# 14장 실행 가능한 프로그램 작성

## 14.2 중간 형태

다이애나(Diana, 에이다를 위한 기술적 중간 속성 표기법, Descriptive Intermediate Attributed Notation for Ada)는 에이다에 특화된 트리 기반의 IF로 독일의 칼스루에 대학교, 카네기 멜론 대학교, 인터메트릭스(Intermetrics), 소프테크(Softech), 타르탄 연구소(Tartan Laboratories)의 연구자들이 협력해서 개발했다. 다이애나는 TCOL과 AIDA라는 이전의 중간 형태의 기능을 통합한다. RTL(레지스터 전송 언어, Register Transfer Language)은 선형의 유사어셈블리 IF로 GNU gcc와 이를 계승한 도구들에서 사용할 목적으로 개발되었다.

### 【14.2.1】 다이애나

다이애나는 매우 복잡하지만(문서만 200쪽이다) 고도로 규칙적이라 여기서 적어도 훑어볼 수는 있다. 다이애나는 IDL[SS89]라는 이전 표기법을 사용해서 형식적으로 기술된다. IDL은 인터페이스 기술 언어(Interface Description Language)를 말한다.[1] IDL은 기계와 구현 독립적 방법으로 추상 자료형을 기술하는 데 널리 쓰인다. IDL-기반 도구를 사용하면 추상 자료형의 구체적인 인스턴스를 아스키로 나타낸 표준 선형 표현으로 변환하거나 그 반대를 수행하는 루틴을 자동으로 구성할 수 있다. IDL은 다이애나에 완벽히 적합하다. IDL의 다른 용도로는 다중데이터베이스 시스템, 분산 네트워크의 메시지 전달, 이종 병렬 기계를 위한 컴파일 등이 있다. 다이애나는 에이다 컴파일러의 전단과 후단 사이의 인터페이스를 제공하는 것 외에 좀 더 넓은 프로그램 개발 환경에서 에이다 코드 조각의 표준 표현으로도 빈번히 쓰인다.

---

주1. 불행히도 "IDL"이라는 용어는 인터페이스 기술 언어(다수가 존재)의 일반 범주와 다이애나에서 사용하는 특정 인터페이스 기술 언어 모두를 의미한다.

추상적으로 다이애나 구조는 트리로 정의되지만 반드시 트리로 나타낼 필요는 없다. 여러 플랫폼 간, 여러 업체가 만든 상품 간의 이식성을 보장하기 위해 다이애나를 사용하는 모든 프로그램은 선형 아스키 형식을 읽고 쓸 수 있어야 한다. 업체들은 트리 노드에 새로운 속성을 추가하는 방법으로 다이애나를 확장할 수 있지만(그리고 사실 그렇게 하도록 권장되지만) 표준을 따르는 다이애나를 생성하는 도구는 모든 표준 속성을 생성해야 하며 절대로 표준 속성을 표준과 다른 목적으로 사용하면 안 된다. 마찬가지로 표준을 따르는 다이애나를 이용하는 도구는 부가 속성이 제공되는 경우 이 속성의 정보를 이용할 수 있으나 표준 속성만이 주어졌을 때도 올바르게 기능할 수 있어야 한다.

에이다 컴파일러는 다이애나 트리의 노드를 별도의 패스에서 구성하고 장식한다. 다이애나 사용설명서에서는 속성 문법이 구성 패스를 구동하도록 권장한다. 이 패스는 트리 노드의 어휘 속성과 구문 속성을 수립한다. 어휘 속성은 식별자 이름의 철자와 구성소의 위치(파일명, 행 번호, 열 번호)를 포함한다. 구문 속성은 트리 자체의 부모–자식 연결이다.[2] 그 다음의 트리 탐색에서는 트리 노드의 의미 속성과 코드 기반 속성을 수립한다. 코드 기반 속성은 에이다 소스에 명시된 숫자 정밀도와 같은 저수준 속성을 나타낸다.

다이애나에서 심볼 테이블 정보는 별도의 구조가 아니라 선언의 의미 속성으로 표현된다. 원하는 경우 다애이나 구현은 트리 기반 추상 인터페이스를 유지하는 한 편의를 위해 별도의 구조로 심볼 테이블 정보를 꺼낼 수도 있다. 그 후 이름이 나오는 부분은 트리에서 "십자형 연결"을 통해 선언과 연결된다. 그러므로 속성이 완전히 주어진 다이애나 구조는 사실상 트리라기보다는 DAG다. 십자형 연결은 모두 의미 속성 사이에 존재하므로 (어휘 속성과 구문 속성으로 형성된) 초기 구조는 실제로 트리다.

