을 자동으로 추가한다. 속성 정보 멤버 변수는 클래스의 속성 선언문이 있는 경우 동일한 이름의 변수 선언문을 구성한다.

클래스 정의 자동 생성 단계에서 멤버 변수 선언문의 자동 생성이 완료되면, 각 클래스의 구문 분석 행동을 정의하여 메소드로 사용하기 위해 3.2.1.2의 각 구문 분석 메소드의 선언문을 클래스 정의에 추가하여 구성한다.

객체 생성 메소드는 생성된 파싱 아이템에 따라 서로 다른 함수가 호출된다. 이 때 호출하는 구문 분석 프로그램의 입장에서 만약 waiting class의 종류에 따라 매번 서로 다른 프로그램을 호출해야 한다면, 이는 자동 생성시 언어 의존적으로 생성되는 부분이 복잡해짐을 의미한다. 본 연구에서는 객체지향 프로그래밍 언어의 중복지정 방법을 이용하여 효과적인 코드를 자동 생성한다. 하나의 클래스에는 클래스의 waiting class 수 만큼에 해당하는 객체 생성 메소드가 클래스 정의에 선언되고, 이 메소드를 정의하는 프로그램이 생성된다. 이러한 방법은 이 메소드를 호출하는 파싱 드라이브 프로그램 관점에서는 객체 생성시 어떠한 waiting class의 파싱 아이템을 전달하는가에 상관없이 동일한 호출 방법을 사용할 수 있으므로 대상 시각언어에 독립적인 방법을 제공한다.

다음에 생성하는 메소드는 backtracking 메소드이다. Backtracking 메소드는 생성된 complete object가 파싱 아이템이 아닌 경우 사용된 waiting class의 객체를 다음 단계에 재사용할 수 있도록 처리하는 함수이다. 다음 단계는 생성규칙내의 제약조

건을 처리하는 메소드를 선언하고 정의한다. 이는 if 문을 사용하여 제약조건에 나타난 조건이 참이면, 참임을 반환하고, 거짓이면 거짓임을 반환하는 프로그램을 생성하여 정의한다. 의미 함수 메소드는 생성규칙의 의미 함수 정의문을 생성 코드 작성언어의 명령문으로 변환하여 하나의 함수 형태의 프로그램으로 자동 생성된다.

이와 같이 자동 생성되는 메소드들은 객체 생성 메소드를 제외하고는 각각 하나의 클래스를 정의할 때 한번 선언되고, 모든 클래스 정의에서 동일한 이름으로 생성된다. 이와 같이 동일한 이름을 사용함으로써 언어 독립적 메소드인 파서 드라이브 프로그램에서 대상 시각언어와 상관없이 각 메소드를 호출하여 구문 분석을 수행할 수 있다. 그림 8은 클래스 생성 절차이다.

```
Algorithm Class Generation
 begin
  repeat
   get a production;
   repeat
    get a token;
    case (a kind of token)
     lhs:define a new class;
    rhs:declare a member variable;
        declare and define a makeobject function using AOW Graph;
    declaration statement:declare a member variable;
     assignment statement:declare a semantic function and define it;
     constraint:declare a constraint_check function and define it;
    end case;
   until End of a production;
   declare and define a backtracking function using AOW Graph;
  until End of file;
 end.
```

그림 8. 클래스 생성 절차

3.3 의미 수행 코드의 명세 및 자동 생성 방법

의미 수행 코드를 이용한 문법-지시적 변환 방법은 가장 일반적인 전단부 컴파일러 생성기의 자동 생성 방법이다. 이 방법은 실제 시스템 프로그래밍 과정에서 사용자에게 융통성을 허용하는 효과적인 개발 방법을 제공한다.

