韓國 컴퓨터情報學會 論文誌
第13卷 第2號, 2008. 3.

2008-13-2-1-4

C 코딩 스타일 검증기의 설계 및 구현

황 준 하*

Design and Implementation of a C Coding Style Checker

Junha Hwang*

요 약

지금까지 C 언어에 대한 다양한 코딩 스타일이 제시되어 왔으나 코딩 스타일에 대한 종합적인 검토가 부족하였다. 본 논문에서는 대표적인 C 코딩 스타일에 포함된 코딩 규칙들을 분석하고 그 외에 새로운 코딩 규칙들을 추가함으로써 새로운 C 코딩 스타일을 제안하고 있다. 아울러 CStyler라고 명명한 자동화된 C 코딩 스타일 검증기를 설계하였으며 Lex와 Yacc를 활용하여 이를 구현하였다. CStyler는 전처리가 수행된 후의 코드뿐만 아니라 전처리가 수행되기 전의 소스 코드에 대해서도 검증이 가능하도록 설계되었으며, 사용자가 새로운 코딩 규칙을 추가할 수 있도록 함으로써 유연성을 개선하였다. 본 논문에서 제시한 코딩 스타일과 코딩 스타일 검증기는 C 언어 교육과 향후 정적 분석 도구를 개발하고 확장하기 위한 연구에 활용될 수 있을 것으로 사료된다.

Abstract

Various coding styles for C language have been proposed so far but there has been a lack of synthetic review about them. In this paper, I propose a new C coding style by analyzing coding rules that are included in the representative C coding styles and by adding new coding rules besides them. In addition, I designed an automated C coding style checker named CStyler which was implemented using Lex and Yacc. It is designed to be able to verify unpreprocessed source code as well as preprocessed source code. And I improved its flexibility by being able to add a new coding rule by end user. I think that the new C coding style and coding style checker, CStyler, can be utilized for education and for future research to develop and extend a static analysis tool.

▸ Keyword : C 언어(C Language), 코딩 스타일(Coding Style), 렉스/야크(Lex/Yacc), 정적 분석(Static Analysis)

• 제1저자 : 황준하
• 접수일 : 2008. 1. 23, 심사일 : 2008. 2. 14, 심사완료일 : 2008. 2. 16.
* 금오공과대학교 컴퓨터공학부 조교수
※본 논문은 금오공과대학교 학술연구비에 의하여 연구된 논문임.

www.kci.go.kr

Ⅰ. 서 론

일반적으로 소프트웨어 개발 과정은 요구 분석, 설계, 구현, 테스트, 유지 보수의 단계로 진행된다[1]. 요구 분석은 주어진 문제를 분석하고 이해하는 단계이고 설계는 분석 결과를 어떻게 프로그램으로 구성할 것인가를 결정하는 단계이며 구현은 설계 결과를 기반으로 컴퓨터에서 수행 가능한 원시 코드를 작성하는 단계이다. 테스트는 구현 결과가 요구 분석 내용에 따라 적합하게 구현되었는지를 검사하는 단계로서 이 단계가 성공적으로 완료된 후 소프트웨어를 시장에 배포하게 된다. 그러나 실질적으로는 소프트웨어를 배포했다고 해서 소프트웨어의 개발이 완료되는 것은 아니며, 그 이후에도 프로그램 오류나 사용자의 요구에 따라 끊임없이 유지 보수 작업을 수행해야만 한다.

소프트웨어 개발 단계 중 구현 단계를 통해 원시 코드를 작성하는 작업을 흔히 '코딩'이라고 부른다. 코딩을 위해서는 우선 고객의 요구나 프로그래머의 지식 등을 고려하여 적절한 프로그래밍 언어를 선택해야만 한다. 프로그래밍 언어의 선택과 함께 소프트웨어 개발 시 고려해야 할 중요한 요소로는 프로그래밍 스타일이 있다. 프로그래밍 스타일이란 문장의 외형적 패턴이나 구성 요소들의 형태를 의미하는 것으로서 다양한 규칙들을 포함할 수 있다. 좋은 프로그래밍 스타일이 어떤 것인지 쉽게 정의할 수는 없다. 소프트웨어를 개발하는 프로그래머 혹은 그룹마다 필요로 하는 규칙들이 다를 수 있기 때문이다. 그러나 일반적으로는 읽기 쉽게 작성된 코드가 좋은 코드라고 할 수 있다. 즉, 프로그램은 이해하기 쉽게 작성되어야 한다. 이는 구현 단계뿐만 아니라 테스트 및 유지 보수의 비용을 줄이는 데 결정적인 역할을 담당한다[2]. 이해하기 어려운 코드는 오류를 발견하고 수정하기 힘들기 때문에 잠재적인 오류를 더 많이 포함할 우려가 있으며 이로 인해 유지 보수 비용이 증가하게 된다.

이와 같은 이유로 지금까지 많은 사람들이 프로그래밍 언어가 갖고 있는 기본적인 구문 규칙 외에 소위 "코딩 표준"이라고 하는 코딩 스타일을 제안해 왔으며, 많은 소프트웨어 개발 업체 및 그룹 역시 코드의 품질을 향상시키기 위해 나름대로의 코딩 표준을 제정하여 사용하고 있다. 코딩 표준의 제정은 이에 대한 준수 여부를 판단하기 위한 과정을 동반하게 되는데, 일반적으로는 동료 검토(Peer Review) 과정을 통해 수작업으로 수행하는 것이 보통이다. 물론 동료 검토 과정은 코딩 스타일에 대한 검증뿐만 아니라 기계적으로는 수행하기 힘든 의미를 파악하고 오류를 검출하기 위해 활용될 수 있다. 이 때 가능한 한 코딩 스타일에 대한 검증 과정을 자동화할 수 있다면 동료 검토 단계의 소요 시간을 단축하고 정확도를 높일 수 있을 것이다.

