論文 97-22-12-23

VHDL-to-C 사상을 위한 VHDL 컴파일러 전반부의 설계

正會員 공 진 홍*, 고 형 일*

A Design of VHDL compiler Front-end for the VHDL-to-C mapping

Jin-Hyeung Kong*, Hyung-Il Goh* *Regular Members*

※본 연구는 한국과학재단의 '96 종료 전문 연구(KOSEF 941-0900-050-1) 및 광운대학교 '97 학술연구비로 지원되었음

요 약

본 논문에서는 VHDL '87 및 '93 LRM의 전체 사양을 지원하며 VHDL-to-C 사상의 전처리 과정을 수행하는 VHDL 컴파일러 전반부의 설계 및 구현에 대해서 논한다. VHDL 컴파일러 전반부는 1) VHDL의 계층적 구조체, 선언 영역 및 가시성, 다중 정의 및 동형 이의어, 병행적 다중 스택 구조를 표현하기 위해서 분석 터미널 데이터에 심볼 트리를 구성하였으며, 2) VHDL 고유의 객체, 타입 및 서브타입, 속성과 연산자 등을 나타내기 위한 구조체 및 지원 함수를 설계하였고, 3) VHDL의 병행문/순차문, 행위/구조 기술, 동기 메커니즘 등을 분석하여 VHDL-to-C 사상에 필요한 어의 정보를 구축하고, 4) VHDL 분석 과정에서 어의 데이터의 저장 및 검색이 효과적으로 이루어지도록 어의 토큰 정의 및 어의 전파 기능 등을 설계하였다. Validation suite를 이용한 실험에서 VHDL 컴파일러 전반부는 LRM 전체 사양을 분석할 수 있음을 확인하였고, VHDL의 계층성/가시성/병행성/어의 검사 등을 효과적으로 처리하기 위해 설계 및 구현된 심볼 트리와 어의 토큰 등의 분석 데이터 모델에 대한 성능 분석 실험에서 VHDL 컴파일러 전반부는 20~30%의 개선 효과를 보였다.

ABSTRACT

In this paper, a design and implementation of VHDL compiler front-end, aims at supporting the full-set of VHDL '87 & '93 LRM and carrying out the preprocessing of VHDL-to-C, is described. The VHDL compiler front-end includes 1) the symbol tree of analyzed data to represent the hierarchy, the scope and visibility, the

*論文番號:97321-0910
接受日字:1997年 9月 10日

overloading and homograph, the concurrent multiple stacks in VHDL, 2) the data structure and supporting routines to deal with the objects, the type and subtype, the attribute and operation in VHDL, 3) the analysis of the concurrent/sequential statements, the behavior/structural descriptions, the synchronizing mechanism to build-up semantic data for the VHDL-to-C of the compiler back-end, 4) the definition of semantic token and the propagation of symbol & type to improve the registration and retrieval procedure of analyzed data. In the experiments with Validation Suite, the VHDL compiler front-end could support the full-set specification of VHDL LRM '87 & '93; and in the experiments to assess the performance of symbol tree and semantic token for the VHDL hierarchy/visibility/concurrency/semantic checking, the improvement of about 20~30% could be achieved.

I. 서 론

VHDL(VHSIC HDL)은 ANSI/IEEE 표준안[1, 2]으로 제정된 인공 정형언어(formal language)이다. 이 언어는 전자 하드웨어 시스템의 설계 영역인 문서화(documentation)와 시뮬레이션(simulation), 합성(synthesis) 및 정형 검증(formal verification) 등에서 이용되고 있는 실질적인(*de facto*) 표준 하드웨어 기술 언어(HDL)이다. 1987년에 표준화된 이후에 VHDL의 사용은 꾸준히 증가되어 왔으며[3], 대부분의 집적회로 설계 CAD 시스템에 VHDL 설계 검증 및 합성의 지원이 포함되어 있다. 기존의 VHDL 설계 검증 시스템들은 시뮬레이션을 수행하는 방식에서 대형 설계에 적합한 컴파일러와 디버깅을 위한 해석기로 구분되며[4], 컴파일러의 목적 코드로써 이식성 및 호환성이 우수한 C 언어[5]와 검증 속도를 높이기 위한 원시 코드[6, 7]를 이용하고 있다. 또한 VHDL 지원 범위에서도 설계 검증을 위한 LRM 전체(full-set) 사양의 시스템과 설계 합성을 위한 부분(subset) 사양의 시스템으로 나누어진다. 국내에서는 LRM의 부분 사양을 지원하는 VHDL 컴파일러[8] 및 해석기[9]의 개발이 보고되고 있으나, LRM 전체 사양을 위한 VHDL 설계 검증 시스템에 대한 연구 결과는 아직까지 발표된 적이 없다.

본 연구실에서는 VHDL 설계 검증을 위해서 VHDL 프로그램을 컴파일링하여 시뮬레이션 하는 시스템을 개발하고 있다. LRM 전체 사양을 지원하는 VHDL 설계 검증 시스템은 대형 설계에 대하여 충분한 검증 속도를 제공할 수 있는 컴파일러 방식으로 구현되고 있으며, 다양한 호스트 컴퓨팅 환경에 대한 VHDL 시뮬레이션의 호환성 및 이식성을 유지하고자 C 언어를 실행 목적 코드로써 사용하였다. 실제로 시뮬레이션을 위한 VHDL 컴파일러는 구성에서 전반부(front-end)와 후반부(back-end)로 나누어진다. VHDL 컴파일러의 전반부는 입력된 VHDL 설계 프로그램을 분석하여 LRM의 문법에 관한 정적(static) 오류를 검사하고 분석 결과를 중간 형식(intermediate format, IF) 데이터로 저장하며, 후반부는 분석 데이터를 검색해서 VHDL-to-C 사상을 통해서 실행 가능한 C 코드 형태의 VHDL 시뮬레이터를 생성한다. 본 논문에서는 VHDL-to-C 사상을 위한 분석 데이터를 생성하는 VHDL 컴파일러 전반부의 설계 및 구현에 대하여 기술하고자 한다.

