Journal of Korea Multimedia Society Vol. 10, No. 4, April 2007(pp. 537-547)

자바와 C 언어 결합을 위한 자바 전처리기

이창환[†], 오세만[††]

요 약

컴퓨터 기술 발전에 따라 컴퓨터는 복잡하고 다양한 작업을 실행하고 있다. 또한 프로그램 복잡도 증가와 사용 언어의 한계 때문에 둘 이상의 프로그래밍 언어를 사용하여 응용 프로그램을 구현되고 있다. 그러나 현재 여러 언어를 동시에 사용하여 프로그램을 작성하거나 이를 구현하는 일반적인 방법론은 없는 상태이다. 본 논문에서는 위와 같은 문제점을 해결하기 위하여 여러 언어의 혼합 사용 형태와 구현 방식을 이종 언어 결합도(Heterogeneous Language Integration Degree)를 통해 분류하였다. 또한 이종 언어 결합도를 통해, 현재 구현된 객체지향 언어인 자바와 절차형 언어인 C 언어를 동시에 사용하는 방법인 JNI(Java Native Interface)를 이종 언어 결합도 측면에서 개선한 자바 전처리기를 설계하고 구현하였다.

Java Preprocessor for Integration of Java and C

Changhwan Yi[†], Seman Oh[††]

ABSTRACT

According evolution of computer technology, computers execute complex and several tasks. Because of the complexity of program and restriction of programming language, applications are implemented using one more programming language. But it is no general methodology for using several languages and implementing it. This paper classified usages of programming language integration and methodology for implements programming languages integration through HLID(Heterogeneous Language Integration Degree) for solving above problem. And using HLID, it designed and implemented Java Preprocessor that improvement JNI – current implementation for integration between object-oriented language Java and procedural language C.

Key words: Java Preprocessor(자바 전처리기), Java(자바), JNI(JNI)

1. 서 론

컴퓨터 기술 발전에 따라 컴퓨터는 다양한 작업을 실행하고 있고 작업은 갈수록 복잡해지고 있다. 이런 작업은 하드웨어 의존적인 어셈블리 언어부터 작업 의존적인 스크립트 언어 등과 다양한 언어로 작성된다. 또한 프로그램 복잡도 증가와 사용 언어의 한계 때문에 응용 프로그램을 둘 이상의 프로그래밍 언어로 구현하는 경향이 늘고 있다. 최근 주요 언어로 대두된 자바도 플랫폼 의존적인 작업을 수행하기 위해서는 C 언어와 같이 사용해야 하는 경우가 생기고 있다. 자바와 C와 같이 여러 언어를 동시에 사용하여 프로그램을 작성하는 경우, 내부적으로 언어 간에 데이터와 흐름을 교환할 수 있는 언어 간의 정의된 절차가 필요하지만 일반적으로 정의된 방법은 없는 상태이다.

본 논문에서는 여러 언어의 혼합 사용 형태를 이종 언어 결합도(Heterogeneous Language Integra-

※ 교신저자(Corresponding Author) : 오세만, 주소 : 서울시 중구 필동 동국대학교(100-715), 전화 : 02)2260-3342, FAX : 02)2265-8742, E-mail : smoh@dongguk.edu

접수일 : 2007년 2월 2일, 완료일 : 2007년 2월 26일

[†] 정회원, 링크젠
(E-mail : yich@dongguk.edu)
[††] 정회원, 동국대학교 컴퓨터공학과

tion Degree)를 통해 분류할 것이다. 또한 현재 구현된 객체지향 언어인 자바와 절차형 언어인 C 언어의 이종 언어 결합 방식인 JNI의 이종 언어 결합도를 확인하고, 결합도를 증가시킨 자바 전처리기를 설계하고 구현할 것이다.

2. 이종 언어 환경의 결합

2.1 언어 환경

컴퓨터는 프로그램을 저장하고 실행할 수 있는 자료 구조와 알고리즘의 집합으로 정의할 수 있다. 그리고 프로그래밍 언어 구현은 언어로 작성된 프로그램을 실행할 때 사용되는 알고리즘과 자료 구조를 가지는 컴퓨터를 정의하는 것이다. 이와 같이 언어 구현을 통해 만들어진 컴퓨터를 가상 컴퓨터(Virtual Computer)라고 한다. 가상 컴퓨터의 기계어는 구현된 언어이며, 입력은 언어로 작성된 소스 프로그램이다[1]. 그림 1은 가상 컴퓨터의 작동 모습을 보여주는 것으로, 프로그램을 정의한 프로그래밍 언어 소스코드와 외부 정보를 입력으로 받아, 결과를 출력한다.

이런 가상 컴퓨터는 내부적으로 하드웨어와 소프트웨어의 여러 계층으로 구성된다. 임의의 프로그래밍 언어를 위한 가상 컴퓨터는 환경에서 제공하는 하드웨어와 소프트웨어의 요소를 파악하고 이와 같은 요소를 이용하여 구현한다. 자세히 살펴보면, 가상 컴퓨터를 실행할 하드웨어와 구현하는데 필요한 운영체제나 컴파일러, 어셈블러, 링커와 같은 여러 시스템 프로그램, 구현할 언어를 위한 컴파일러나 인터프리터, 디버거 같은 것으로 구성되어 있다. 또한 가상 컴퓨터의 입력으로 들어가는 응용 프로그램을 만드는데 사용되는 편집기나 통합 환경, 기타 여러 프로그램이 있을 수도 있다. 이와 같은 것으로 새로운 환경을 만들어 낼 수 있으면, 이를 구현한 언어를 위한 언어 환경(Language Environment)이라 할 수 있다. 이에 본 논문에서는 특정 언어로 기술된 소스 코드를 입력하고 실행하는데 필요한 모든 하드웨어와 소프트웨어로 만들어지는 환경을 언어 환경이라고 부를 것이다.