본 논문에서는 C 언어에 대한 코딩 스타일을 재검토하고 이에 대한 검증 작업을 자동화하기 위한 C 코딩 스타일 검증기의 설계 방안 및 구현 결과를 제시하고 있다. 현재 소프트웨어 개발을 위해 가장 많이 사용되고 있는 프로그래밍 언어로는 C, C++, Java를 꼽을 수 있다. 그 중에서도 C 언어는 1972년 Dennis Ritchie에 의해 만들어진 이후로 현재까지도 가장 많이 사용되고 있는 언어이다. 그만큼 C 코딩 표준에 대한 논의 역시 활발히 진행되어 왔다. 본 논문에서는 지금까지 발표된 C 코딩 표준들 중 Indian Hill C Style을 비롯한 총 3가지 코딩 표준을 분석하고, 그 외에 필요하다고 판단되는 규칙들을 추가함으로써 C 코딩 표준을 재정리하였다. 본 논문에서 제시하고 있는 새로운 코딩 표준은 C 언어 교육을 위해 활용될 수 있으며 아울러 조직에 적합한 코딩 스타일을 개발하기 위한 참고 자료로 활용될 수 있을 것이다.

그리고 본 논문에서는 C 코딩 표준에 포함되어 있는 규칙들에 대한 자동화된 검증을 위해 CStyler로 명명한 C 코딩 스타일 검증기를 설계하고 구현 결과를 제시하고 있다. 물론 지금까지 자동화된 C 코딩 스타일 검증기에 대한 개발 사례가 없었던 것은 아니다. C 코딩 표준의 제정과 함께 관련 검증 툴을 개발하여 배포하는 경우도 있으며 나아가 일부 소프트웨어 업체에서는 상용 검증 툴을 개발하여 고가에 판매하는 경우도 있다. 그러나 대부분의 경우에 있어서 구현 방안이 정확히 공개되어 있지 않은 실정이다. 본 논문에서는 Lex와 Yacc를 적절히 활용함으로써 자동화된 C 코딩 스타일 검증기를 효과적으로 구현할 수 있는 방안을 제시하고 있다. CStyler는 코딩 규칙의 특성에 따라 전처리를 수행한 후의 코드뿐만 아니라 전처리를 수행하기 전의 원시 코드에 대해서도 검증이 가능하도록 설계되어 있으며, 사용자가 검증기를 수행하는 도중에 새로운 코딩 규칙을 추가할 수 있도록 설계함으로써 시스템의 유연성을 개선하였다.

본 논문의 구성은 다음과 같다. Ⅱ장에서는 코딩 스타일과 관련된 기존 연구들에 대해 소개하며 Ⅲ장에서는 기존 코딩 스타일을 분석한 후 이들을 종합하고 추가로 필요하다고 판단되는 코딩 규칙들을 포함하여 새로운 C 코딩 스타일을 제안한다. Ⅳ장에서는 본 논문에서 제안하는 C 코딩 스타일 검증기의 설계 내용에 대해 기술하며 Ⅴ장에서는 구현 결과를 소개한다. 마지막으로 Ⅵ장에서는 결론 및 향후 과제를 설명한다.

www.kci.go.kr

II. 관련 연구

지금까지 프로그래밍 언어에 대한 다양한 코딩 표준이 발표되어 왔다. 이 중 C 언어와 관련된 대표적인 코딩 표준으로는 Indian Hill C Style[3], GNU Coding Standard[4], MISRA C Guideline[5, 6]이 있다. Indian Hill C Style은 1990년 AT&T사의 벨연구소에 의해 발표되었으며 1997년 개정판이 발표되었다. GNU Coding Standard는 GNU에서 발표하였으며 GNU 소프트웨어 개발 시 준수해야 할 사항들로서 코딩 스타일 외에 설치, 매뉴얼 작성 등 다양한 측면의 내용을 포함하고 있다. MISRA C Guideline은 자동차 산업 소프트웨어 신뢰성 협회(Motor Industry Software Reliability Association)에서 개발한 것으로서 당초에는 자동차 산업용 소프트웨어 제작 시 고려해야 할 코딩 표준을 포함하고 있었으나 현재는 자동차 산업 외의 모든 분야에 있어서 적용이 가능하도록 작성되어 있다.

C 언어 외에도 C++, Java를 비롯하여 다양한 프로그래밍 언어들에 대한 코딩 표준이 발표되어 왔으며, 각각 해당 언어의 특성에 따른 코딩 규칙들을 포함하고 있다[7, 8].

코딩 표준은 그 자체로서 의미 있는 산출물이 될 수도 있지만 또 한편으로는 코딩 스타일 검증기의 개발에 큰 영향을 미치게 된다. 예를 들어 UNIX 시스템의 코딩 스타일 검증 프로그램인 Lint는 Indian Hill C Style을 포함하고 있으며 코딩 표준이 변함에 따라 Lint 프로그램 역시 수정 보완되고 있다. MISRA C Guideline 역시 상용 툴의 개발에 큰 영향을 미치고 있다. Telelogic사의 RuleChecker라는 제품은 MISRA C Guideline에서 제시하고 있는 많은 코딩 규칙들을 포함하고 있으며[9], Gimpel Software사의 PC-lint와 FlexLint라는 제품 역시 MISRA C Guideline을 기본적으로 탑재하고 있다[10]. 그러나 상용 툴의 경우 세부적인 구현 방안이 공개되어 있지 않아 재현 및 확장이 용이하지 못하다는 단점이 있다. 따라서 본 논문에서는 코딩 스타일 검증기를 설계하고 구현하기 위한 방안을 소개함으로써 향후 이에 대한 활용이 가능하도록 하였다.

지금까지 소개한 코딩 스타일 검증기들은 사실상 코딩 스타일 검증 기능을 포함하는 정적 분석기(static analyzer)로 분류될 수 있다. 정적 분석은 구현이 완료된 코드에 대한 테스팅 방법 중 하나로서 프로그램을 수행하지 않고 코드를 분석하여 결함을 찾아내는 것을 의미하며, 코딩 스타일 검증 외에 버퍼 오버플로우 등 보다 복잡한 분석 기능들을 포함하고 있다. 경우에 따라서는 정적 분석을 위한 한 가지 주제가 단위 연구 과제가 될 정도로 대단히 난이도가 높은 문제가 될 수도 있다[11, 12]. 본 연구에서는 코딩 스타일 검증기에 대한 전체적인 구조와 관련된 설계 및 구현에 초점을 맞추고 있는 만큼 코딩 스타일과 직접적으로 관련된 코딩 규칙들을 우선적으로 고려하고 있다.

III. C 코딩 스타일

본 장에서는 3가지 코딩 스타일인 Indian Hill C Style, GNU Coding Standard, MISRA C Guideline을 분석, 종합하고 그 외에 필요한 코딩 규칙을 추가함으로써 새로운 C 코딩 스타일을 제안한다.