VHDL 컴파일러의 전반부는 두 가지의 주요 기능을 효과적으로 처리하도록 설계되었다. 첫 번째는 입력된 VHDL 설계 프로그램을 LRM에 정의된 문법에 따라서 분석하고 확인하는 기능이다. 이를 위해서는 VHDL 프로그램에 대한 어휘 요소(lexical element)와 구문 생성(syntactic production)의 분석을 수행하고 문법 오류를 검사해야 한다. 본 연구에서는 어휘 요소를 분석하는 스캐너(scanner)와 생성 규칙에 따라 구문을 환원시키는 파서(parser)를 컴파일러 자동화 도구인 flex[10]와 bison[11]을 통해서 자동 생성시켰다. 이때 LRM의 VHDL 문법은 어의 문법을 포함한 확장 BNF(Extended Backus-Naur Form) 표기법으로 기술되어 있으나, 자동화 도구의 입력 파일은 정규표현(regular expression) 및 LALR(1)의 문법 기술 형식을 요구해서 많은 차이점을 가진다. 따라서 컴파일러 자동화 도구의 입력 파일을 기술하는 과정에서 많은 변형을 요구하게 되는데, 본 연구에서는 어휘 및 구문 문법 기술 단계에서 "어의 토큰(semantic token)"이라는 새로운 어휘 요소를 정의하여 VHDL 문법 기

술에 대한 변형을 줄이면서 문법 기술의 모호성에 따른 생성 과정의 충돌 현상을 최소화시키고자 했다. VHDL 컴파일러 전반부의 두번째 기능은 VHDL 프로그램의 분석 데이터를 구축하는 컴파일러 후반부의 VHDL-to-C 사상을 위한 전처리(preprocessing) 기능이다. VHDL 컴파일러의 전반부는 설계 프로그램을 분석하고 구문을 환원시켜 논터미날(nonterminal)과 터미날(terminal)로 이루어진 파싱(parsing) 트리를 구성한다. 이러한 분석 데이터는 컴파일러 후반부의 VHDL-to-C 사상 과정에서 입력으로 이용되는데, VHDL 구조체의 구성에 관한 정보는 트리 구조로 연결된 논터미날로써 나타나며 어의에 관한 내용은 터미날의 심볼, 타입 및 속성 등으로 저장된다. 이와 같은 분석 데이터는 C 프로그램 생성을 위해서 VHDL 프로그램의 구문 및 어의 정보를 효과적으로 표현하며 검색될 수 있도록 구성되어야 한다. 본 연구는 VHDL 프로그램을 분석하여 VHDL-to-C 사상을 위한 어의 정보를 추출하고 C 프로그램의 생성에 적합한 형태로 변환시키는 전처리 과정을 VHDL 컴파일러의 전반부에서 설계하였다. 또한 파싱 트리의 논터미날 및 터미날에 관한 어의 정보를 효율적으로 저장 및 검색하기 위한 분석 데이터 모델을 설계하였다. 이를 위해서 VHDL 프로그램에서 선언된 심볼의 유효영역(scope) 및 가시성(visibility)을 C 언어로 용이하게 사상될 수 있도록 계층적인 심볼 테이블을 설계하였으며, 논터미날의 사상 과정에서 하위 구조에 대한 검색을 한 단계로 최소화시킬 수 있도록 파싱 트리의 환원 과정에 하위 논터미날 및 터미날의 어의 정보를 상향식으로 상위 논터미날에 전달하는 메커니즘을 포함시켰다. 이밖에도 분석된 데이터를 설계 라이브러리로써 외부에 구축하고 분석 중인 VHDL 프로그램에 외부의 분석 라이브러리 데이터를 검색하여 분석 중인 데이터에 포함시키는 과정을 VHDL 컴파일러 전반부에서 처리하였다.

본 논문은 VHDL 시뮬레이션을 위한 컴파일러 시스템에서 전반부의 설계 및 구현에 대해서 기술한다. 먼저 VHDL의 어의 특성을 표현하기 위해서 설계된 분석 데이터 모델을 설명하고, VHDL 컴파일러의 전반부의 어휘 분석기와 구문 생성기를 설계 및 구현하는 과정에 대하여 논한다. 설계 및 구현된 VHDL 컴파일러 전반부의 전처리 분석 성능을 Validation Suite를 통해서 평가하고, 마지막으로 본 연구의 결론을 맺는다.

Ⅱ. VHDL 분석 데이터 모델링

VHDL 시뮬레이션 어의를 C 프로그램으로 표현하기 위해서는 VHDL의 특성을 분석하여 VHDL-to-C 사상에 적합한 분석 데이터를 구축하여야 한다. VHDL 컴파일러 전반부에서 분석된 데이터는 논터미날(nonterminal)의 AST(Abstract Syntax Tree)와 터미날(terminal)의 심볼 트리로 구성된다. 또한 이와 같은 분석 데이터를 VHDL 설계 프로그램에서 재사용하기 위해서 라이브러리를 외부 저장 매체에 구축하는데, 중간 데이터 형식을 이용한 데이터의 저장 및 검색, 복원 과정이 VHDL 컴파일러 전반부에서 수행된다.