본 논문에서도 의미 수행 코드를 기반으로 한 컴파일러 자동 생성 방법을 사용한다. 그러나 의미 수행 코드를 이용한 의미 처리 방법은 기존의 방법과는 차별되는 특성을 가진다. 첫 번째 특성은 의미 명세의 분리이다. 기존의 방법들은 의미 명세를 구문 명세와 결합하여 명세한다. 그러나 본 연구에서는 구문 명세와 의미 명세를 분리해서 작성하고 분리된 생성기를 통해 구문 분석기와 의미 수행 코드를 생성한다. 이 방법은 구문 분석기와 의미 수행 코드 생성 과정의 변경 및 확장 등의 유지 보수 측면에 효과적인 방법을 제공한다. 두 번째 특성은 의미 명세 과정에서 구문 정보를 제외한 정적 의미의 속성을 이용한 명세와 프로시쥬어 방식의 동적 의미 명세 표현을 하나의 방식으로 표현할 수 있는 방법을 통해 다양한 의미 표현 방식을 제공하는 것이다.

3.3.1 의미 수행 코드의 정의

본 연구에서 정의하는 의미 수행 코드의 입력 명세는 구문 명세의 클래스에 의존적으로 정의된다. PCG에 기반을 둔 구문 명세를 통해 각 생성규칙은 실질적으로 하나의 클래스로 정의할 수 있다. 따라서, 각 생성 규칙의 의미 수행 코드는 결과적으로 각 클래스의 의미 수행 코드를 나타내는 하나의 메소드로 처리한다. 정의된 의미 수행 코드는 실질적으로 각각 하나의 클래스가 자신의 인스턴스인 파싱 아이템을 생성하였을 때 수행된다. 즉, 파싱 드라이버는 하나의 클래스가 자신의 파싱 아이템을 생성했을 때 그 클래스의 의미 수행 메소드를 수행한다.

의미 수행 코드의 변환은 구문 명세에 의존하여 수행되므로 명세내용은 구문 명세의 클래스 정의와 상관 관계가 있다. 의미 수행 코드 명세는 이미 구문 구조가 생성된 후 수행함으로 명세 과정에서 구문 구조를 표현하는 정의는 필요하지 않다. 단지 해당 클래스의 이름과 이 클래스에 대한 의미 수행 코드만이 필요하다. 의미 수행 코드 생성기는 클래스의 이름을 공통키로 하여 구문 명세와 다른 파일에 명세한 의미 수행 코드를 해당 클래스의 메소드로 정의한다. 의미 수행 코드 명세시 이름은 반드시 구문 명세에 정의된 클래스의 이름이어야 하고, 의미 수행 코드는 생성되는 컴파일러 소스 프로그램의 작성 언어와 동일한 객체지향 프로그래밍 언어로 작성한다.

의미 수행 코드 정의에서 기술해야 하는 입력 명세의 내용은 크게 클래스 선언 부분, 의미 정의 부분, 사용자 프로그램 정의 부분의 세 부분이다.

클래스 선언 부분은 생략할 수 없는 부분으로 대상 시각언어의 구문 명세에서 사용한 클래스들의 이름을 선언한다. CLASS라는 키워드를 사용하고, 선언하는 클래스의 이름들을 키워드 뒤에 나열한다.

의미 정의 부분은 반드시 선언해야 하는 부분으로 구문 분석기 정의에서 명세된 각 클래스의 의미 수행 코드를 정의하는 부분이다. 각 클래스의 의미 정의는

클래스의 이름과 의미 수행 코드의 부분으로 구분된다. 클래스의 이름을 선언하는 부분은 클래스의 유형에 따라 차이가 있다. And class인 경우는 그 클래스에 대한 구문 정의가 한 번 존재하고, 따라서 의미 수행 코드 명세도 한 번 정의한다. 따라서 클래스의 이름만으로 의미 수행 코드 명세를 할 수 있다. 그러나 Or class인 경우는 동일한 클래스의 이름으로 정의해야 하는 의미 수행 코드 명세가 하나 이상이다. 따라서 이 들을 구별할 수 있는 명세 방법이 필요하다. 본 연구에서는 waiting class의 이름을 이용한다. 즉, <or-classname>:<waitingclassname>의 형식으로 클래스의 이름을 정의한다. 예를 들어 클래스 <SUBTREE>는 Or class로 자신의 waiting class는 <CONSTANT>, <PLUSNODE> 등 6개이다. 따라서 <SUBTREE>에 대한 의미 수행 코드 정의가 <SUBTREE>:<CONSTANT>와 같은 형태로 구별하여 6번 명세된다.