3.1 코딩 규칙의 분류 체계 및 분석

하나의 코딩 표준은 많은 코딩 규칙들로 구성되어 있다. 유사한 규칙들을 하나의 그룹으로 묶음으로써 규칙들에 대한 이해를 증진시키고 관리를 용이하게 할 수 있을 뿐만 아니라 교육적 목적을 위해서도 큰 도움이 될 수 있다[13].

기존 연구 [14]에서는 코딩 스타일의 분류에 대한 필요성을 언급하는 동시에 코딩 스타일의 범주를 Typographic Style, Control Structure Style, Data Structure Style로 분류하였다. Typographic Style은 외형적 모양과 관련된 규칙들을 의미하고 Control Structure Style은 함수 호출이나 제어문과 같은 제어 구조와 관련된 규칙들을 의미하며, Data Structure Style은 주로 변수의 사용과 관련된 규칙들을 의미한다. 즉, [14]에서는 코딩 규칙의 기능에 따라 범주를 분류하였다.

Indian Hill C Style은 C 언어의 문법적 구성 요소를 기준으로 총 11개의 범주로 분류하고 있는데, 파일, 주석, 함수 선언, 함수를 제외한 선언, 공백, 단순문, 복합문, 연산자, 명명, 상수, 매크로로 분류하고 있다. GNU Coding Standard은 코딩 스타일 전체가 하나의 장으로 구성되어 있어 명시적인 분류 체계가 나타나 있지 않다. 마지막으로 MISRA C Guideline은 주석, 식별자, 타입, 상수, 연산자, 제어문, 함수 등 총 17개의 범주로 분류하였는데 이는 C 언어의 구성 요소에 따라 분류한 Indian Hill C Style과 유사한 것이다.

코딩 규칙을 C 언어의 문법적 구성 요소에 따라 분류하는 것이 프로그래머의 입장에서 보다 자연스러운 접근 방법이 될 수 있다. 그러나 기능에 따른 분류 역시 해당 규칙의 특성을 이해하는 데 도움이 될 수 있다. 따라서 본 연구에서는 코딩 규칙의 분류 기준으로 문법적 구성 요소와 기능 두 가지 모두

www.kci.go.kr

를 고려하였다. 우선 1차적으로 문법적 구성 요소에 따라 총 10개의 범주로 분류하였는데, 파일, 주석, 변수와 상수, 타입, 문장, 연산자와 수식, 제어문, 함수, 구조체와 공용체, 전처리문으로 구성된다. 그리고 2차적으로 기능에 따라 레이아웃(Layout), 이름(Naming), 제어 흐름(Control Flow), 데이터 흐름(Data Flow)과 같이 4가지 범주로 분류하였다. 레이아웃은 외형적 모양과 관련된 것이고 이름은 변수, 함수 등의 이름 명명 규칙과 관련된 것이다. 제어 흐름은 코드의 제어 흐름과 관련된 규칙을 포함하며 데이터 흐름은 변수 값의 수정과 같이 데이터의 조작과 관련된 규칙을 의미한다. 문법적 구성 요소에 따른 분류와 기능에 따른 분류는 서로 포함 관계에 있지 않다. 예를 들어 레이아웃 규칙들 중에는 제어문과 관련된 것도 있고 주석과 관련된 것도 있다. 역으로 제어문 관련 규칙들 역시 어떤 것은 레이아웃과 관련될 수 있고 어떤 것은 제어 흐름과 관련될 수 있다. 따라서 본 연구에서는 코딩 규칙들을 문법적 구성 요소 분류 체계에 따라 정리하되 각 코딩 규칙이 기능에 따른 분류의 4가지 범주 중 어디에 포함되는지를 병기하였다.

Indian Hill C Style은 총 63개의 코딩 규칙을 포함하고 있으며 GNU Coding Standard는 30개의 코딩 규칙을 포함하고 있다. 사실 두 가지 코딩 스타일 모두 코딩 규칙을 명시적으로 나열하고 있는 것은 아니다. 따라서 서술식 문장으로부터 임의로 코딩 규칙을 추출하여 분석하였다. MISRA C Guideline은 코딩 규칙 번호가 명시적으로 표시되어 있으며 총 123개의 코딩 규칙을 포함하고 있다. 그러나 "표준 라이브러리의 사용 금지"와 같이 일반적인 C 프로그래밍에 있어서 무의미하다고 판단되는 코딩 규칙을 제외하면 총 93개의 코딩 규칙으로 구성된다.

〈표 1〉은 Indian Hill C Style과 GNU Coding Standard 그리고 MISRA C Guideline이 포함하고 있는 세부 코딩 규칙들을 분류 기준에 따른 각 범주 별로 규칙 개수를 순서대로 표기(Indian/GNU/MISRA)한 것이다. 구성에 따른 분류에 있어서는 3가지 코딩 스타일 모두 제어문과 함수 및 변수와 관련된 코딩 규칙이 가장 많이 차지하고 있다. 그러나 기능에 따른 분류에 있어서는 Indian Hill C Style과 GNU Coding Standard가 Layout 및 Name과 관련된 코딩 규칙이 대다수를 차지하고 있는 반면에 MISRA C Guideline은 Data가 가장 많으며 모든 범주에 있어서 비교적 고른 분포를 나타내고 있다.

〈표 1〉 Indian/GNU/MISRA 코딩 스타일 분석
〈Table 1〉 Analysis of Indian/GNU/MISRA Coding Styles

구성 \ 기능	Layout	Name	Control	Data	합계
파일	3/1/0	1/3/0	2/0/1	0/0/1	6/4/2
주석	3/0/1	0/0/0	0/0/0	0/0/0	3/0/1
변수	6/3/1	5/4/8	0/0/1	3/0/6	14/7/16
타입	0/0/0	2/1/1	0/0/0	0/0/9	2/1/10
문장	2/1/1	0/0/0	0/0/0	0/0/0	2/1/1
연산자	2/2/1	0/0/0	0/0/0	1/0/10	3/2/11
제어문	12/3/6	0/0/0	0/0/7	0/1/6	12/4/19
함수	6/5/4	3/2/8	0/0/3	0/0/2	9/7/17
구조체	3/2/1	2/0/1	0/0/0	0/0/0	5/2/2
전처리문	3/1/6	1/1/5	0/0/2	3/0/1	7/2/14
합계	40/18/21	14/11/23	2/0/14	7/1/35	63/30/93