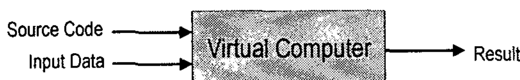

그림 1. 프로그래밍 언어에 대한 가상 컴퓨터

2.2 이종 언어 환경 결합

이종 언어 환경 결합(Heterogeneous Language Environment Integration)은 어떤 가상 컴퓨터의 결과를 얻기 위해 동시에 서로 다른 여러 개의 언어 환경을 사용하는 것을 말한다. 이종 언어 환경 결합은 새로운 언어 환경을 만들게 되고, 이를 가지는 가상 컴퓨터가 새로 만들어지게 된다. 운영체제를 구현하기 위해 C 언어와 어셈블리 언어를 동시에 사용하는 것이 있고, 자바에서 자바 실행 환경(Java Runtime Environment)이 제공하지 못하는 기능의 구현을 할 때 사용하는 JNI(Java Native Interface)가 대표적인 예이다.

이종 언어 환경 결합은 서로 다른 언어 환경을 동시에 사용하기 때문에, 자기 언어 환경에 있는 정보나 기능을 다른 언어 환경에서 접근할 수 있게 하는 방법이 있어야 한다. 이 방법을 일반적으로 언어 환경간의 인터페이스라고 하고, 언어 환경에 따라 단순한 자료 구조 형태부터 복잡한 함수나 클래스 메소드와 같은 다양한 형태로 정의된다.

여러 언어 환경이 동시에 사용이 되더라도 일반적으로 그림 2와 같이 하나의 주 언어 환경에서 여러 부 언어 환경에 있는 기능이나 정보를 사용하는 형태를 가진다.

그림 2와 같이 언어 환경 결합하기 위한 인터페이스를 사용하기 복잡한 경우, 결합에 필요한 정보 참조나 인터페이스를 생성해주는 생성기와 같은 결합을 쉽게 하기 위한 여러 도구가 제공되기도 한다.

2.3 이종 언어 환경 결합도

이종 언어 환경 결합도(*Heterogeneous Language* Environment Integration Degree)는 서로 다른 언어 환경을 결합하여 사용할 때 결합된 언어 환경의 사용자가 결합된 언어 환경이 새롭게 보이는 정도로 정의

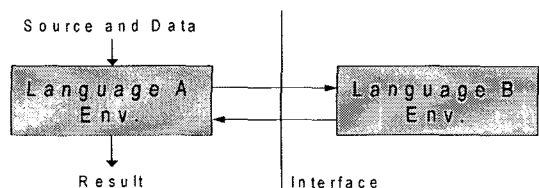

그림 2. A 언어 환경에서 B 언어 환경을 사용하는 과정.

할 수 있다. 결합도가 강할수록 결합된 언어 환경이 새로운 언어 환경처럼 보이게 되고, 약할수록 결합된 언어 환경을 이루는 각각의 언어 환경이 외부로 보이게 된다. 또한 언어 환경을 결합하기 위해 사용자가 다루는 단계 수가 작을수록 결합도가 강하고, 수가 많을수록 결합도가 약하다고 할 수 있다. 그러나 결합에 필요한 단계가 많아도 결합 과정을 사용자가 인식하지 못하면 강한 결합도라고 말하고, 반대로 결합 단계가 작더라도 결합 과정을 사용자가 인식하면 약한 결합도라고 말한다.

이종 언어 환경 결합도는 언어 처리 과정을 따라 파일 결합(File Integration), 문법 결합(Syntax Integration), 의미 결합(Semantic Integration)으로 나눌 수 있다. 파일 결합은 독립적으로 작성된 소스나 중간 형태의 파일을 이용하여 결합을 하는 것을 말한다. 문법 결합은 언어의 처리 과정 중 문법을 다루는 단계에서 언어를 결합하는 것을 말한다. 의미 결합은 의미 처리 과정에서 결합하는 것을 말한다. 파일 결합, 문법 결합, 의미 결합으로 갈수록 사용자는 새로운 언어 환경으로 인식을 하게 되고 강한 결합도를 가지게 된다.

■ 파일 결합

파일 결합은 각 언어 환경에서 중간 단계나 최종 단계에서 생성되는 파일을 사용하여 이종 언어 결합을 하는 것을 말한다. 언어 환경 결합에 필요한 인터페이스의 사용과 검사는 사용자에 의해서 이루어진다. 여기에 속하는 예는 C와 어셈블리의 결합, 자바와 C나 C++ 언어 결합이 있다.

파일 결합은 언어 환경을 구현하는 구현자 입장에서는 인터페이스만 정의하므로 구현자의 노력이 적게 들고, 언어 결합을 쉽게 할 수 있는 장점이 있다. 그러나 결합을 사용하는 사용자가 인터페이스를 깊게 이해해서 사용하는 단점을 가지고 있다. 또한 다른 결합에 비해 사용자가 결합 과정을 직접 다루기 때문에 사용 방법이 복잡하고, 사용자가 관련된 코드를 직접 작성을 하기 때문에, 오류가 발생할 가능성이 높은 문제점을 가지고 있다.

파일 결합에 필요한 파일을 생성하는 방법은 개발자가 직접 인터페이스를 익혀 인터페이스 규칙에 맞는 파일을 생성하는 방법과 생성기를 사용하여 소스 파일이나 중간 형태 파일에서 생성하는 방법, 독립적으로 기술된 인터페이스 정의 파일에서 생성하는 방법이 있다. C 언어와 어셈블리 언어의 결합은 사용자가 필요한 생성하는 방법을 사용하고, 자바와 C/C++ 언어 결합을 위해 사용되는 JNI는 소스 코드에서 필요한 파일을 생성하는 방법을 사용하고 있다. 원격 프로시저 호출을 구현하기 위해서 독립적으로 기술된 인터페이스 정의 파일에서 파일을 생성하는 방법을 사용하고 있다.