각 클래스의 의미 수행 코드 정의 부분은 다시 세 부분으로 구분된다. 첫 번째 부분은 속성 정의 부분이다. 속성 정의 부분에서 정의되는 속성은 정적 의미 형태로 정의되는 속성 값을 의미 수행 코드에서 사용할 수 있도록 한다. 두 번째 부분은 의미 수행 코드 명세에서 사용하는 변수를 선언하는 부분이다. 이는 생성 코드 정의에 사용하는 범용 객체지향 프로그래밍 언어의 변수 선언 규칙에 따라 정의한다. 마지막 부분은 의미 수행 코드 부분으로 범용 객체지향 프로그래밍 언어로 작성한다. 범용 객체지향 프로그래밍 언어로 작성한 부분은 자동 생성기에서 변환이 일어나지 않는다. 즉, 작성된 코드는 변환 작업이 없이 생성된 컴파일러의 의미 수행 코드로 사용하므로, 구문 상의 오류가 있다할지라도 의미 분석기 생성기에서 오류 검사를 하지 않는다. 오류가 있다면 시각언어 컴파일러 소스 프로그램을 생성한 후 실행 프로그램을 만드는 과정에서 검사된다.

사용자 프로그램 정의 부분은 위의 두 부분과는 달리 선택적 정의부분이다. 의미 정의 부분에서 각 클래스의 의미 수행 코드를 정의하면서 사용자 정의 프로그램을 사용한다면 반드시 이 부분에서 사용자 정의 프로그램을 정의해야만 한다.

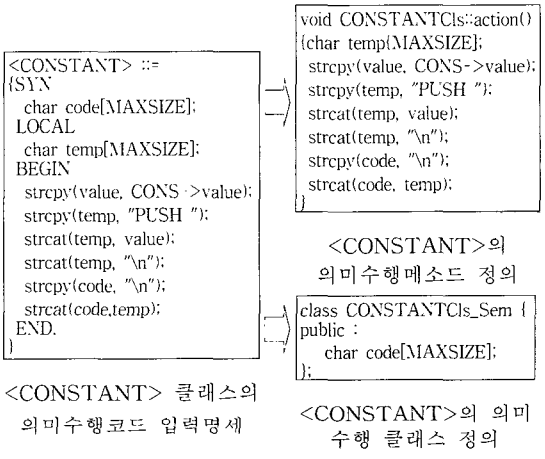
그림 9는 ETL의 컴파일러를 구현하기 위한 의미 수행 코드 정의의 일부분이다. 언어의 의미는 Expression Tree 프로그램을 간단한 스택 기계를 위한 프로그램으로 번역함으로써 정의한다.

3.3.2 의미 수행 코드 생성기

의미 수행 코드 생성기는 의미 수행 코드 입력 명세를 받아 들여 두 개의 프로그램을 생성한다. 하나는 각 의미 수행 코드 입력 명세에서 사용하는 속성과 변수를 멤버 변수로 하는 클래스 정의이고, 나머지 하나는 입력 명세상의 절차적 형태로 작성된 프로그램 코드로부터 구성된 의미 수행 메소드 정의이다.

```
<CONSTANT> ::=  {SYN
                     char code[MAXSIZE];
                   LOCAL
                     char temp[MAXSIZE];
                   BEGIN
                     strcpy(value, CONS->value);
                     strcpy(temp, "PUSH ");
                     strcat(temp, value);
                     strcat(temp,"\n");
                     strcpy(code, "\n");
                     strcat(code, temp);
                   END.
                 }
                 ..............
<SUBTREE>:<TIMESNODE> ::=
                 {SYN
                   char code[MAXSIZE];
                  BEGIN
                   strcpy(value, SUB.times->value);
                   strcpy(code, "\n");
                   strcpy(code, SUB.times->code);
                  END.
                 }
<SUBTREE>:<DIVIDENODE> ::=
                 { SYN
                    char code[MAXSIZE];
                   BEGIN
                    strcpy(value, SUB.divide->value);
                    strcpy(code, "\n");
                    strcpy(code, SUB.divide->code);
                   END.
                 }
%%
```

그림 9. ETL의 의미 수행 코드 생성기 입력명세 예의 일부분



생성된 클래스 정의는 해당 클래스의 상위 클래스로 정의함으로써 이 클래스의 모든 변수들이 해당 클래스 정의에서도 사용 가능하도록 한다. 의미 수행 메소드는 모든 클래스에 대해 동일한 이름으로 정의한다. 이에 대한 클래스 메소드 선언문은 구문 분석기 생성기 내의 클래스 생성기에 의해 선언된다. 예를 들어, 클래스 <CONSTANT>에 대한 의미 수행 코드 명세가 다음과 같을 때, 의미 수행 코드 생성기는 다음과 같은 부분의 프로그램을 생성한다.