2.1 AST 데이터

VHDL 프로그램에 대한 어휘 및 구문 분석 과정에서 유도되는 파스 트리는 어휘를 단말 노드로 하고 구문 규칙 단위를 중간 노드로 표현한다. 이같은 분석 데이터는 불필요한 정보를 갖기 때문에 메모리 오버헤드와 컴파일러의 속도 저하를 야기시킨다. 따라서 일반적으로 컴파일러에서는 의미를 갖는 터미날과 이를 연결하기 위한 최소한의 논터미날만을 이용해서 파스 트리를 표현하는 AST 데이터를 유도하게 된다. 논터미날은 VHDL의 구문 규칙 단위를 나타내며 계층적인 구문 생성 관계에 따라서 VHDL 프로그램의 AST를 구성하게 된다. AST의 논터미날 구조체는 VHDL 컴파일러의 분석 과정에서 다음의 주요 역할을 담당한다.

- 구문 생성 규칙의 단위로서 VHDL 구문의 계층성을 표현하기 위한 단위
- 하위 논터미날 및 터미날에 대한 정보 검색을 위한 트리 구조의 구성 단위
- 어의 검사 및 속성 관리의 처리 단위
- 하위 터미날의 어의 정보를 상위 논터미날로 전파시키기 위한 중간 저장 단위

이외에도 AST의 논터미날은 VHDL 컴파일러 후반부의 VHDL-to-C 사상 과정에서 하위 트리의 구성

및 어의 정보를 검색하여 C 언어로 사상하는 처리 단위로써 이용된다.

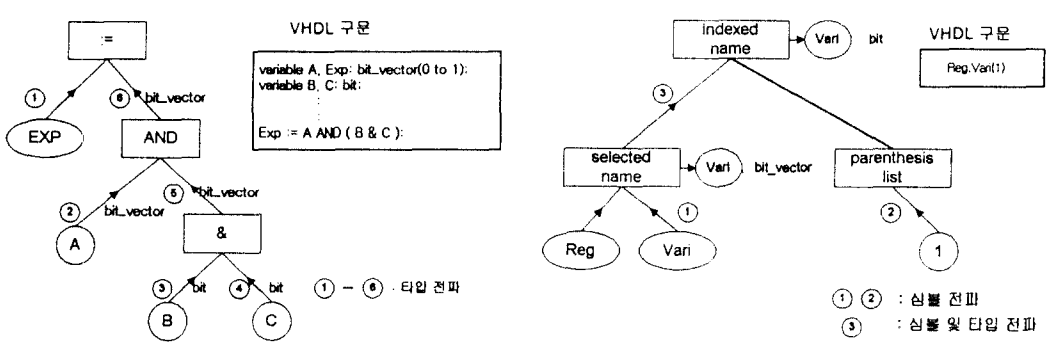
분석 AST 데이터의 터미날은 신호/변수/상수/서브프로그램/리터럴/타입 심볼의 정보를 저장하고, 새로운 타입의 경우에는 타입 트리에 등록시키며, 분석된 객체의 속성 정보를 심볼에 저장한다. 논터미날이 생성되면 하위 AST의 타입과 심볼 등의 어의 정보를 상향식으로 전파시켜 문법 검사를 위한 AST의 하향식 순회 검색 오버헤드를 감소시킨다. 그림 1(a)는 연산자 논터미날에 대한 타입 전파를 AST에서 보이고 있다. 또한 그림 1(b)에서와 같이 색인(index) 이름 및 유효 영역의 선택(selection) 확장을 이용한 VHDL 이름 구문에서 하위 AST로부터의 심볼 및 타입 전파는 AST의 별도 검색 과정이 없이 분석 과정에서 해당 논터미날의 심볼과 타입이 결정되도록 한다. 이와 같은 AST의 타입 및 심볼 전파를 이용하는 VHDL 논터미날 구문들이 표 1에 정리되어 있다.

VHDL의 어의 문법은 객체의 사용 내역(contexts) 및 속성에 관한 정적 규칙과 객체의 내용(contents) 및 동작에 관한 동적 규칙으로 구분된다. 시뮬레이션 과정에서 디버깅되는 동적 어의 규칙과 달리 VHDL 정적 어의 규칙은 컴파일러 전반부의 분석 과정에서 AST의 논터미날 및 터미날 정보를 이용해서 검사될 수 있다. VHDL은 객체의 사용내역에 관해서 정적 어의 규칙을 엄격하게 정의하여, VHDL 프로그램의 분석 과정에서 수행되는 정적 어의 검사는 시뮬레이션 과정에서 발생할 수 있는 많은 실행 시간(run-time) 오류를 사전에 검출해서 전체 프로그램의 디버깅 오버헤드를 크게 감소시키도록 한다. VHDL 컴파일러 전반부는 구문 분석 과정에서 환원된 논터미날에 대해서 Type, Flow, Uniqueness 및 Name 등에 관한 정적 어의 검사를 수행한다[12].

VHDL 분석 과정에서 구성되는 AST는 필요에 따라서 컴파일러 후반부의 VHDL-to-C 사상에 적합한 형태로 트리 구조가 변환되기도 한다. 실제로 VHDL은 이름(named) 및 위치(position) 연관을 지원하나, C 언어는 위치 연관만을 제공하기 때문에, VHDL의 이름 연관을 C 언어의 위치 연관으로 변환시키는 과정이 VHDL-to-C 사상을 위해서 전처리된다.