■ 문법 결합

문법 결합은 주 언어 환경의 소스 코드에 부 언어 환경의 소스 코드를 삽입하는 방식으로 결합하는 방법이다. 주 언어 환경에 삽입된 소스 코드는 처리기에 의해 주 언어 환경을 위한 소스 파일과 부 언어 환경을 위한 소스 파일로 분리된다. 또한 주 언어 환경과 부 언어 환경간의 결합을 위한 인터페이스 코드가 자동으로 삽입된다. 이런 문법 결합이 파일 결합과 다른 점은 부 언어 환경의 실행을 기술한 소스 코드를 삽입하는 것이다. 주 언어 환경에서는 삽입된 부 언어 환경의 소스 코드를 인식하기 위한 인식자가 주 언어 환경에 추가된다. 삽입되는 부 언어 환경의 소스 코드는 원 문법적 형태를 유지한다. 이런 문법 결합은 인라인 어셈블리 기능을 지원하는 일부 C 언어 컴파일러에서 볼 수 있다[2]. 또한 웹 서버에서 사용되는 서버측 스크립트 언어인 ASP와 PHP, JSP 도 두 언어 환경을 결합한 예이다[3-5].

문법 결합은 파일 결합에 비해 다음과 같은 장점을 가지고 있다. 첫째 이종 언어 결합에 따른 사용자의 거부감을 줄일 수 있다. 둘째 생성기에 의해 미리 정의된 인터페이스를 만족하는 코드를 자동으로 생성하기 때문에 오류 발생 가능성 줄일 수 있다. 셋째 다음에 설명할 의미 결합에 비해 문법 결합을 처리하는 도구의 구현이 쉬운 장점이 있다. 그러나 문법적으로 결합이 되었기 때문에, 이종 언어 결합에 따른 의미 오류를 처리하지 못하고, 결합 인터페이스의 사용 과정을 단순화하지 못하는 단점이 있다.

■ 의미 결합

의미 결합은 주 언어 환경에 의미를 확장한 부 언어 환경의 소스 코드가 삽입하여 이종 언어 결합을 하는 것이다. 부 언어 환경의 의미를 확장하기 위해서는 새로운 문장과 연산을 추가하거나 있는 문장과 연

산을 재정의한다. 또한 주 언어 환경에 존재하는 자료 구조를 부 언어 환경에서 직접 사용하는 경우도 의미 결합으로 볼 수 있다. 확장된 의미와 주 언어 환경을 참조를 가지는 삽입 코드는 처리기를 통해서 주 언어 환경과 부 언어 환경으로 분리되고, 결합 인터페이스를 사용한 소스 코드로 자동으로 변환된다. 의미 결합의 예로 C 언어에 SQL 문장을 삽입하여 사용하는 내장 SQL 기법, 네트워크 시뮬레이션을 위해 사용되는 NS-2에서 사용되는 C++와 OTCL의 결합, HTML 문서에 삽입된 자바스크립트를 들 수 있다[6-9].

의미 결합은 다른 환경에 있는 자료나 기능을 사용하는 필요한 복잡한 인터페이스 사용 과정이 간단한 연산이나 문장으로 변경되어, 다른 결합에 비해 오류 발생 가능성 감소시키는 장점을 가지고 있다. 그러나 의미 결합은 다른 결합에 비해 구현하기 힘든 단점을 가지고 있다.

3. 자바와 C 언어 환경 결합

3.1 객체 지향 언어 환경과 절차형 언어 환경의 결합

객체지향 언어 환경과 절차형 언어 환경은 매우 다른 특징을 가지고 있기 때문에 두 환경을 결합하기 위해서는 많은 문제를 해결해야 한다. 객체지향 언어 환경에서 절차형 언어 환경의 자원이나 기능을 사용하기 위해서는 먼저 이름 공간과 이름 매핑 문제를 해결해야 한다. 또한 다른 환경의 자원을 사용하기 때문에, 객체지향 언어 환경에서 지원하는 자원 관리 방법을 사용하지 못하는 문제가 있다. 절차형 언어 환경에서 객체지향 언어 환경을 이용하기 위해서는 객체지향 언어 환경에 있는 객체를 절차형 언어 환경에서 참조할 수 있는 방법 제공해야 한다. 이 때 객체지향 언어 환경의 객체 구조를 외부에 공개하던지, 내부 데이터에 접근할 수 있는 방법을 제공해야 한다. 그리고 객체지향 언어 환경에서 절차형 언어 환경의 자원을 사용할 때와 마찬가지로 자원 관리 문제를 해결해야 한다.

3.2 자바 언어와 C 언어의 결합

자바 언어 환경은 대표적인 객체지향 언어 환경이다. 자바는 스레드, 파일등과 같은 모든 자원은 객체로 관리한다. 객체 관리를 위해 가베지 컬렉션(Garbage Collection)이라는 자동화된 방법을 사용한다. 자동화된 관리를 위해 포인터를 지원하지 않으며, 객체의 내부 구조를 외부로 노출시키지 않고 있다.

C 언어 환경은 대표적인 절차형 언어 환경으로, 최근에는 거의 모든 하드웨어에서 어셈블리와 함께 꼭 구현되는 언어 환경이다. 이전의 어셈블리 역할을 수행하며 운영체제나 시스템 프로그램 구현에 많이 사용된다. 또한 언어 자체와 라이브러리를 분리하여, 다양한 하드웨어에 구현하기 쉬운 장점을 가지고 있다.