입력 명세에서 기술된 의미 수행 코드는 action() 메소드의 코드로 기술되고 지역변수 temp는 action() 메소드의 지역변수로 선언된다. 다음 속성으로 기술된 code를 멤버변수로 하는 클래스 정의를 생성한다. 생성되는 클래스의 이름은 입력명세의 클래스 이름에 "_Sem"이 추가된 형태 즉 "CONSTANTCls_Sem"로 정의되고, 이 클래스는 CONSTANTCls 클래스의 상위 클래스로 정의된다. 이와 같은 기능을 수행하는 의미 수행 코드 생성 알고리즘은 그림 10과 같다.

```
Algorithm Action Generator
 begin
  repeat
   get a production;
   repeat
    case (class name)
     declaration statement;
      case (kind)
        SYN:declare a member variable;
        LOCAL:declare a local variable of action function;
      end case;
     description statement;
      define a member function whose name is action();
   end case;
  until End of a production;
 until End of file;
end.
```

그림 10. 의미 수행 코드 생성 알고리즘

3.3.3 의미 수행 코드 생성기와 유지보수성 향상

본 연구에서 제시하는 자동 생성 방법은 의미 수행 코드의 명세 및 생성을 구문 분석기 생성 단계와 분리한다. 이는 기존의 자동화 도구에서 의미 수행 코드 명세가 구문 분석기 생성기의 명세와 함께 기술되고 생성하는 방법과 비교하여 볼 때 재사용 및 유지 보수성의 향상을 유도한다. 이러한 의미 수행 코드의 분리 명세 및 생성을 통한 효과를 구체적으로 살펴보면 다음과 같다.

첫째, 동일 시각언어의 언어 자체의 확장이 아닌 컴파일러를 실행할 대상 기계의 언어가 다른 경우 즉, 다른 시스템인 경우 새로운 컴파일러 후단부의 변경이 필요하다. 예를 들어, 생성할 어셈블리 코드가 ADD에서 ADD A B와 같은 형태의 코드로 변경되었다면, 이는 어휘 분석 단계나 구문 분석 단계인 컴파일러의 전단부와는 관계가 없는 후단부 즉, 의미 수행 코드 부분만 변경함을 의미한다. 이 때 의미 수행 코드 명세가 구문 분석기 명세와 결합되어 있다면, 구문 분석기에는 변화가 없음에도 불구하고, 구문 분석기를 다시 생성하는 작업과정이 진행되어 많은 시간과 자원이 소모된다. 그러나 본 연구에서 제시하는 클래스를 기반으로 한 의미 수행 코드 명세 및 생성 방법은 의미 수행 코드 생성기의 입력 명세 부분만의 재작성과 재생성 후 기존 구문 분석기 생성물과 결합 수행할 수 있다. 따라서 컴파일러 후단부의 수정 및 변경시 기계 의존적 측면의 유지 보수성을 향상시킨다.

둘째, 동일 시간언어의 다양한 도구 개발시 구문 분석기 생성 단계를 수행할 필요없이 도구의 의미와 관련된 코드 생성에서 해결할 수 있다. 예를 들어 대상 시각언어의 컴파일러 개발 후 문법-지시적 편집기나 인터프리터 등 다른 도구를 개발할 때 어휘 및 구문의 변화가 없으면, 구문에 따라 수행되는 의미 수행 코드의 내용을 개발자의 의도대로 작성한 의미 수행 코드 명세의 작성 및 생성만으로 다양한 도구 개발에 활용할 수 있다.