3.2 새로운 C 코딩 스타일

본 연구에서는 새로운 C 코딩 스타일, 즉, 코딩 규칙들을 정의하기 위해 Indian Hill C Style의 63개 코딩 규칙과 GNU Coding Standard의 30개 코딩 규칙 그리고 MISRA C Guideline의 93개 코딩 규칙을 참고하였다. 그러나 새로운 C 코딩 스타일이 기존 3가지 코딩 스타일의 단순 합집합을 의미하는 것은 아니다. 3가지 코딩 스타일 사이에는 중복된 규칙이나 모순된 규칙이 존재할 뿐만 아니라 경우에 따라서는 삭제 또는 수정이 필요하다고 판단되는 규칙도 존재한다. 따라서 새로운 C 코딩 스타일은 이와 같은 모든 상황을 감안하여 작성되었으며 또한 3가지 코딩 스타일에 포함되어 있는 코딩 규칙 외에 필요하다고 판단되는 새로운 코딩 규칙을 포함하고 있다.

새로운 C 코딩 스타일은 총 137개의 코딩 규칙을 포함하고 있다. 〈표 2〉는 새로운 C 코딩 스타일에 포함된 코딩 규칙들을 분류 기준에 따라 정리한 것이다. 프로그램을 구성하는 가장 중요한 요소인 변수와 제어문 그리고 함수에 대한 코딩 규칙이 대다수를 차지하고 있으며 전처리문에 대한 규칙도 적지 않은 부분을 차지하고 있다. 그리고 예상한 바와 같이 프로그램의 외형을 결정하는 레이아웃과 식별자에 대한 명명 규칙이 많은 반면에 제어 흐름과 데이터 흐름에 대한 규칙은 상대적으로 낮은 비율을 차지하고 있다.

www.kci.go.kr

<표 2> 새로운 C 코딩 스타일 분석
<Table 2> Analysis of the New Coding Style

구성 \ 기능	Layout	Name	Control	Data	합계
파일	4	3	3	2	12
주석	5	0	0	0	5
변수와 상수	6	16	0	6	28
타입	0	3	0	0	3
문장	3	0	0	0	3
연산자와 수식	6	0	0	6	12
제어문	20	0	4	4	28
함수	9	14	2	1	26
구조체와 공용체	5	1	0	0	6
전처리문	6	5	1	2	14
합계	64	42	10	21	137

<표 3>은 기능에 따른 분류를 기준으로 각 범주에 포함된 코딩 규칙의 예를 보인 것이다. 코딩 규칙 8.17의 경우 수행 결과에 따라 어떤 함수가 특정 조건 하에서는 호출될 수도 있고 또 다른 조건 하에서는 호출되지 않을 수도 있다. 물론 해당 조건이 항상 true 또는 false를 반환하는 경우 정적 분석만으로도 호출 여부를 판단할 수 있을 것이다. 그러나 해당 조건이 항상 동일한 결과를 반환한다는 것을 자동으로 인식하는 것은 쉬운 일은 아니다. 따라서 규칙 8.17에 대한 준수 여부는 단순히 1회 이상 호출되는지를 기준으로 판단하게 된다. 그 외의 규칙들은 코드의 가독성을 높임으로써 잠재적 오류를 줄이기 위한 것들이다. <표 3>의 예에서 보인 코딩 규칙을 포함한 모든 코딩 규칙들에 대한 설명은 [15]에 기술되어 있다.

<표 3> 코딩 규칙의 예
<Table 3> Examples of Coding Rule

분류	번호	설명
Layout	[8.2]	함수 선언문이 다음 줄로 이어짐으로써 함수 매개변수가 2줄 이상에 걸쳐 나올 경우 다음 줄은 탭 1개만큼 들여 쓰거나, 첫 줄의 매개변수가 시작되는 위치와 같은 열에 위치시킨다.
Name	[3.17]	변수 이름들은 특수문자를 제외한 알파벳 문자만을 비교했을 때 대소문자를 무시하고 동일한 이름으로 명명해서는 안 된다. ▪int FirstCount, first_count; // X
Control	[8.17]	정의된 함수는 반드시 1회 이상 호출되어야 한다.

Data	[6.3]	마지막 대입 연산자를 제외하고 수식 내에서 대입문을 사용하지 않는다. ++와 -- 연산자는 그 자체로 대입문으로 간주된다.
		▪d = (a = b + c) + r; // X
		▪b = a++; // X

IV. CStyler의 설계

4.1 전체 시스템 구성

CStyler는 <그림 1>과 같이 Lex와 Yacc를 이용한 코드 분석기(Coding Rule Checker)를 중심으로 하여 입력 처리기(Input File Manager), 코딩 규칙 관리기(Rule Manager), 확장 전처리기(Extended Preprocessor) 및 사용자 인터페이스(User Interface)로 구성된다.

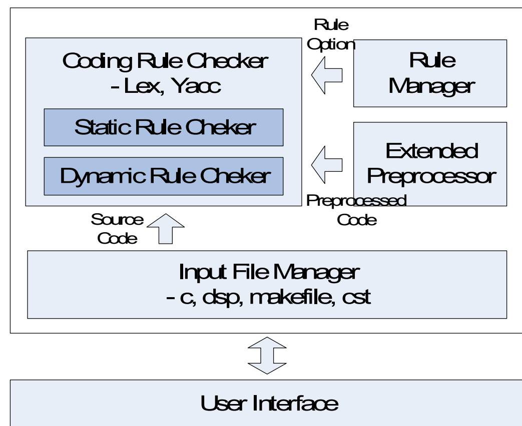

그림 1. CStyler의 전체 구성
Fig 1. Overall Structure of CStyler

코드 분석기는 본 시스템의 핵심 모듈로서 정적 코드 분석기와 동적 코드 분석기로 나누어진다. 정적 코드 분석기는 CStyler 개발 시 미리 정의된 코딩 규칙을 처리하는 모듈이며, 동적 코드 분석기는 사용자에 의해 추가된 새로운 규칙을 처리하는 모듈이다. 두 가지 모두 Lex와 Yacc를 활용하고 있으며 동작 방식 또한 유사하다. 동적 코드 분석기에 대해서는 4.3절에서 보다 상세히 설명한다.