IDL(그리고 다이애나 정의)는 4.6절과 유사한 트리 문법 표기법을 사용한다. BNF와 달리 이 표기법은 단순히 트리의 외형(즉, 산출물)이 아니라 완전한 구문 트리를 정의한다. 전형적인 구문 분석 트리에 있는 다수의 "쓸모없는" 노드를 피하기 위해 IDL은 클래스와 노드라는 두 종류의 기호를 구별한다. 노드는 "관심의 대상이 되는" 기호로 다이애나 트리에 존재하는 것들이다. 클래스는 "관심의 대상이 되지 않는" 기호로 문법 구성을 용이하게 하기 위해 존재한다. 사실상 클래스와 노드 간의 구별은 그림 14.5의 생산에 나온 `A : B` 표기법(4.6절에서 소개)과 동일한 목적을 가진다.

**예 14.17**

**다이애나의 ExpressionTree 추상화**

©(심화학습에 있는) 그림 14.9에는 다이애나 사용설명서[GWEB83, 26쪽]에서 가져온 IDL 예가 나와 있다. 여기에 정의된 `ExpressionTree` 추상화는 다이애나에서 실제로 해당하는 부분보다 훨씬 간단하지만 IDL 표기법을 설명하는 데는 충분하다. `(1 + 3) * 2`에 대한 `ExpressionTree`는 ©(심화학습에 있는) 그림 14.10에 나타냈다. 클래스

주2. 여기서 사용하는 용어는 잠재적으로 혼란을 줄 수 있다. 이제까지는 구문 분석 트리의 노드에 덧붙인 주석을 말하는 데 "속성"이라는 용어를 사용했다. 다이애나는 구문 트리의 노드에 저장된 모든 정보를 말하는 데 속성이라는 용어를 사용한다. 이 정보는 트리의 구조를 정의하는 다른 노드에 대한 참조를 포함한다.

(EXP와 OPERATOR)는 트리에 나오지 않는다는 점에 주의하자. 트리에는 노드(tree와 leaf)만 나온다.

```
Structure ExpressionTree Root EXP is
    -- ExpressionTree는 추상 자료형의 이름이다.
    -- EXP는 문법의 시작 기호(목표)기호다.

    Type Source_Position ;
       -- 이 유형은 전용(구현 독립적) 유형이다.

    EXP ::= leaf | tree ;
       -- EXP는 클래스다. 관례상 클래스명은 대문자로 나타낸다. 클래스는
       -- "::=" 생산으로 정의한다. 클래스의 우편에는 클래스나 노드인 단일 개체가
       -- 교대로 나와야 한다.

    tree => as_op: OPERATOR, as_left: EXP, as_right: EXP ;
    tree => lx_src: Source_Position ;
    leaf => lx_name: String ; lx_src: Source_Position ;
       -- tree와 leaf는 노드다. 이들은 실제로 ExpressionTree에 포함되는 기호다.
       -- 노드의 속성(하위 구조 포함)은 "=>" 생산으로 정의한다. 동일한 노드에 대한
       -- 다수의 생산은 서로 대체할 수 있는 대안이 아니며, 각기 부가적인 속성을
       -- 정의한다. 그러므로 모든 tree 노드는 as_op, as_left, as_right,
       -- lx_src와 같이 네 개의 속성을 가진다.
       -- 모든 leaf는 lx_name and lx_src라는 두 개의 속성을 가진다.
       -- 관례상 다이애나는 어휘 속성 앞에 'ls_' 추상 구문 속성 앞에 'as_',
       -- 의미 속성 앞에 'sm_', 코드 속성 앞에 'cd_'를 사용한다.

       -- 좀 더 실질적인 예에서 leaf는 자신의 선언 노드를 식별하는 sm_dec 속성을
       -- 가진다. 선언 노드에서는 부가적인 속성들이 유형, 유효 범위 등을 기술한다.

    OPERATOR ::= plus | minus | times | divide ;
    plus => ; minus => ; times => ; divide => ;
       -- OPERATOR는 네 개의 표준 이진 연산자로 구성된 클래스다. 빈 생산은
       -- 연산자의 이름을 통해 필요한 모든 것을 얻을 수 있다는 사실을 반영한다.
       -- OPERATOR는 네 개의 표준 이진 연산자로 구성된 클래스다. 빈(null) 생산은
       -- 연산자의 이름을 통해 필요한 모든 것을 얻을 수 있다는 사실을 반영한다.
       -- 빈 생산과 빈 하위 트리가 필요 없도록 트리 노드의 연산자를 전용 유형으로
       -- 만들 수도 있지만 이렇게 하면 연산자는 더 이상 표기법의 기계 독립적인 부분이
       -- 아니게 되며 이는 수용 불가능하다.
End
```