4. 구현 및 적용사례

*VisCC*는 Sun 1000에서 *lex*와 *yacc*을 이용하여 C 언어로 구현한다. *VisCC*에 의해서 자동 생성되는 컴파일러 프로그램의 작성 언어는 C++언어를 사용한다. 사용자 인터페이스는 Visual Basic을 사용하여 구현하였고, File, Edit, Generator, Build, Execution, Help의 6가지 메뉴로 구성되어 있다.

이와 같은 *VisCC*를 이용하여 ETL의 컴파일러 개발 과정을 살펴보면 다음과 같다. ETL의 입력명세를 입력받아 *VisCC*가 생성한 C++ 소스 코드는 여러 개의 프로그램으로 분리 생성된다. 그림 11은 각각 작성된 ETL 입력명세 화면(왼쪽)과 생성된 ETL 컴파일러 소스 코드를 보여준 화면(오른쪽)이다.

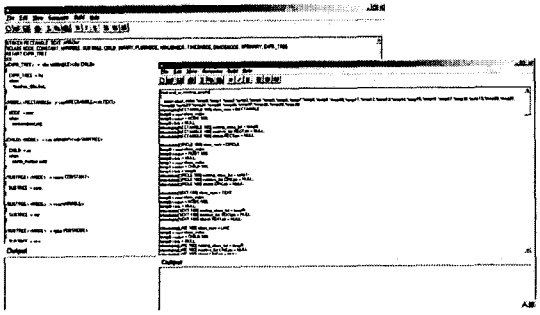
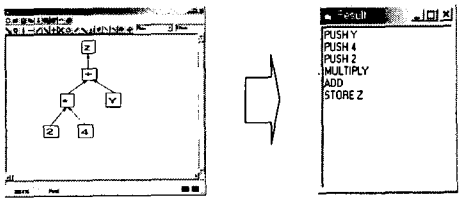


그림 11. *VisCC*에서 작성된 ETL 입력 명세 화면(왼쪽)과 생성된 ETL 컴파일러 소스 프로그램 화면(오른쪽)

다음은 하나의 ETL 프로그램을 *VisCC*를 사용하여 구현한 ETL 컴파일러에서 수행시킨 결과이다.



두 번째 적용 사례인 Prograph는 Labview와 더불어 상업적으로 사용하여 성공한 가장 대표적인 시각언어이다. 본 연구에서 Prograph의 구문 중 구문 분석기가 필요한 핵심적인 구문을 정의할 수 있는 문장들로 구성된 Prograph의 인터프리터를 *VisCC*를 이용하여 구현하였다. 그림 12는 *VisCC* 환경에서 작성한 Prograph의 입력명세 화면(왼쪽)과 이를 *VisCC*를 수행시켜 생성된 Prograph 인터프리터 소스 프로그램 화면(오른쪽)의 일부분이다.

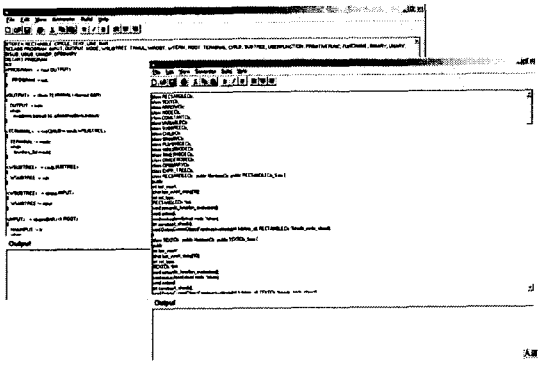
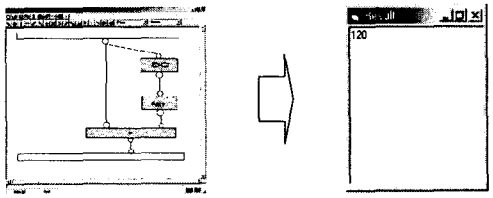


그림 12. *VisCC*에서 작성된 Prograph 입력 명세 화면(왼쪽)과 생성된 Prograph 인터프리터 소스 프로그램 화면(오른쪽)

다음은 factorial을 수행하는 Prograph 프로그램을 *VisCC*로 개발한 인터프리터로 실행시킨 결과이다.