2.2 심볼 트리 데이터

VHDL 컴파일러 전반부의 분석 과정에는 VHDL-to-C 사상을 위한 분석 데이터의 터미날 정보를 효과적으로 관리할 수 있는 데이터 구조의 설계가 요구된다. 일반적으로 분석 데이터의 터미날은 컴파일러 전반부에서 분석한 토큰을 나타내는 단위이며 심볼과 타입 및 속성 정보를 포함하게 된다. 심볼은 사용자가 정의한 인식자(identifier) 및 리터럴(literal)에 대한 컴파일러 내부의 구조체로서 그림 2(a)와 같이 이름과 객체 및 타입, 속성 정보를 표현하며 선언 영역 및 가시성을 위한 계층 정보를 나타낸다. 타입은 VHDL의 타입 선언으로 정의된 심볼을 관리하기 위한 구조



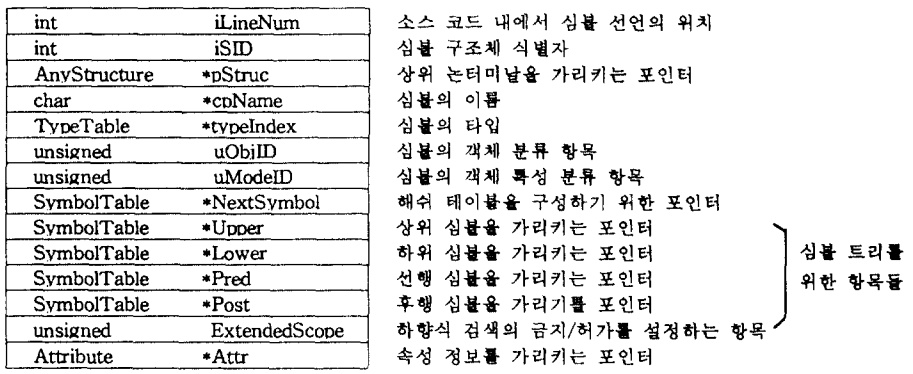
(a) 타입 전파 (b) 심볼 전파

그림 1. AST의 타입 및 심볼 전파
Fig. 1. Type and symbol propagation in AST

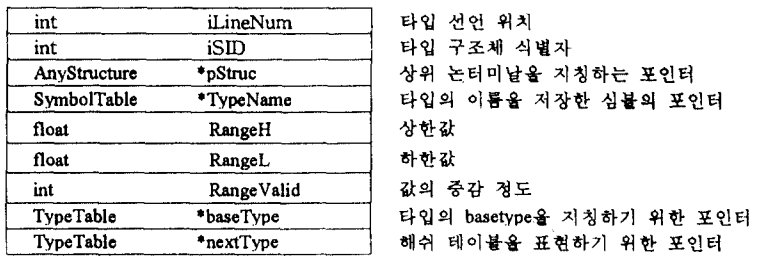
표 1. 타입 및 심볼 전파의 VHDL 구문

Table 1. VHDL syntax of type and symbol propagation

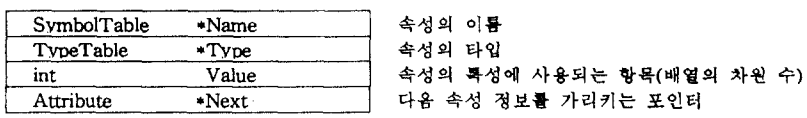
전파 항목	적용 구문 분류	VHDL 구문
타입	expression 구문	expression, relation, shift_expression(93), simple_expression, term, factor, primary 구문
	기타 구문	signal assignment, variable assignment 구문
		signal/variable/constant declaration에서 초기값을 포함한 구문
		port_map, generic_map 구문
		function_call, procedure_call 구문
		return 구문
		condition 구문
		name 구문
심볼	name 구문	name, selected_name, attribute_name, indexed_name, slice_name (signal_name, variable_name, subprogram_name, constant_name)



(a) 심볼 구조체



(b) 타입 구조체



(c) 속성 구조체

그림 2. 터미날 정보를 위한 데이터 구조체

Fig. 2. Data structures for the terminal information

체로서 그림 2(b)와 같이 심볼 및 연산(subprogram, allocation, assignment, qualification)이 가질 수 있는 값의 범위와 유형에 관한 특성을 나타내며, 타입의 특성들 사이의 종속 관계를 표현한다. 터미날의 속성은 모든 심볼에 대해서 VHDL이 정의한 내재적 속성 정보를 저장하거나 사용자가 임의로 설정한 속성 정보를 표현하기 위한 구조체로서 이름, 타입 및 특성 항목 등으로 그림 2(c)와 같이 구성된다.

일반적인 프로그램 분석 과정에서는 AST 데이터와는 별도로 심볼의 효율적 등록 및 검색을 위해서 심볼 테이블이 구성되어 관리된다. VHDL은 그림 3에 나타난 바와 같이 선언 영역간에 고유의 계층 관계를 정의하고 있어서, 영역 심볼들 사이에 상하위 계층성 및 중첩 관계 등이 표현되어 선언된 심볼의 유효 범위 및 심볼의 가시성을 확인하는데 이용된다. 본 연구는 이와 같은 VHDL 심볼에 대한 계층적 구조의 필요성과 함께 심볼의 저장 및 검색의 오버헤드를 고려해서 해쉬(Hash) 테이블의 리스트 구조 및 계층적 트리 구조의 심볼 테이블을 설계하였다. 해쉬 구조는 심볼의 신속한 검색 및 저장을 위하여 이용되며, 트리 구조는 심볼의 영역 및 가시성을 효율적으로 처리하기 위해서 구성되었다. 심볼의 해쉬 구조는 1024 크기의 테이블에 심볼 인식자의 해쉬 코드를 색인화시키고, 동일한 해쉬 코드의 심볼들을 입력 순서에 따라서 단일 연결 리스트 구조로써 등록시키게 된다. 트리 구조는 계층적 표현을 통해서 라이브러리 및 설계 단위를 관리하며, 각 설계 단위는 계층화된 중간노드의 선언 영역과 종단노드의 선언으로 표현된다. 심볼 테이블의 계층적 트리는 VHDL 선언 영역의 상하위 및 중첩 관계(그림 3)를 나타내는데, 선언 영역은 설계 라이브러리와 LRM의 선언 지역(설계 단위, 서브프로그램, 프로세스문, 블럭문, 생성문, 루프문, 콤포넌트 선언, 레코드형 선언)으로 구성된다. 또한 다중 정의를 처리하기 위해서 열거형 선언은 독립된 선언 영역으로 구분된다. 이와 같은 트리 구조의 심볼 테이블은 VHDL의 계층성 및 가시성을 직접적으로 나타내기 때문에 AST에 비해서 심볼 관리를 보다 효과적으로 지원한다. 그림 4는 VHDL 프로그램에 대해서 심볼 트리 구조 및 AST 구조가 유도된 결과를 분석 진행 중과 분석 완료 후로 비교해서 나타내었다. 분석이 진행중일 때에 하향식으로 구성되는 심볼 트리는 심볼 S의 정보를 검색하는 것이