**그림 14.9 | 다이애나를 정의하는 데 쓰이는 IDL 표기법의 예**

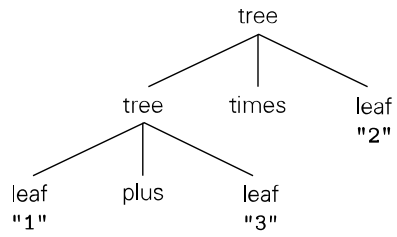


**그림 14.10** | ©(심화학습에 있는) 그림 14.9의 IDL 정의를 이용한 (1 + 3) * 2의 추상 구문 트리. 모든 노드는 Source_Position 유형의 src 속성을 가지지만 여기서는 보이지 않았다.

## 【14.2.2】 GNU RTL

많은 독자가 무료 소프트웨어 재단에서 배포하는 GNU 계열의 컴파일러에 익숙할 것이다. 이러한 컴파일러는 학계에서 매우 널리 사용되며 현업에서도 점차 더 널리 사용되고 있다. C, C++, 오브젝티브 C, 에이다 95, 포트란, 파스칼에 대한 GNU 컴파일러를 이용할 수 있다. C 컴파일러인 gcc는 최초의 GNU 컴파일러로 가장 널리 쓰인다. 상용으로 중요한 모든 것을 포함한 십여 개의 프로세서 구조를 위한 후단이 존재한다. 약 25개 가량의 다른 언어로 된 GNU 구현도 있지만 이들은 gcc에 기반하지 않는다.

어떤 면에서 gcc는 두 개의 IF를 가진다. 첫 번째 IF는 구문 트리고 두 번째 IF는 레지스터 전송 언어(RTL)이라는 명령어와 유사한 수식의 목록이다. 문법의 동작 루틴은 구문 트리를 구성한다. 소스 프로그램의 각 문장의 끝에서 동작 루틴은 해당 문장에 대한 구문 트리 일부를 RTL로 변환한다. 각 서브루틴의 끝에서 동작 루틴은 코드 개선과 목표 코드 생성을 수행하기 위해 컴파일러의 나머지 단계를 호출한다(이러한 단계는 수행될 코드 개선 집합에 따라 20개까지 있을 수 있다). 일단 주어진 서브루틴이 컴파일되면 이에 대한 구문 트리와 RTL은 회수할 수 있다(디버거가 사용하기 위해 저장하는 정보는 제외). 구문 트리와 RTL은 모두 파일에 작성되기보다는 여러 컴파일러 단계에 걸쳐 메모리에 유지된다. RTL은 사람이 읽을 수 있는 외부 형식을 가지며, 이 외부 형식은 컴파일러가 쓰고 (부분적으로) 읽을 수 있지만 자동 처리에는 적합하지 않다.

구문 트리는 컴파일러 전단과 후단 사이의 인터페이스를 구성한다. 구문 트리는 다소 언어 의존적이고 구문 트리에서 RTL로의 변환은 구문 분석과 인터리빙될 수 있기 때문에 전단과 후단의 구분이 완전하지는 않다. 새로운 언어에 대한 전단을 구성하려면 기존의 형식으로는 쉽게 표현할 수 없는 언어 기능을 지원하기 위해 구문 트리의 정의를 확장하고, 이러한 확장을 지원하기 위해 구문 트리에서 RTL로 변환하는 루틴을 수정해야 한다. 실제로는 후단에도 약간의 수정을 해야 하는 경우가 많다. 이러한 수정은 수정된 후단이 이전에 컴파일 가능했던 모든 언어에 대해 계속 동작하도록 이뤄져야 한다.