코드 분석기는 코딩 규칙의 성격에 따라 전처리를 수행한 후의 코드를 대상으로 할 수도 있고 전처리를 수행하지 않은 원시 코드를 대상으로 할 수도 있는데 확장 전처리기가 이것을 가능하게 한다. 일반적인 전처리기를 사용할 경우 전처리 후에는 전처리와 관련된 모든 정보를 잃어버리게 됨에 따라 이와 관련된 코딩 규칙의 검증이 불가능해진다. 그러나 확장 전처리기는 전처리를 수행함과 동시에 원시 코드도 남겨둠으로써 전처리 후의 소스 코드뿐만 아니라 원시 코드에 대한 검증도 가

www.kci.go.kr

능하다. 이에 대해서는 4.2절에서 보다 상세히 설명한다.

입력 처리기는 단 하나의 소스 파일에 대한 검증뿐만 아니라 프로젝트 단위의 검증이 가능하도록 Makefile과 Visual C++ 프로젝트 파일(dsp) 그리고 자체 프로젝트 관리 파일(cst)을 처리함으로써 하나의 프로젝트와 관련된 모든 소스 파일들을 관리하는 기능을 담당한다.

규칙 관리기는 기 정의된 코딩 규칙 정보와 사용자에 의해 추가된 코딩 규칙을 관리하는 기능을 담당하고 있으며, 모든 규칙 정보는 ini 파일로 처리된다. 규칙 정보는 이름, 설명, 전처리 여부, 적용 범위, 값 등으로 구성된다. 전처리 여부는 전처리 후의 코드를 대상으로 할 것인지를 의미하는 것으로서 코딩 규칙 구현 시 결정되는 참고 정보이다. 적용 범위는 하나의 프로젝트를 대상으로 할 것인지 또는 하나의 파일을 대상으로 할 것인지를 의미하는 것으로서 사용자가 결정할 수 있도록 되어 있다. 값은 해당 규칙 검증에 필요한 옵션 값을 의미하는 것으로서 예를 들어 "제어문(조건문과 반복문)은 4-depth 이상으로 중첩되어 나타날 수 없다"라는 코딩 규칙의 경우 depth가 이에 해당한다.

4.2 Lex와 Yacc 그리고 전처리기의 활용

Lex와 Yacc는 소스 코드의 문법을 검증하기 위한 유용한 도구이다[16]. 본 연구에서도 Lex와 Yacc는 코딩 규칙을 검증하기 위한 핵심 도구로서 활용되고 있다. 그러나 C 언어의 경우 전처리 과정을 끝낸 후의 소스 코드에 대해서는 파서가 동작하는 데 전혀 문제가 발생하지 않지만 전처리 과정을 거치지 않은 원시 코드에 적용하고자 할 경우에는 문제가 발생할 수 있다. C 언어는 전처리문으로 인해 매우 자유로운 표현력을 구사할 수 있지만, 이로 인해 전처리문을 포함한 코드의 분석 자체를 매우 어렵게 만든다. 대부분의 코딩 규칙의 경우 전처리 후의 결과를 대상으로 하고 있기 때문에 문제가 되지 않지만 일부 전처리문과 관련된 규칙에 있어서는 단순히 전처리 후의 코드만을 대상으로 할 경우 해당 규칙의 검증 자체가 불가능하게 된다.

〈그림 2(a)〉와 같은 원시 코드가 있다고 가정하자. 먼저 main 함수 내의 코드에 대해 전처리 과정을 거치지 않고 분석을 시도한다면 UINT32의 경우 identifier로 인식되며 이로 인해 에러가 발생한다. SUM 매크로 함수 역시 함수와 같이 인식될 수도 있지만 이에 대한 프로토타입 선언 및 정의가 없으므로 불완전한 상태가 된다. 이번에는 전처리 과정을 거친 후의 코드인 〈그림 2(b)〉를 살펴보자. 모든 문장들은 C 언어 문법에 부합되므로 전혀 문제가 되지 않는다. 그러나 원시 코드의 내용을 모두 잃어버리게 되므로 원시 코드에 대한 분석이 불가능하게 된다. 예를 들어 코딩 규칙들 중 "매크로 함수 내에서는 전역 변수를 사용할 수 없다"라는 규칙이 있다. 그런데 〈그림 2(a)〉의 SUM(var1, var2)는 이를 위배하고 있음에도 불구하고 〈그림 2(b)〉에서는 매크로 함수의 존재조차 인식하지 못하므로 이에 대한 검증이 불가능하다.

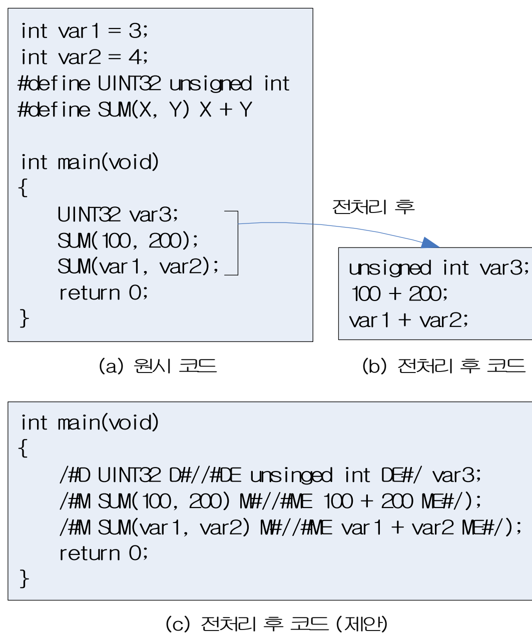

그림 2. 원시 코드와 전처리 후 코드의 예
Fig 2. An Example of Source Code and Preprocessed Code

CStyler의 확장 전처리기는 이와 같은 문제를 해결하기 위해 전처리 과정 수행 시 전처리 후의 코드뿐만 아니라 원시 코드의 전처리 관련 요소들을 그대로 포함하도록 하였다. 결국 Lex와 Yacc를 이용한 분석 단계에서는 필요에 따라 원시 코드 또는 전처리 후의 코드를 선택적으로 사용할 수 있게 된다. 〈그림 2(c)〉는 확장 전처리기를 통해 〈그림 2(a)〉의 main 함수 코드를 대상으로 전처리를 수행한 결과를 나타낸 것이다. '/#'와 '#/'는 각각 특정 영역의 시작과 끝을 나타내기 위한 특수 지시자이고 종류에 따라 D, DE, M, ME와 같은 지시자가 추가될 수 있는데, 각각 Define, Define Expand, Macro, Macro Expand의 약자로서 Expand는 전처리 후의 코드를 의미한다.