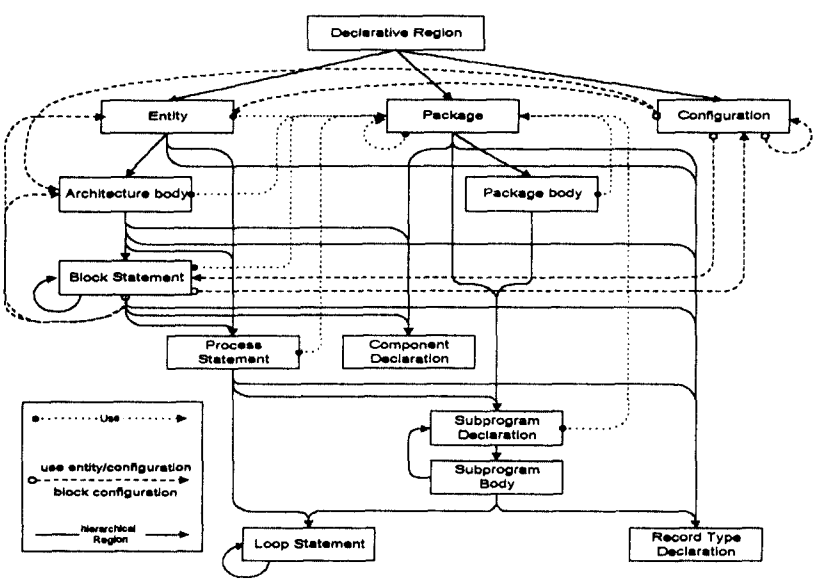


그림 3. VHDL 선언 영역의 계층성과 확장
Fig. 3. Hierarchy and expansion of VHDL declaration

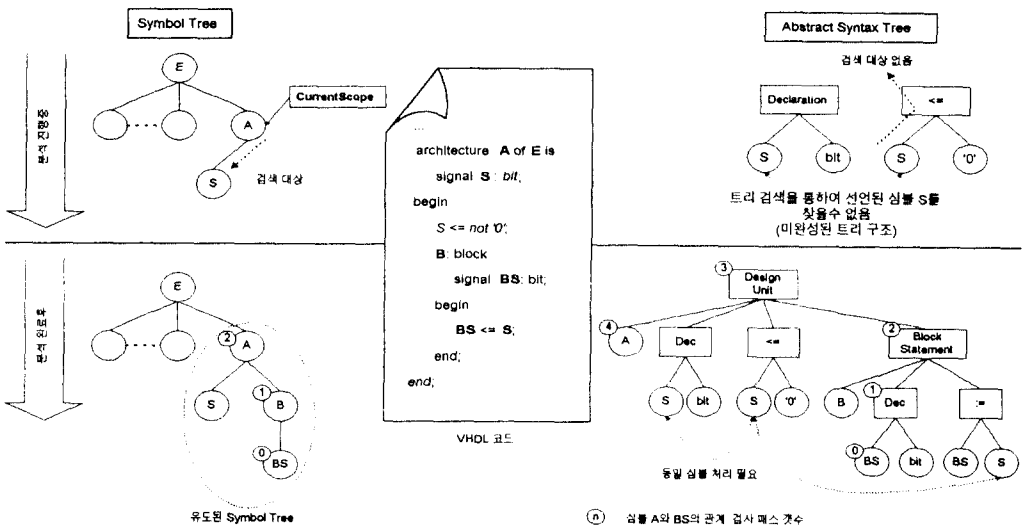


그림 4. 심볼 트리와 AST의 비교
Fig. 4. AST vs. Symbol tree

가능하나, 상향식으로 구성되는 AST는 트리 구조가 미완성이기 때문에 심볼의 검색이 VHDL 설계 단위의 분석이 완료될 때까지 지연된다. 또한 분석이 완료된 후의 심볼 BS의 검색에서 심볼 트리의 순회(0→1→2)는 AST의 순회(0→1→2→3→4)와 비교해서 효과적인 것을 보인다.

계층적인 VHDL 설계 단위의 병행문들은 앞에 설명한 바와 같이 하드웨어의 병행 동작들을 표현하기 위하여 각각 스레드마다 독립된 스택 구조를 요구한다. 반면에 C 언어는 순차적 단일 스레드를 위한 단일 스택 구조만을 제공한다. 따라서, VHDL 컴파일러 후반부의 VHDL-to-C 사상에서는 시뮬레이션 커널이 병행적으로 수행시킬 각각의 C 의사 프로세스 함수들에 독립된 스택 구조를 관리할 방법이 요구된다. 본 연구에서는 병행적인 VHDL 지역 심볼들을 유지시키기 위해서 의사 프로세스의 심볼들을 시뮬레이터 커널의 전역 심볼로써 표현한다. 이때 지역 심볼들의 전역 선언으로 발생될 수 있는 인식자 중복은 심볼이 선언된 계층적 영역의 인식자들을 이용해서 구분시켜 줄 수 있다. 이와 같은 심볼의 전역 선언 및 인식자 확장은 VHDL 컴파일러 전반부의 분석 과정에서 구성하는 심볼 트리를 통해서 해결된다. 심볼 트리에는 VHDL 선언 영역이 계층화되어 있으며 각 선언 영역의 심볼이 등록되어 있기 때문에 심볼 트리의 하향식 순회에 따른 선언 영역의 인식자 문자열을 각 심볼의 확장자로 이용하면 모든 심볼이 시뮬레이터 커널의 선언 영역에서 고유의 인식자를 갖게되는 것이다. 그림 5는 VHDL 다중 스레드의 동형 이의어에 대한 심볼 인식자의 구분 방법을 나타내고 있다. 3개의 동형 이의어 심볼 S는 심볼 트리의 계층 정보를 이용해서 E→A→S, E→A→P0→S, E→A→P1→S로 구분될 수 있다. 이와 같은 심볼 인식자의 확장은 VHDL 병행 수행 단위인 프로세스 P0와 P1에 대한 C 의사 프로세스의 심볼을 커널의 전역 선언으로 처리하더라도 독립적으로 유지 및 관리되어, 실질적으로 VHDL의 다중 정적 스택 구조를 C 언어의 단일 정적 스택 구조로써 처리할 수 있도록 한다.