- JNI (Java Native Interface)

JNI(Java Native Interface)는 자바와 C/C++ 언어 환경을 결합하기 위해서 자바에서 제공하는 기능이다[10]. 자바에서 C/C++와의 연결 지점을 선언하기 위해서는 네이티브 메소드(native method)라 불리는 특별한 메소드를 사용한다. 이 메소드는 그림 3처럼 ***native*** 키워드를 수정자(Modifier)에 추가하여 선언한다. 자바에서 메소드를 네이티브 메소드로 선언을 하면, 자바 컴파일러는 클래스 파일의 메소드 정보에 메소드의 정의 없이 네이티브 메소드라는 정보만을 추가한다. 이 정보는 *javah*가 JNI를 위한 C/C++ 헤더 파일을 생성할 때와 java 실행 환경에서 네이티브 메소드를 호출할 때 사용된다. C/C++ 코드인 네이티브 코드는 *javah*가 생성한 원형을 가지는 함수에 기술하며 JNI에서 정의한 방법에 따른 자바 객체에 대한 연산은 가지고 있다. JNI의 결합도는 파일 결합이고, 클래스 파일에서 네이티브 메소드에 대한 정보를 *javah*가 가져와 C를 위한 스켈레톤를 생성한다.

```
<AClassWithNativeMethod.java>
public class AClassWithNativeMethod {
    public native void theNativeMethod();
    public void aJavaMethod() {
        theNativeMethod();
    }
}

<AClassWithNativeMethods.c>
#include <stdio.h>
#include "AClassWithNativeMethods.h"
JNIEXPORT void JNICALL
Java_AClassWithNativeMethods_theNativeMethod(
    JNIEnv* env, jobject thisObj) {
    printf("Hello JNI World\n");
}
```

그림 3. JNI 예제 코드

■ JNI 전처리기

JNI 전처리기(JNI PreProcessor)는 자바 소스코드의 네이티브 메소드 선언 문법을 확장하여, 그림 4처럼 네이티브 메소드가 C 코드를 가질 수 있는 기능을 가지도록 자바 소스코드를 처리하는 도구이다[11]. 이 도구를 통해 네이티브 메소드에 네이티브 코드 포함하는 형태로, 자바와 C 언어 환경은 문법 결합의 결합도를 가질 수 있었다.

4. 자바 전처리기

4.1 개요

자바 전처리기(Java Preprocessor)는 JNI의 사용과정을 단순화하고, JNI를 사용할 때 필요한 여러 파일을 하나의 파일로 통합하여 소스 파일 관리의 편의성을 증가시키고, 네이티브 메소드에서 점 연산자를 이용한 자바 객체 연산 방법의 제공을 위한 목적으로 설계되고 구현이 되었다. 이를 위해 자바 전처리기는 JNI 전처리기가 제공하는 네이티브 코드 삽입 기능과 더불어 네이티브 코드에서의 자바 객체에 대한 연산을 점 연산자(.)로 할 수 있도록 재정의하였다. 네이티브 코드 내에서의 자바 객체에 대한 점 연산은 자동으로 같은 의미의 JNI 함수 호출 패턴으로 변환된다[12]. 자바 전처리기는 이전 C의 점 연산자(.)의 의미를 재정의하였으면, 문법 결합도를 가진 JNI 보다 강한 결합도인 의미 결합도를 가지고 있다.

그림 5는 JNI의 사용 과정을 단순화한 자바 전처리기의 사용과정을 보여주고 있다. 이전 JNI의 경우에는 자바 클래스 파일을 생성한 후, JDK에 포함된 *javah* 프로그램을 실행하여 C/C++헤더 파일을 생성

```
<AClassWithNativeMethod.jpp>
public class AClassWithNativeMethod {
    native {
        #include <stdio.h>
    }

    public native void theNativeMethod() {
        printf("Hello JNI World\n");
    }

    public void aJavaMethod() {
        theNativeMethod();
    }
}
```

그림 4. JPP 입력 파일 예

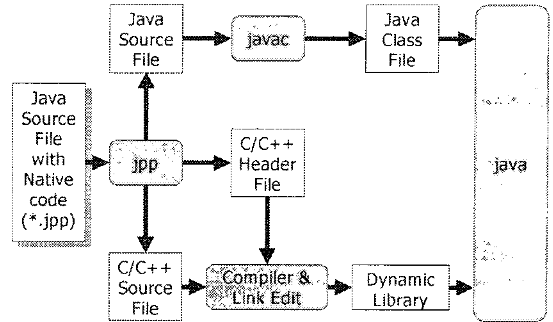

그림 5. 자바 전처리기 사용과정

한다. 새로 추가된 이 과정이 클래스 파일 생성과 클래스 파일 실행 사이에 이루어지기 때문에 사용자는 JNI 사용 과정을 복잡하게 느끼게 된다. 또한 네이티브 메소드의 실행 내용을 정의한 C/C++파일을 컴파일하기 위해서는 위의 과정을 통해 C 헤더 파일을 생성하는 작업이 필요하다.

그러나 자바 전처리기의 경우에는 한 번의 자바 전처리기 실행을 통해 JNI를 사용하는데 필요한 모든 파일을 생성하고, 네이티브 메소드내의 자바객체 연산을 같은 의미의 JNI 함수 호출로 변환하기 때문에 사용과정과 네이티브 메소드에서의 자바객체 연산이 단순화되었을 뿐 아니라 코드 관리를 편리하게 유지할 수 있게 된다.

소스 파일 관리의 편의성은 그림 6의 자바 전처리기 입력 파일의 예와 같이 자바 코드와 C/C++ 코드를 한 파일에 같이 기술을 하여 얻을 수 있다. 한 파일에 기술하기 때문에 여러 파일을 관리할 때 생기는 문제를 줄일 수 있다. 또한 네이티브 메소드의 시그네춰(Signature)나 복귀형(Return Type)이 변하는 경우에도 자바 전처리기의 실행만으로 변경된 내용을 갖는 자바 소스파일과 C/C++ 헤더파일, 소스 파일을 생성할 수 있다. 이전 방법으로는 변경된 내용이 자바 소스파일에만 있기 때문에, C/C++ 헤더 파일과 소스 파일에 변경된 내용이 적용되기 위해서는 자바 소스파일을 컴파일하고 C/C++ 헤더 파일을 *javah*를 통해서 생성하고, C 소스파일은 사용자가 직접 수정을 해야만 한다.