5. 결 론

본 연구에서 제시하는 클래스를 기반으로 한 문법-지시적 변환 방법을 이용한 시각언어 컴파일러 자동 생성 방법은 PCG와 그 구문분석 방법이 가지는 클래스와 객체를 이용하여 시각언어 구문을 쉽고 자연스럽게 기술하는 구체적인 메카니즘을 제공, 그래픽 객체를 생성규칙으로 정의하는 명세방법 제공, 시각언어 구문 분석 모델링 방법 제공이라는 특성을 기반으로 하여, 의미 수행 코드를 이용한 컴파일러 후단부 자동 생성 방법을 제시함으로써 다음과 같은 특성 및 효과를 나타낸다.

- 클래스 기반의 의미 수행 코드를 이용한 문법-지시적 변환 방법을 제공한다. 시각언어의 구문에 따라 수행되는 의미 수행 코드 정의 방법, 생성 방법, 문법-지시적 수행 방법을 제공함으로써, 구문 분석기 다음 단계의 효과적인 구현 방법을 제공했다.
- 분리된 어휘 분석 단계를 제공한다. 구문 분석 과정에서 어휘 분석 단계를 분리시켜 취급함으로써 구문 명세의 복잡성을 감소시키고, 파서의 입력 객체의 수를 감소시키며, 파싱 이전의 단계에서 그래픽 객체 생성시의 오류를 분석할 수 있도록 했다.
- 수정 및 확장의 용이성을 제공한다. 시각언어 구문의 변경 및 확장시 문법의 수정으로 컴파일러를 재작성할 수 있으므로 직접적인 컴파일러 개발에 비해 수정 및 확장에 탄력적이다. 또한 의미 수행 코드의 분리 정의 방법을 통해 동일한 시각언어의 다양한 도구 구현에 효과적으로 사용될 수 있다.

참 고 문 헌

[1] 오세만, 컴파일러 입문(2nd ed.), 정익사, 1994.

[2] 이기호, 김경아, "객체지향 시각언어 문법의 정의 및 파싱방법", 한국정보과학회논문지(B), 제25권, 제2호, pp.715-721, 1998.
[3] 이기호, 김경아, "문법기반 객체지향 시각언어를 위한 컴파일러 생성기", 한국정보과학회논문지(B), 제26권, 제3호, pp.431-440, 1999.
[4] Alfred V.Aho, Ravi Sethi and Jeffrey D.Ullman, *Compilers principles, Techniques, and Tools*, Addison-Wesley, 1986.
[5] G. Costagliola, G. Tortora, S. Orefice and A. D. Lucia, "Automatic Generation of Visual Programming Environments," IEEE Multimedia Vol.3, No.3, pp.56-66, 1995.
[6] Etienne Gaonon, "SableCC: An Object-Oriented Compiler Framework," MS. thesis, Mcgill University, 1998.
[7] Eric J. Golin and Tom Magliery, "A Compiler Generator for Visual Languages," in Proceedings of 1993 IEEE Workshop on Visual Languages, pp.314-321, 1993.
[8] Kyung-Ah Kim and Kiho Lee, "Defining and Parsing Visual Languages with Object-Oriented Visual Language Grammar," in Proceeding of IASTED/SE'98, pp.119-124, 1998.
[9] Andy Schürr, "PROGRES, A Visual Language and Environment for PROgramming with Graph REwriting Systems," Technical Report AIB94-11, Archen University, 1994.
[10] D-Q. Zhang and K. Zhang, "VisPro: A Visaul Language Generation Toolset," in Proceedings of 1998 IEEE Symposium on Visual Languages, pp.208-125, 1998.

김 경 아

1990년 2월 이화여자대학교 전자계산학과 졸업(이학사)
1992년 2월 이화여자대학교 대학원 전자계산학과 졸업(이학 석사)
2001년 2월 이화여자대학교 대학원 컴퓨터학과 졸업(공학박사)
2001년 3월~2002년 2월 이화여자대학교 컴퓨터학과 대우전임강사
2002년 3월~현재 명지전문대학 컴퓨터정보과 조교수
관심분야 : 프로그래밍 언어, 시각언어, 형식언어, 컴파일러

교 신 저 자

김 경 아 120-776 서울특별시 서대문구 홍은3동 356-1 명지전문대학 컴퓨터정보과