〈그림 2(c)〉의 경우 원시 코드와 전처리 후의 모든 코드가 포함되어 있으므로 필요에 따라서는 원시 코드의 분석이 가능하며, 또 한편으로는 전처리 후의 코드를 분석하거나 참조가

www.kci.go.kr

가능하다. 예를 들면 5째 줄의 경우 '/#M' 지시자를 통해 매크로 함수의 등장을 인지한 후 '/#ME' 블록의 분석 결과 전역변수가 사용되고 있음을 감지할 수 있다. 3째 줄의 경우 역시 원시 코드만을 대상으로 한다면 UINT32를 type으로 해석하지 못함에 따라 더 이상 분석이 어려운 상황이 발생할 수도 있지만, 원시 코드와 함께 전처리 후의 코드까지 참조가 가능하므로 전혀 문제가 되지 않는다.

물론 〈그림 2〉의 예만을 놓고 본다면 원시 코드만으로 이상의 문제를 해결할 수 있는 방법이 없는 것은 아니다. 만약 전처리기와 분석기를 완전히 결합하여 분석과 전처리를 한꺼번에 고려할 수 있다면 문제를 해결할 수도 있다. 그러나 이 경우 분석기 자체의 복잡도가 매우 높아져 구현이 어려워진다. 뿐만 아니라 매크로를 보다 복잡하게 정의한다면 전처리기와 분석기의 결합만으로도 분석이 불가능한 경우가 생길 수 있으리라 예상된다. 따라서 본 연구에서는 단순성과 확장성을 고려하여 전처리와 분석 단계를 분리하되 전처리를 통해 전처리 후의 코드뿐만 아니라 원시 코드를 함께 가지고 있는 것이 보다 바람직한 방법이라 판단하였다.

4.3 사용자 코딩 규칙의 추가

본 논문에서 제시하고 있는 코딩 스타일은 총 137개의 코딩 규칙을 포함하고 있다. 이는 기존의 주요 코딩 스타일을 종합 분석하고 새로운 규칙을 추가하여 만듦으로써 그 자체로서 방대한 규칙들을 포함하고 있다고 할 수 있다. 그러나 프로그램을 작성하는 산업체나 팀 또는 프로젝트에 따라서는 또 다른 코딩 규칙을 필요로 할 수도 있다. 예를 들면 함수명의 명명 규칙이 프로젝트명에 따라 좌우될 수 있다. 이와 같은 경우를 대비하여 사용자 측면에서 유연성 있는 시스템, 즉, 새로운 코딩 규칙을 쉽게 추가할 수 있도록 하는 것은 매우 중요한 일이다.

새로운 코딩 규칙의 추가에 대처하는 첫 번째 방법은 새로운 코딩 규칙을 고려하여 CStyler 자체를 수정하고 재컴파일하는 것이다. 그러나 사용자 측면에서는 CStyler 자체를 이해해야 된다는 부담이 있으며 CStyler 프로그램의 보안 유지에도 바람직하지 못하다. 두 번째 방법은 CStyler 자체를 재컴파일하지 않고 새로운 코딩 규칙을 추가하는 것이다. 사용자 측면에서는 가장 이상적인 방식이라 할 수 있지만 방법론에 있어서 이를 달성하기란 쉬운 일이 아니다. Telelogic사의 RuleChecker는 새로운 코딩 규칙을 추가하기 위한 방안으로 스크립트 언어인 TCL을 사용하고 있다[9]. 이는 C 프로그래머에게 새로운 언어에 대한 습득을 요구함으로써 부담을 가중시키게 된다. 본 연구에서는 CStyler의 코딩 분석을 위한 핵심 내용인 Lex와 Yacc의 구현 사항을 사용자에게 드러냄으로써 CStyler의 재컴파일 또는 전체적인 프로그램에 대한 이해 없이도 새로운 코딩 규칙을 추가할 수 있도록 하였다.

새로운 코딩 규칙의 추가 및 실행과 관련하여 제공되는 기능은 〈표 4〉와 같다. 여기서 추가와 컴파일이 핵심 기능이다. 추가를 수행하면 규칙 관리기는 새로운 코딩 규칙을 등록한 후 C 언어 분석을 위한 틀을 갖춘 Lex와 Yacc 파일을 제공하며 사용자는 새로운 코딩 규칙에 맞게 이를 수정할 수 있다. 컴파일을 수행하면 Lex, Yacc 파일로부터 파서를 생성한 후 이를 토대로 C 컴파일러를 통해 실행 모듈을 생성한다. 실행 모듈은 이후의 해당 코딩 규칙 검증 시 사용된다.

〈표 4〉 새로운 코딩 규칙을 위한 기능
〈Table 4〉 Functions for Adding New Coding Rules

기능	설명
추가	새로운 코딩 규칙을 등록하고 이에 대한 Lex, Yacc 템플릿을 제공한다. 사용자는 필요에 따라 Lex, Yacc 파일을 편집할 수 있다.
열기	새로 추가한 규칙을 편집하기 위해 불러온다.
삭제	새로 추가한 규칙을 삭제한다.
컴파일	해당 규칙의 Lex, Yacc 파일로부터 파서를 생성한 후 C 컴파일러를 통해 실행 모듈(exe)을 만든다.

본 논문에서 제안하는 방법 역시 C 초보자가 사용하기에는 어려움이 있으리라 예상된다. 그러나 컴퓨터공학 전공자의 경우 컴파일러를 비롯한 다양한 교과목을 통해 Lex와 Yacc에 대해 학습하고 있기 때문에 비교적 적은 노력으로 사용 방법을 익힐 수 있으리라 생각된다. 아마도 C 언어 분석을 위해 새로운 언어를 습득해야 하는 노력에 비하면 훨씬 적은 비용이 들 것으로 판단된다.

V. 구현 결과

CStyler는 Visual C++ 6.0을 사용하여 Windows 운영체제 상에서 구현되었다. 그리고 파서 구현을 위한 도구로는 Flex와 Bison을 활용하였는데, 앞에서는 설명의 편의를 위해 각각 Lex와 Yacc로 지칭하였다.