VHDL 컴파일러 전반부는 프로그램의 분석 과정에서 심볼 트리를 이용해서 선언 영역 및 심볼을 등록하고 심볼의 가시성을 확인하기 위한 검색을 수행한다. 그림 5는 심볼 트리에 대한 등록 및 검색 과정을 보이고 있다. 새로운 선언은 중복선언 유무의 확인을 위하여 현재 선언 영역에 대한 검색을 수행한 후 현재 선언 영역의 중간노드에 종단노드로서 등록

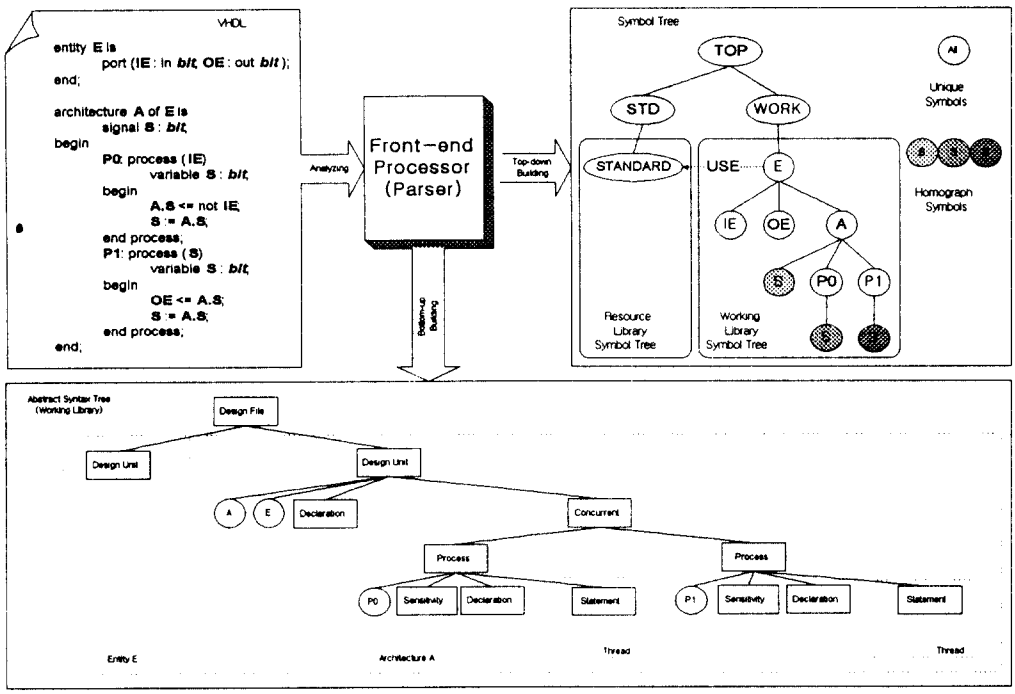


그림 5. 분석 과정의 심볼 트리 구성 및 활용

Fig. 5. Derivation and utilization of symbol tree in the VHDL compiler front-end

표 2. VHDL의 유효 영역 확장

Table 2. Expansions of VHDL scope

유효 영역의 확장	VHDL 기술
library내의 주 설계 단위 (Use 문)	library_name.primary_unit use STD.STANDARD;
entity를 이용한 architecture body (Use entity 문)	entity(architecture body) use entity GATE(BEHAVIOR);
package 내의 모든 선언들 (selected name)	package_name.declaration_name TEXTIO.INPUT
record 자료형 내의 요소 선언들 (selected name/aggregate)	record_name.element/record_name(element,...) day.hour/day(hour => 1, min => 2, sec => 3)
attribute 선언들 (attribute name)	prefix'attribute_designator BIT'LOW
subprogram 내의 formal 파라메터 (association list)	subprogram(parameter,...) WRITE(L => Data, VALUE => '0')
component 내의 local generic/port (association list)	component configuration C:CMP port map (IN0 => PIN1, OUT2 => PIN2);
entity 내의 formal generic/port	block configuration for C:CMP use entity GATE(BEHAVIOR) port map (IN0 => PIN1, OUT2 => PIN2);
block 내의 formal generic/port	block header 구문 : 위와 동일함

된다. 참조된 심볼에 대해서는 심볼 트리에 대한 direct 검색으로써 현재 선언 영역으로부터 상향식으로 선언 영역을 검색하여 선언 심볼의 가시성을 검사한다. 유효영역이 확장된 경우에 대해서는 direct 검색을 수행한 후에 선택된 선언 영역으로부터 하향식으로 선언 영역을 검색하는 selection 방법에 의해서 선언 심볼을 확인하게 된다. 이와 같은 심볼 트리를 이용한 VHDL 유효 영역 및 가시성의 분석과 검색 과정은 VHDL 프로그램의 문법 검사와 VHDL-to-C 사상을 위한 전처리로써 수행된다.