이런 기능을 가진 자바 전처리기를 사용하기 위해서는 그림 6과 같은 자바 전처리기 입력 파일을 사용자가 만들어야 한다. 이 파일은 *.jpp 확장자를 가지고 있고, C/C++ 코드를 포함하고 있는 자바 소스 파

```
<AClassWithNativeMethod.jpp>
public class AClassWithNativeMethod {
    native {
        #include <stdio.h>
    }

    public native void theNativeMethod() {
        printf("Hello JNI World\n");
    }

    public void aJavaMethod() {
        theNativeMethod();
    }
}
```

그림 6. 자바 전처리기 입력 파일 예

일이다. 자바 전처리기는 앞에서 말한 바와 같이 이 *.jpp 파일에서 자바 소스파일과 C/C++ 헤더 파일과 소스파일을 생성하고, 생성된 파일을 사용자가 직접 *javac*와 C/C++ 컴파일러로 컴파일하여 자바 환경에서 실행할 수 있는 파일을 생성한다.

자바 소스 코드에 네이티브 코드를 같이 기술하도록 하기 위해 자바 전처리기에서는 자바의 *native* 키워드의 의미를 확장하였다. 자바 언어 명세(Java language specification)에 있는 *native* 키워드의 의미는 자바 컴파일러에 메소드가 네이티브 메소드이라는 것을 알려주는 역할만 한다[13]. 그러나 JPP에서는 *native* 키워드가 C/C++ 코드 영역을 JPP에게 알려주는 역할을 하며, 네이티브 코드 문장 영역을 선언하고 네이티브 코드를 가지는 네이티브 메소드를 정의 할 수도 있도록 한다. 또한 입력 파일의 네이티브 코드에는 점 연산자 "."를 사용한 자바 객체 연산을 같이 기술 할 수 있다.

그림 7에서 전자는 네이티브 문장영역을 정의를 나타내고, 후자는 네이티브 메소드를 정의한 것이다. EBNF(Extended Backus-Naur Form)로 기술된 자바 문법에 네이티브 메소드 문장 영역과 네이티브 메소드 정의를 위한 생성 규칙을 추가하였다[14].

```
public class NativeKeywordExam {
  native {
    /* C/C++ 코드 기술 */
  }

  native NativeMethod() {
    /* C/C++ 코드 기술 */
  }
}
```

그림 7. 네이티브 문장과 네이티브 메소드 정의 방법

정리하면 JPP는 네이티브 코드를 가진 자바 소스 파일(*.jpp)에서 자바 소스 코드(*.java)와 네이티브 코드를 가진 C/C++ 소스 파일(*.c)을 생성하고, 네이티브 라이브러리를 만드는데 필요한 C 헤더 파일(*.h)을 생성하는 일을 한다.

그림 8은 자바 전처리기의 구조를 나타낸다. 입력으로 JPP 파일을 받아 JPPFileInfoStorage에 수집된 정보를 저장하고, 저장된 정보는 자바 소스 파일 생성기, C/C++ 헤더파일 생성기, C/C++ 소스파일 생성기에서는 자바 소스파일, C/C++ 헤더파일, C/C++ 소스파일을 생성하는데 사용한다. 이 중 C/C++ 소스파일 생성기에서는 입력파일인 *.jpp 파일에 대한 정보를 가지고 있는 JPPFileInfoStorage와 자바 클래스 라이브러리에 대한 정보를 가져올 수 있는 JavaClassesInfoStorage를 참조하여 점 연산자를 사용한 자바객체에 대한 연산을 JNI 함수 패턴으로 변환하여 *.c의 C 소스로 출력한다.

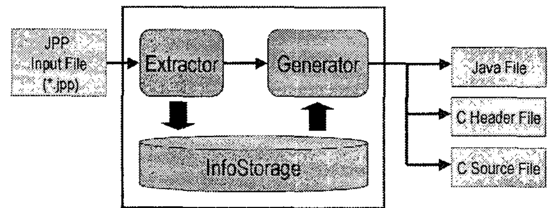

그림 8. 자바 전처리기 구조도

이 JPP는 다음과 같은 자바 클래스 구성되어 있다.

- Extractor class
- InfoStorage classes
  - JPPFileInfoStorage class
  - JavaClassesInfoStorage class
- Info classes
  - ClassInfo class
  - FieldInfo class
  - MethodInfo class
  - CodeInfo class
- Generator classes
  - JavaGenerator class
  - CHeaderGenerator class
  - CSourceGenerator class

4.2 정보 추출기

정보 추출기는 자바 전처리기 입력 파일(*.jpp)안의 정보를 읽어 정보 저장소에 정보를 저장하는 일을 한다. 정보 저장소에는 클래스에 대한 정보와 자바 코드와 C/C++ 코드에 대한 정보가 저장된다. 이 정보를 찾기 위해서 정보 추출기는 클래스 정보를 수집하고 자바 코드와 C/C++ 코드를 분리하는 파서를 사용한다. 또한 네이티브 코드에 있는 C 문장만을 파싱하여 AST(Abstract Syntax Tree)를 생성하기 위한 별도의 파서가 같이 사용된다.