〈그림 3〉은 전체적인 실행 화면을 보여주고 있다. CStyler의 화면 구성은 기본적으로 현재 C 개발 도구 중 가장 많이 사용되고 있는 Visual C++와 유사하다. 메뉴와 도구모음을 제외하면 크게 프로젝트창과 작업창 그리고 결과창으로 나누어진다. 프로젝트창에는 현재 작업 중인 프로젝트에 포함된 소

www.kci.go.kr

스 코드와 헤더 파일 목록이 나열된다. CStyler가 처리할 수 있는 프로젝트로는 Visual C++ 프로젝트, Makefile, 자체 프로젝트가 있으며 폴더 단위의 처리도 가능하다. 이를 통해 개별 소스 코드 별 검증은 물론 프로젝트 또는 폴더 단위의 검증이 가능하다. 〈그림 3〉의 경우 Makefile을 통한 프로젝트 관리 예를 보인 것이다. 작업창에서는 특정 소스 코드와 헤더 파일 및 사용자 코딩 규칙에 대한 편집이 가능하다. 코딩 규칙에 대한 검증 결과를 비롯한 각종 정보들은 결과창을 통해 확인할 수 있다. 참고로 CStyler는 코딩 규칙에 대한 검증 기능 외에 파일 또는 프로젝트 단위의 라인 수, 크기 등의 통계 정보를 제공하며, 함수 별 정보와 함수 사이의 호출 관계를 분석하고 이에 대한 정보도 제공해 준다.

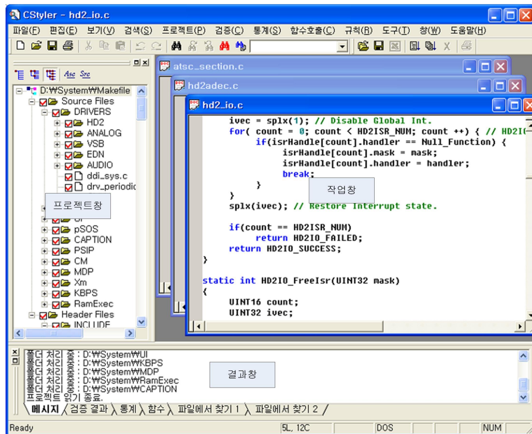

그림 3. 화면 구성
Fig 3. Screen Layout

〈그림 4〉는 하나의 샘플 코드에 대한 다음 5가지 코딩 규칙의 검증 결과를 보여주고 있다.

- 〔Rule 3.6〕 변수 이름의 첫 문자로 소문자 사용
- 〔Rule 7.17〕 조건문에서 대입연산자 사용 금지
- 〔Rule 8.17〕 호출되지 않는 함수 정의 금지
- 〔Rule 10.5〕 매크로 확장 결과 전역 변수 사용 금지
- 〔Rule 10.9〕 함수({ }) 내에서 #define 사용 금지

각 규칙에 대한 적용 여부는 옵션 설정을 통해 변경이 가능하다. 〈그림 4〉의 결과창을 보면 각 코딩 규칙 별로 의도적으로 추가한 오류에 대해 적절한 오류 메시지를 출력하고 있음을 알 수 있다. 첫 번째부터 세 번째까지의 코딩 규칙은 전처리 후의 코드를 대상으로 검증을 수행하며, 〔Rule 10.9〕의 경우에는 원시 코드를 대상으로 검증을 수행한다. 〔Rule 10.5〕의 경우에는 기본적으로 전처리 후의 코드를 대상으로 하되 필요하다면 원시 코드를 통해 매크로의 존재 및 내용을 확인할 수도 있다. 이와 같이 CStyler는 원시 코드와 전처리 후의 코드 모두를 대상으로 검증이 가능하도록 설계되어 있다.

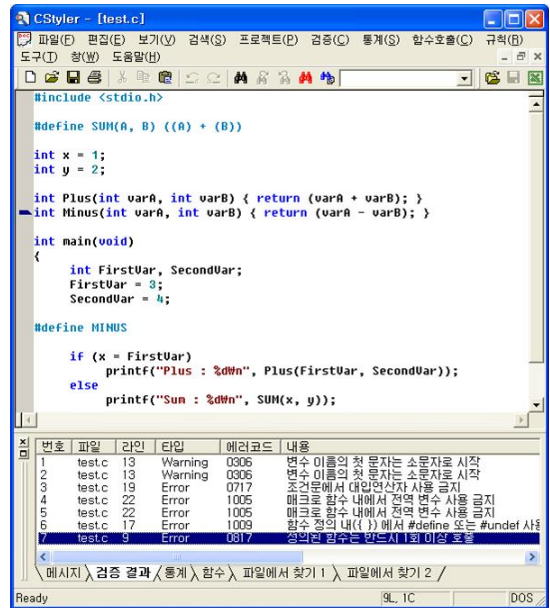

그림 4. 검증 결과의 예
Fig 4. An Example of Verification Results

〈그림 5〉는 새로운 코딩 규칙을 추가하는 예를 보여주고 있다. 예를 들어 "함수명은 'FUNC_'로 시작해야 한다"라는 코딩 규칙이 필요하다고 가정하자. 이에 대한 검증은 문법 규칙들의 집합인 Yacc 파일을 적절히 수정하면 된다. 먼저 CStyler를 통해 새로운 코딩 규칙을 추가하면 Lex와 Yacc의 템플릿이 제공된다. 그러고 나서 사용자는 함수 선언과 관련된 문법 규칙인 direct_declarator 내에서 함수명이 등장한 다음의 위치에 함수명을 검사하는 코드를 삽입하면 된다. 〈그림 5〉에서는 CheckFuncName이라는 함수를 만들고 이를 호출함으로써 해결하였다. 물론 사용자는 Lex와 Yacc에 대한 사용법을 숙지하고 있어야만 한다. 그러나 C 언어 문법과 관련된 대부분의 틀을 제공해 주므로 Lex와 Yacc에 대한 기본적인 사용법만 익히면 간단한 코딩 규칙을 추가하고 편집하는 데 큰 어려움이 없을 것으로 예상된다.