VHDL 선언의 유효 범위는 선언의 참조가 가능한 범위를 가리키는데, 기본적으로 기술된 선언 위치로부터 계층적인 하위 선언 영역의 끝까지를 나타낸다. VHDL에서는 기본적인 블록 구조의 선언 영역 이외에도 표 2의 선언들은 선택적으로 유효 범위를 확장시킬 수 있다. 이외에도 구성 선언은 엔티티 및 아키텍쳐 몸체(use entity 문)와 다른 구성 선언(use configuration 문)의 유효 영역을 확장시킬 수 있으며, 기존에 분석되어 있는 설계 라이브러리(use library 문)를 유효 영역에 포함시키는 것이 가능하다. 이와 같은 VHDL 선언 영역의 유효 범위에 대해서 식별자의 선언 유무를 검사하게 된다. 식별자의 선언이 유효 범위에서 발견되었을 때 가시성이 확인되었다고 하며, 선언에 관한 내용을 검색하는 것이 가능하다.

VHDL에서 가시성의 확인은 선언이 하나 이상 존재함을 뜻하는데, 두개 이상의 선언이 유효 범위에 있는 경우는 다중 정의(overloading)나 동형 이의어(homograph)에서 발견된다. 다중 정의는 두개 이상의 선언에서 어떤 선언이 문맥에 적합한지를 결정하는 과정을 요구하며, 동형 이의어에서는 VHDL 기술에 따라서 유효 영역의 선언을 선택하는 처리 과정이 필요하다. 서브프로그램, 열거형 리터럴 및 이름(VHDL 93')에 대한 다중 정의의 경우는 direct 방법에 따라서 심볼의 signature 정보나 타입 정보를 이용해서 적합한 선언을 결정하게 된다. 실제로 VHDL 컴파일러의 전반부는 다중 정의된 심볼에 대해서 타입 검색 및 확인을 통하여 원하는 심볼이 인식되도록 전처리 과정을 수행한다. 다중 정의가 가능한 심볼을 구분하기 위해서 구문 규칙의 중간 수행 함수를 이용하여 다중 정의 플래그(overload flag)를 설정하고, 분석 과정에서는 동일한 선언 영역에서 동일 이름의 심볼이 등록될 수 있도록 하여 검색 과정에서 심볼의 이름과 sig-

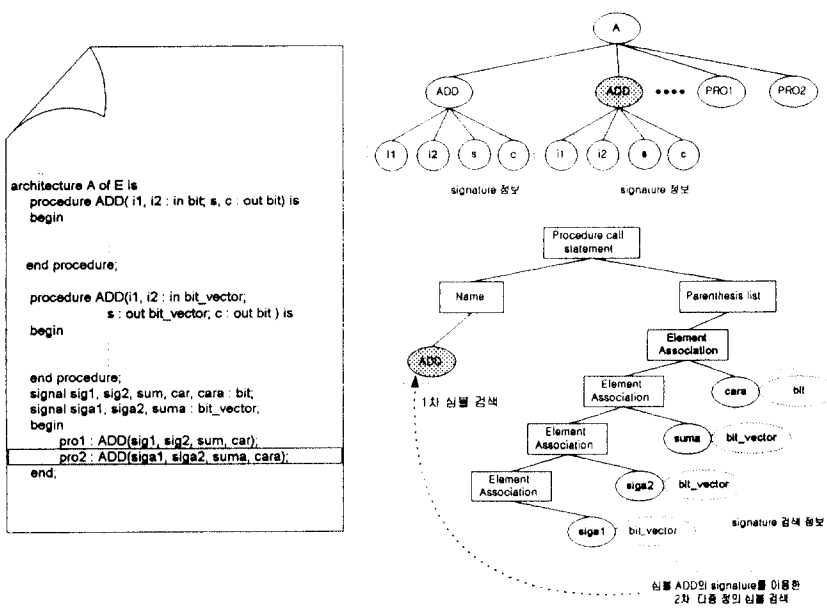


그림 6. 다중 정의 심볼의 검색

Fig. 6. Retrieval of overloaded symbol

nature 정보 및 타입 정보를 이용하여 검색하는 추적 과정을 수행한다. 그림 6과 같은 서브프로그램에서 이름이 일치하는 ADD 심볼이 검색되면, 등록된 피연산자의 타입(bit_vector, bit_vector, bit_vector, bit)과 일치하는 가를 검색하는 문법 검사가 이루어진다. 열거형 리터럴의 심볼에 대해서는 이름이 일치하는 심볼을 검색하여 연산자의 또 다른 피연산자 심볼의 타입과 일치하는 가를 확인하는 검색 과정을 수행한다.