클래스 정보 수집하는 파서를 살펴보면, 파서에 의해서 수집되는 클래스에 대한 정보는 클래스 자체의 정보, 클래스내의 필드와 메소드에 대한 정보로 나누어진다. 클래스에 대한 정보는 클래스의 이름, 수정자에 대한 정보이고, 필드에 대한 정보는 필드의 이름, 형, 수정자에 대한 정보이다. 메소드에 대한 정보는 메소드의 이름, 복귀형, 인자형과 인자 이름, 수정자에 대한 정보이다. 이 정보를 수집하기 위한 파서는 일반적인 자바 파서가 문장 블록까지 전부 파싱에 하는데 비해, 문장 블록을 무시하고 파싱을 한다. 그리고 파서에 입력으로 들어오는 문자는 자바 코드와 C/C++ 코드를 가지는 자바 소스 파일과 C/C++ 소스파일을 만들기 위해 자바 코드 영역과 C/C++ 코드 영역을 보관하기 위한 곳에 따로 저장을 한다. C/C++ 코드가 네이티브 메소드의 문장 영역에 있는 경우에는 클래스 정보 중에서 관련 메소드 정보를 찾을 수 있는 참조(reference)를 같이 저장한다. 파서에 들어오는 입력이 자바 코드인지 C/C++ 코드인지를 구분하는 일은 파서가 *native* 키워드를 인식한 이후의 처음으로 나타나는 문장 블록은 C/C++ 코드로 생각하고, C/C++ 코드를 AST로 변환하여 C/C++ 코드와 같이 저장을 한다.

4.3 정보 저장소

정보 저장소는 JPP 파일 정보 저장소와 자바 클래스 정보 저장소로 이루어져 있다. JPP 파일 정보 저장소는 InfoStorage 클래스와 이를 상속받은 JPP-FileInfoStorage 클래스로 구현되었다. JPPFileInfo-Storage 클래스는 앞에서 설명한 클래스와 코드에 대한 정보만이 아니라, 파일 생성에 필요한 자바 소스 파일 이름과 같은 기타 정보도 같이 가지고 있다. InfoStorage에 저장된 정보는 Info 계열 클래스를 통해서 생성기에 전달이 된다. 이 클래스에는 Class-Info 클래스와 FieldInfo 클래스, MethodInfo 클래스, CodeInfo 클래스가 있다.

이 중 MethodInfo 클래스에서는 그림 9과 같은 형식의 네이티브 코드에 대한 AST 정보, 심벌 정보, 결과 C 소스 생성을 위한 네이티브 소스 정보 등을 가지고 있다.

자바 클래스 정보 저장소는 이후에 설명할 C/C++ 소스 생성기에서 점 연산자를 사용한 자바객체 연산을 JNI 함수 호출로 변환하는데 필요한 자바 클래스 라이브러리에 대한 정보를 제공하는 역할을 한다. C/C++ 소스 생성기에서 필요한 정보를 JPP 파일 정보 저장소와 같은 형식으로 제공하여, 동일한 인터페이스로 자바 클래스에 대한 정보를 참조할 수 있도록 하는 것이다. 이를 위해서 내부적으로 자바 리플렉스(reflection) API를 사용하여, 자바 클래스 정보를 C/C++ 소스 생성기에 전달한다.

4.4 파일 생성기

파일 생성기는 자바 소스 파일 생성기와 C/C++ 헤더파일 생성기, C/C++ 소스파일 생성기로 구성되어 있다. 자바 소스파일 생성기는 JPP 파일 정보 저장소에 저장된 자바 코드 정보를 사용하여 파일을 생성한다. 이 때 생성하는 자바 소스 파일 이름은 자바 전처리기 입력 파일 이름과 같은 이름을 사용하고, 이 정보도 JPP 파일 정보 저장소에서 가져온다.

자바 소스파일 생성기와 다음에 설명할 C/C++ 헤더파일 생성기, C/C++ 소스파일 생성기는 Generator 클래스를 상속받아 구현이 되었다. 이 중 자바 소스 파일 생성기는 JavaGenerator 클래스로 구현되었다. JavaGenerator는 JPPFileInfoStorage에서 CodeInfo

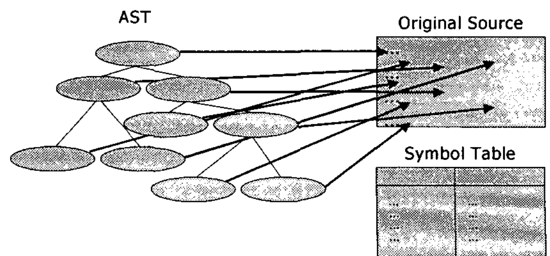

그림 9. MethodInfo의 네이티브 소스 정보 구조

클래스를 통해 자바 코드 정보를 얻어오고, 자바 코드인지를 나타내는 CodeInfo 클래스의 필드를 사용해서 자바 코드를 가지고 있는 자바 소스파일을 생성한다.

C/C++ 헤더파일 생성기는 JPP 파일 정보 저장소에 저장된 클래스와 메소드 정보를 사용하여 파일을 생성한다. 이 때 생성하는 C/C++ 헤더파일 이름은 클래스 이름과 같은 이름을 사용한다. 생성기는 클래스 이름을 사용해서 헤더 파일 머리 부분을 만든다. 그리고 클래스에 있는 네이티브 메소드 수만큼 네이티브 메소드의 C/C++ 함수 원형을 만들고, 헤더 파일의 꼬리 부분을 만든다.

이런 C/C++ 헤더파일 생성기는 CHeaderGenerator 클래스로 구현되고, JPPFileInfoStorage에서 ClassInfo 클래스와 MethodInfo 클래스를 통해서 네이티브 메소드에 대한 정보를 가져와 C/C++ 함수 원형을 만들고, 메소드 중 네이티브 메소드인지 구분은 MethodInfo 클래스의 필드를 통해서 이루어진다.

C/C++ 소스파일 생성기는 JPP 파일 정보 저장소에 저장된 클래스와 메소드 정보, C/C++ 코드 정보를 사용하여 파일을 생성한다. 이 때 생성하는 C/C++ 소스파일 이름은 클래스 이름과 같은 이름을 사용한다. 생성기는 클래스 내에 포함된 코드 정보를 사용해서 파일을 생성한다. 이때 코드 정보가 네이티브 메소드의 일부라는 것을 나타내면, 관련된 네이티브 메소드 정보를 사용해서 함수의 원형을 만들고 코드를 함수 안에 포함시킨다. 이때 네이티브 코드에 대한 AST를 참조하여 점 연산자를 사용한 자바객체 연산이 있는 경우, 이를 같은 의미의 JNI 함수 호출 패턴으로 변환하는 일도 같이 수행한다.