RLT은 리스프의 S 수식에 약간 기반한다. RTL 수식은 연산자나 수식 유형과 피연산자의 나열로 구성되며 외형상 왼쪽 괄호 바로 안쪽의 원소가 연산자인 괄호 목록으로 표현된다. 내부적으로 RTL 수식은 C 구조체와 포인터로 표현된다. 이렇게 포인터가 풍부한 구조체는 다수의 컴파일러 후단 단계들 사이의 인터페이스를 구성한다. 상수, 메모리나 레지스터에 있는 값에 대한 참조, 산술 및 논리 연산, 비교, 비트-항목 처리, 형 변환, 메모리나 레지스터로의 저장 등과 같이 다양한 수식 유형이 존재한다.

서브루틴의 몸체는 RTL 수식의 나열로 구성된다. 나열에 있는 수식을 `insn`이라고 한다. 각 `insn`은 다음과 같은 6개의 특수 코드 중 하나로 시작된다.

- insn: "평범한" RTL 수식
- jump_insn: 제어를 레이블로 이동시킬 수 있는 수식
- call_insn: 서브루틴을 호출할 수 있는 수식
- code_label: 건너뛰기의 목표 지점으로 가능한 곳
- barrier: 이전의 `insn`이 항상 건너뜀을 나타내는 지시자로 제어는 절대로 여기에 "도달하지" 않는다.
- note: 주석. 루프, 유효 범위, 서브루틴 등의 처음과 끝을 식별하기 위한 9 종류의 주석이 있다.

수식의 나열이 항상 완전한 선형은 아니다. 때때로 지연 공간이 있는 타겟 머신 명령어에 대응되는 2개나 3개의 쌍으로 여러 `insn`가 묶이기도 한다. 십여 종의 (`note`가 아닌) 주석을 각 `insn`에 달 수 있다. 이러한 주석은 부가 작용을 식별하거나, 타겟 머신 명령어나 레지스터를 명시하거나, 값이 정의되고 사용되는 지점을 추적하거나, 배열상에서 반복하는 데 사용되는 레지스터를 자동으로 증가시키거나 감소시키는 등의 목적을 가진다. `insn`은 심볼 테이블과 같이 동적으로 할당된 다양한 구조를 참조할 수도 있다.

**예 14.18**
RTL insn 나열

코드 조각 `d := (a + b) * c`를 단순화한 insn 나열은 ©(심화학습에 있는) 그림 14.11과 같다. 각 `insn`에서 앞부분의 숫자 세 개는 각기 해당 `insn`의 유일한 식별자, 이전 `insn`의 식별자, 다음 `insn`의 식별자를 나타낸다. 메모리나 레지스터 참조에 대한 `:SI` 모드 식별자는 단일(4바이트) 정수에 대한 접근을 의미한다. 이 그림에서 다양한 `insn` 주석에 대한 필드는 보이지 않았다.

후단은 목표 코드를 생성하기 위해 `insn`을 타겟 머신의 준형식적 기술에 저장된 패턴과 일치시킨다. 이 준형식적 기술과 `insn`의 기계 의존적 부분을 처리하는 루틴은 비교적 적은 수의 별도로 컴파일된 파일에 따로 떼어둔다. 결과적으로 컴파일러 후단의 대부분은 기계 독립적이며, 실제로 새로운 기계로 이식할 때 수정할 필요가 없다.

```
(insn 8 6 10 (set (reg:SI 2)
        (mem:SI (symbol_ref:SI ("a")))))

(insn 10 8 12 (set (reg:SI 3)
        (mem:SI (symbol_ref:SI ("b")))))

(insn 12 10 14 (set (reg:SI 2)
        (plus:SI (reg:SI 2)
            (reg:SI 3))))

(insn 14 12 15 (set (reg:SI 3)
        (mem:SI (symbol_ref:SI ("c")))))

(insn 15 14 17 (set (reg:SI 2)
        (mult:SI (reg:SI 2)
            (reg:SI 3))))

(insn 17 15 19 (set (mem:SI (symbol_ref:SI ("d")))
        (reg:SI 2)))
```

**그림 14.11 | d := (a+b) * c에 대한 단순화한 아스키 버전의 RTL**

## ✓ 확인문제

23. 인터페이스 기술 언어란 무엇인가?

24. 다이애나를 간단히 설명하라.

25. 다이애나에서 속성과 노드 간의 차이를 설명하라.

26. (C 외에) gcc 전단이 존재하는 언어를 세 가지만 말하라.

27. RTL을 간단히 설명하라. gcc의 다른 IF는 무엇인가?