www.kci.go.kr

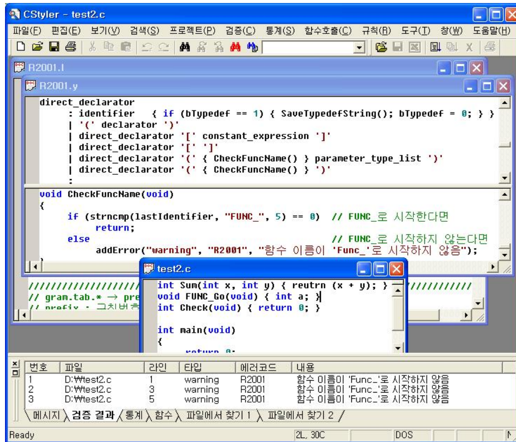

그림 5. 새로운 코딩 규칙의 추가 예
Fig 5. An Example of Adding a New Coding Rule

VI. 결론 및 향후 과제

본 논문에서는 C 프로그램 작성 시 준수해야 할 새로운 C 코딩 스타일을 제안하였으며, 아울러 코딩 규칙들에 대한 자동화된 검증 도구의 설계 및 구현 결과를 제시하였다. 새로운 C 코딩 스타일을 만들기 위해 기존의 Indian Hill C Style과 GNU Coding Standard 그리고 MISRA C Guideline을 참고하였으며 그 외에 필요하다고 판단되는 새로운 규칙들을 추가하였다. 그 결과 총 137개의 코딩 규칙을 포함하고 있는 코딩 스타일이 완성되었다. 본 연구를 통해 구현된 C 코딩 스타일 검증기 CStyler는 구문 분석 도구인 Lex와 Yacc를 활용하여 제작되었으며 향후 확장성을 고려하여 전처리 전, 후의 모든 코드에 대한 분석이 가능하도록 설계하였다. 또한 유연성을 개선하기 위해 실행 시간에 새로운 규칙을 추가할 수 있는 기능을 추가하였다. 본 논문의 연구 결과는 C 언어 교육을 위해 활용될 수 있을 뿐만 아니라 다양한 정적 분석 및 동적 분석 도구를 개발하는 데 활용될 수 있으리라 판단된다.

C 코딩 스타일의 정리 및 자동화된 검증 도구의 개발은 소프트웨어의 품질을 향상시키기 위한 가장 기본적이면서도 우선적으로 선행되어야 할 작업이다. 본 연구 역시 이와 관련된 다양한 연구를 수행하기 위한 출발점으로 인식되어질 수 있다. 그러나 C 언어의 자유도가 높은 만큼 본 연구와 관련하여 개선해야 할 과제도 많이 남아 있다. 우선 현재 CStyler가 가지고 있는 코딩 규칙 별 검증 기능에 대한 미흡한 부분을 개선함과 동시에 나머지 코딩 규칙에 대한 검증 기능 구현도 뒤따라야 한다. 그러나 개별 코딩 규칙의 복잡도에 따라 구현 소요 시간이 매우 오래 걸릴 수도 있고 그 자체로서 정적 분석을 위한 연구 주제가 될 수도 있으며 경우에 따라서는 검증 자체가 불가능할 수도 있으므로 이에 대한 규명이 선행되어야 할 것이다. 또 하나의 향후 과제로는 원시 코드와 전처리 후의 코드에 대한 검증을 위해 제시한 방법을 보완하는 것이다. 본 논문에서는 #define 상수와 매크로를 중심으로 이에 대한 해결 방안을 제시하고 있다. 그러나 #ifdef을 중심으로 한 조건 컴파일과 관련된 코딩 규칙의 개발 및 검증 기능이 미흡한 편이다. 향후로 조건 컴파일과 관련된 유용한 코딩 규칙의 개발이 필요하며, 이에 대한 검증 역시 #define에 대한 해결 방안과 유사한 방식으로 해결이 가능할 것으로 판단된다.

참고문헌

[1] I. Sommerville, Software Engineering 7th ed., Addison Wesley, 2004.
[2] X. Fang, "Using a Coding Standard to Improve Program Quality", Asia Pacific Conference on Quality Software, 2, 73-80, 2001.
[3] L.W. Cannon, et al. "Recommend C Style and Coding Standards", updated version of "Indian Hill C Style and Coding Standards", AT&T Bell Labs, 1997.
[4] R. Stallman, et al. "GNU Coding Standards", 2007.
[5] MISRA-C : Guidelines for the use of the C Language in critical systems, MISRA, 2004.
[6] L. Hatton, "Language Subsetting in an Industrial Context: A Comparison of MISRA C 1998 and MISRA C 2004", Information and Software Technology, 49(5), 475-482, 2007.
[7] H. Sutter, A. Alexandrescu, C++ Coding Standards: Rules and Guidelines for Writing Programs, Addison Wesley, 2004.
[8] M. Zaidman, "Teaching Defensive Programming in Java", Journal of Computing Science in Colleges, 19(3), 33-43, 2004.
[9] Telelogic Tau Logiscope 6.0 RuleChecker C - Reference Manual, Telelogic, 2003.
[10] Gimpel Software, PC-lint & FlexLint, http://www.gimpel.com, 2007.

www.kci.go.kr

〔11〕 D. Evans, D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis", IEEE Software, 19(1), 42-51, 2002.
〔12〕 K.S, Lhee, S.J. Chapin, "Buffer Overflow and Format String Overflow Vulnerabilities", Software-Practice & Experience, 33, 423-460, 2003.
〔13〕 L. Xiaosong, C. Prasad, "Effectively Teaching Coding Standards in Programming", Proceedings of the 6th Conference on Information Technology Education, 239-244, 2005.
〔14〕 P.W. Oman, C.R. Cook, "A Taxonomy for Programming Style", ACM Annual Computer Science Conference on Cooperation, 244-250, 1990.
〔15〕 URL: http://cespc1.kumoh.ac.kr/~jhhwang/cstyler.htm
〔16〕 J. R. Levin, T. Mason, D. Brown, Lex & Yacc, 2nd ed., O'Reilly & Associates, 1992.

저 자 소 개

황 준 하

1995 부산대학교 컴퓨터공학과 졸업 (공학사)
1997 부산대학교 컴퓨터공학과 졸업 (공학석사)
2002 부산대학교 컴퓨터공학과 졸업 (공학박사)
2002~현재 : 금오공과대학교 컴퓨터공학부 교수
관심분야 : 인공지능, 최적화, 기계학습, 프로그래밍언어

www.kci.go.kr