이런 C/C++ 소스파일 생성기는 CSourceGenerator 클래스로 구현되고, JPPFileInfoStorage에서 ClassInfo 클래스와 CodeInfo 클래스를 통해서 파일을 만든다. 이때 CodeInfo 필드를 사용해서 코드가 메소드에 대한 코드인지 검사하여, 메소드 코드라면 MethodInfo 클래스를 통해서 네이티브 메소드에 대한 C/C++ 함수를 만든다.

생성기는 C 소스를 생성하기 위해서 AST를 탐색하여 자바와 관련이 없는 코드는 C 소스에 그대로 출력한다. 자바와 관련된 코드는 네이티브 메소드인 경우에는 변환을 위해서 노드를 더 탐색하고, 생성기에 정보를 제공하기 위한 노드의 경우에는 정보를 얻고, 노드를 무시한다. 생성기가 네이티브 메소드를 검색하는 경우, 객체에 대한 연산을 찾기 위해서, 점 연산자가 사용된 수식을 찾는다. 다른 노드의 경우에는 그대로 독립 파일에 출력한다. 점 연산자가 사용된 수식을 찾으면, 연산자의 피연산자가 자바 객체인지 C 구조체인지를 판별한다. 만약 자바 객체라면, 점에 사용된 자바 객체 연산이 필드에 대한 연산인지, 메소드에 대한 연산인지를 구분해야 한다. 필드에 대한 연산이라면, 필드 값을 읽는 것인지 값을 변경하는 것인지를 구분해야 한다. 위와 같은 구분 과정을 거쳐, 사용된 자바 객체 연산의 종류에 대한 정보와 정보 테이블에 저장된 객체에 필요한 정보를 사용하여 코드 생성에 사용할 코드 형태를 찾아내어 독립 파일에 변환된 코드를 출력한다.

4.5 실행 결과

본 논문에서는 자바 전처리기의 사용 예로 JBlueZ를 사용하였다. JBlueZ는 리눅스에서 작동하는 C로 작성된 블루투스 스택인 BlueZ를 자바에서 사용할 수 있도록 한 자바 클래스이다. 실험으로 JNI 방식으로 자바와 C를 결합한 JBlueZ에 JNI 전처리기와 자바 전처리기를 적용해 보았다. JBlueZ는 단거리 무선 통신 기술인 블루투스의 네트워크 프로토콜 스택을 자바에서 사용하기 위해서 만들어졌다. JBlueZ는 프로토콜 스택 전체를 자바로 구현한 것이 아니라, 이미 C로 구현된 블루투스 네트워크 프로토콜 스택인 BlueZ를 자바에서 사용할 수 있도록 만든 것이다 [15-17].

JNI 결합 방식을 사용한 그림 10와 그림 11과 같은 JBlueZ 소스 코드의 일부에 JNI 전처리기와 자바 전처리기를 적용한 결과를 그림 12와 그림 13와 같다.

5. 결론 및 향후 연구

본 논문에서는 언어 혼합 사용을 연구하기 위해 언어 환경과 이종 언어 환경 결합, 이종 언어 환경 결합도를 정의하였다. 결합도를 파일 결합, 문법 결합, 의미 결합으로 분류 하였고, 결합도에 따라 현재 사용되고 있는 언어 결합을 결합도에 따라 분류를 하였다. 결합도에 따른 분류를 통해 이전의 언어 결합을 개선시키고, 새로운 언어 결합을 쉽게 고안할 수 있도록 하였다. 그리고 대표적인 객체지향 언어인

```
/* BlueZ.java */

/* HCI Inquiry */
/**
 * Perform an HCI inquiry to discover remote Bluetooth devices.
 *
 * See HCI_Inquiry in the Bluetooth Specification for further details of
 * the various arguments.
 *
 * @exception BlueZException If the inquiry failed.
 * @param hciDevID The local HCI device ID (see the hciconfig tool provided
 *     by BlueZ for further information).
 * @param len Maximum amount of time before inquiry is halted. Time = len
 *      x 1.28 secs.
 * @param max_num_rsp Maximum number of responses allowed before inquiry
 *     is halted. For an unlimited number, set to 0.
 * @param flags Additional flags. See BlueZ documentation and source code
 *      for details.
 * @return An InquiryInfo object containing the results of the inquiry.
 * @see #hciInquiry(int hciDevID)
 */
public native InquiryInfo hciInquiry(int hciDevID, int len, int max_num_rsp, long flags) throws
BlueZException;
```

그림 10. JBlueZ의 자바 코드

```
/* BlueZ.c */

JNIEXPORT jobject JNICALL Java_com_appliancestudio_jbluez_BlueZ_hciInquiry
  (JNIEnv *env, jobject obj, jint dd, jint length, jint max_num_rsp, jlong flags)
{
    /* Perform an HCI inquiry - result is returned as a Java InquiryInfo object. */

    /* Perform the HCI inquiry */
    /* Create a new instance of InquiryInfo */
    ii_cls = (*env)->FindClass(env, "com/appliancestudio/jbluez/InquiryInfo");
    ii_add_id = (*env)->GetMethodID(env, ii_cls, "addDevice",
"(Lcom/appliancestudio/jbluez/InquiryInfoDevice;)V");

    /* For each of the responses, create a new InquiryInfoDevice object and add it to the InquiryInfo
class.     */
    for (i=0; i<num_rsp; i++)
    {
        /* Create a new instance of BTAddress */
        /* bdaddr - BTAddress Object */
        /* Create a new instance of InquiryInfoDevice */
        /* Add the new InquiryInfoDevice object to InquiryInfo */
        (*env)->CallVoidMethod(env, info, ii_add_id, info_dev);
    }
    /* Release the memory used for inq_info */
    return info;
}
```

그림 11. JBlueZ의 C 코드

```
/* BlueZ.jpp */
public native InquiryInfo hciInquiry(int hciDevID, int len, int max_num_rsp, long flags) throws BlueZException {
 /* Perform an HCI inquiry - result is returned as a Java InquiryInfo object. */

    /* Perform the HCI inquiry */
    /* Create a new instance of InquiryInfo */
    ii_cls = (*env)->FindClass(env, "com/appliancestudio/jbluez/InquiryInfo");
    ii_add_id = (*env)->GetMethodID(env, ii_cls, "addDevice", "(Lcom/appliancestudio/jbluez/InquiryInfoDevice;)V");

    /* For each of the responses, create a new InquiryInfoDevice object and add it to the InquiryInfo class.    */
    for (i=0; i<num_rsp; i++)
    {
        /* Create a new instance of BTAddress */
        /* bdaddr - BTAddress Object */
        /* Create a new instance of InquiryInfoDevice */
        /* Add the new InquiryInfoDevice object to InquiryInfo */
        (*env)->CallVoidMethod(env, info, ii_add_id, info_dev);
    }
    /* Release the memory used for inq_info */
    return info;
}
```

그림 12. JBlueZ에 JNI 전처리기 방식을 적용한 예

```
/* BlueZ.jpp */
public native InquiryInfo hciInquiry(int hciDevID, int len, int max_num_rsp, long flags) throws BlueZException {
 /* Perform an HCI inquiry - result is returned as a Java InquiryInfo object. */

    /* Perform the HCI inquiry */
    /* Create a new instance of InquiryInfo */

    /* For each of the responses, create a new InquiryInfoDevice object and add it to the InquiryInfo class.    */
    for (i=0; i<num_rsp; i++)
    {
        /* Create a new instance of BTAddress */
        /* bdaddr - BTAddress Object */
        /* Create a new instance of InquiryInfoDevice */
        /* Add the new InquiryInfoDevice object to InquiryInfo */
        info.addDevice(ii_add_id, info_dev);
    }

    /* Release the memory used for inq_info */
    return info;
}
```

그림 13. JBlueZ에 자바 전처리기 방식을 적용한 예

자바와 절차형 언어인 C 언어의 결합에 결합도 개념을 사용하여 결합 방법을 개선된 자바 전처리기를 설계하고 구현하였다.

향후 연구 과제로는 자바와 C가 아닌 다른 언어간에 이종 언어 결합과 결합도 개념을 적용하여 새로운 이종 언어 결합을 구현하는 것이다. 또한 논문에서 구현한 자바 전처기에 문자열 연산을 위한 "+" 연산자와 배열 연산을 위한 "[]" 연산자의 지원을 추가하는 것이다. 그리고 자바 객체 생성/소멸 연산을 위해 C++의 *new*/*delete*와 같은 연산자를 추가할 것이다. 마지막으로 컴파일러의 최적화 기법을 사용하여 단순 변환에서 생기는 의미 없는 JNI 함수 호출을 줄여 실행 성능을 향상 시켜야 할 것이다.

참 고 문 헌

[1] Terrence W. Pratt and Marvin V. Zelkowitz, *Programming Languages: Design and Implementation, 4th edition*, Prentice Hall, 2001.

[2] Microsoft Visual C++ 6.0 Online Document

[3] 오세만, 이용규, 인터넷 프로그래밍 입문, 생능출판사, 2001.

[4] JavaServer Page Technology, http://java.sun.com/products/jsp, 2003.

[5] PHP: Hypertext Preprocessor, http://www.php.net, 2003.

[6] The Network Simulator - ns-2, http://www.isi.edu/nsnam/ns, 2003.

[7] HTML 4.01 Specification, http://www.w3.org/TR/html4/, W3C, 1999.

[8] Document Object Model (DOM) Level 2 HTML Specification, http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/, W3C, 2003.

[9] Introduction to Pro*C, http://www-db.stanford.edu/~ullman/fcdb/oracle/or-proc.html, 2003.

[10] *Java Native Interface*, Javasoft, 1997.

[11] 이창환, 오세만, "JPP: JNI 전처리기," 정보처리학회 논문지 9-A권 1호.

[12] 이창환, 오세만, "JNI 함수 호출을 통한 C에서의 자바 객체 사용," 정보과학회 2002년 봄 학술발표논문집(B) 29권 1호, pp. 340-342, 2002.

[13] *The Java Language: An Overview, White Paper*, Sun Microsystem, 1996.

[14] 오세만, 이양선, 김상훈, 고광만, 자바입문 개정판, 생능출판사, 2000.

[15] *Bluetooth Specification 1.1*, Bluetooth SIG, 2001.

[16] BlueZ - Official Linux Bluetooth protocol stack, http://bluez.sourceforge.net, 2003.

[17] JBlueZ - Java API for BlueZ, http://jbluez.sourceforge.net, 2003.

이 창 환

1998년 2월 동국대학교 컴퓨터공학과 졸업(학사)
2000년 2월 동국대학교 대학원 컴퓨터공학과(공학석사)
2003년 2월~동국대학교 대학원 컴퓨터공학과(공학박사)
2006년 9월~현재 (주) 링크젠 책임연구원
2007년 3월~현재 동국대학교 산업기술연구원 겸임교수
관심분야 : 프로그래밍 언어, 컴파일러, 임베디드 시스템

오 세 만

1985년 3월~현재 동국대학교 컴퓨터공학과 교수
1993년 3월~1999년 2월 동국대학교 컴퓨터 공학과 대학원 학과장
2001년 11월~2003년 11월 한국정보과학회 프로그래밍 언어연구회 위원장
2004년 6월~2005년 12월 한국정보처리학회 게임연구회 위원장
관심분야 : 프로그래밍 언어, 컴파일러, 모바일 컴퓨팅