2.3 라이브러리 데이터

VHDL은 분석 데이터의 라이브러리 구축 및 재사용을 지원하고 있다. 본 연구에서는 VHDL 컴파일러의 전반부가 분석된 데이터를 외부 매체에 저장하고 필요에 따라서 복원시킬 수 있도록 분석 데이터의 저장과 검색을 위한 중간 데이터 구조와 지원 함수를 설계 및 구현하였다. 실제로 VHDL 컴파일러에서 재사용될 설계 라이브러리는 분석 데이터와 C 프로그램의 두 가지 형태로써 생각할 수 있다. C 프로그램 라이브러리는 VHDL 설계가 컴파일러를 통해서 분석 및 사상되어 C 모델링된 결과를 화일로써 저장시켜서 구성된다. 사상된 C 모델 라이브러리의 재사용은 VHDL 컴파일러 후반부에서 C 화일의 검색을 통해서 간단하게 처리될 수 있다. 그러나 이와 같은 C 모델 라이브러리의 재사용은 VHDL 컴파일러 전반부에서 분석 중인 VHDL 프로그램과 C 모델을 생성시킨 분석 데이터와의 상호 연관성이 설정되어야 가능하다. 이를 위해서 분석 데이터의 라이브러리 저장 및 검색 방법이 설계되었다. 앞에 기술한 바와 같이 분석 데이터는 AST 데이터와 심볼 트리 데이터로 구성되는데, 분석된 AST 데이터는 이미 C 모델 라이브러리로써 구현되었기 때문에 분석 중인 VHDL 프로그램의 AST 데이터와의 결합은 컴파일러 후반부에서 중복 처리를 가져올 수 있다. 반면에 분석된 터미날 데이터는 VHDL 심볼의 선언 및 가시성, 타입 및 속성 정보로 구성되어 있기 때문에 분석 중인 VHDL 프로그램과의 관계가 확인되어야 하고, 상호 연관성이 설정되어야 한다. 따라서 본 연구에서는 심볼의 계층 정보, 타입의 분류 정보와 심볼의 속성 정보 등으로 구성된 심볼 트리 데이터의 라이브러리 저장 및 검색 방법을 구현하였다. 분석된 터미날 데이터를 라이브러리로써 저장 및 검색하기 위해서 그림 7의 중간 형식 데이터가 설계되었다. 라이브러리 데이터의 구조체는 라이브러리 인식자, 색인 테이블과 데이터 저장 레코드의 3 부분으로 구성되며, 색인 테이블 및 데이터 저장 레코드 부분은 심볼과 타입, 속성 및 리터럴에 대해서 각각 나누어진다.

색인 테이블은 메모리 복원 속도를 향상시키기 위하여 그림 8과 같이 화일 오프셋(offset)과 포인터 정보로 구성되어 터미날 데이터간의 상호 연결 정보를 라이브러리 화일에 저장된다. 저장 과정에서 색인 테이블 내에 터미날 데이터의 메모리 주소 순으로 정렬되어 등록되고, 심볼/타입/속성/리터럴 데이터의 색인 테이블의 등록이 완료된 후에 색인 테이블 및 터미날 데이터를 라이브러리 화일에 저장하게 된다. 복원 과정에서는 라이브러리 화일에서 터미날이 등록된 색인 테이블들을 복원하여 화일내의 터미날 데이터를 정렬된 순서로 복원하며, 모든 터미날의 메모리 복원이 완료된 이후 각각의 터미날 데이터내의 포인

char	*FileIdentifier	중간 데이터 단위의 이름
char	*FileInformation	생성 날짜 등의 추가 정보
IndexTable	SymbolIndexTable[ ]	심볼의 메모리 주소를 색인키로 사용한 색인 테이블
IndexTable	TypeIndexTable[ ]	타입의 메모리 주소를 색인키로 사용한 색인 테이블
IndexTable	AttributeIndexTable[ ]	속성의 메모리 주소를 색인키로 사용한 색인 테이블
IndexTable	LiteralIndexTable[ ]	리터럴의 메모리 주소를 색인키로 사용한 색인 테이블
SymbolTable	*SymbolRecord[ ]	심볼 구조체 저장(binary data, unique length)
TypeTable	*TypeRecord[ ]	타입 구조체 저장(binary data, unique length)
Attribute	*AttributeRecord[ ]	속성 구조체 저장(binary data, unique length)
char	*LiteralRecord[ ]	리터럴 데이터 저장(text data, variable length)

그림 7. 라이브러리 데이터의 중간 형식
Fig. 7. Intermediate format of library data

터 정보를 복원된 새로운 메모리 주소로 갱신하여, 심볼 트리/타입 트리/심볼의 리터럴 정보 및 속성 정보의 복원을 완료한다.

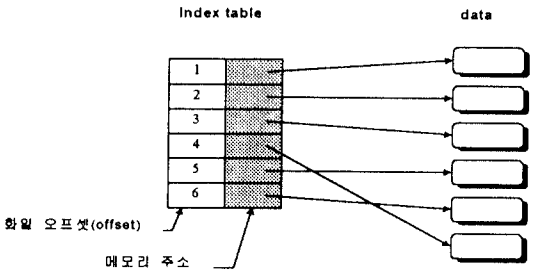


그림 8. 색인 테이블의 구성
Fig. 8. Construction of index table

Ⅲ. VHDL 컴파일러 전반부 설계

컴파일러 전반부의 기본 동작은 어휘 분석의 스캐너(scanner)와 구문 분석의 파서(parser)를 통해서 수행된다. 또한 스캐너와 파서에 포함된 수행(action) 함수들은 분석 데이터에 대해서, VHDL 프로그램의 문법 오류를 검사 및 보고하며, VHDL-to-C 사상에 적합한 데이터 형태로 구축하는 후반부에 대한 전처리 과정을 수행한다. 여기서는 기존의 컴파일러 자동화 도구를 이용한 스캐너와 파서의 설계에 대해서 기술하고, 구현된 스캐너와 파서의 구성 및 동작에 관해서 논한다.

3.1 Front-end의 설계

VHDL 컴파일러 전반부의 스캐너와 파서는 어휘 분석기 생성기 flex v.2.5와 구문 분석기 생성기 bison v.1.25를 이용해서 그림 9와 같이 발생된다. Bison은 "table.h"를 포함한 "vhdl.yacc" 화일을 입력으로 "parser.h"와 "parser.c"의 파서 프로그램을 생성한다. "vhdl.yacc" 화일은 VHDL 구문 문법의 기술과 파서의 구문 데이터 처리 과정으로 구성되며 "table.h" 화일은 분석 데이터의 구조체를 정의한다. VHDL 구문 문법은 구문 규칙의 생성(production) 단위로써 AST를 구성하는 터미날 및 논터미날을 정의하고, VHDL의 구문 생성을 CFG(context-free grammar) 표현의 규칙으로 기술한다. 구문 데이터의 처리 과정은 "vhdl.yacc" 화일의 사용자 정의 수행 함수로써 생성된 푸쉬다운 오토마타(push-down automata, PDA) 파서에 대해서 구문 분석 기능을 지원한다.

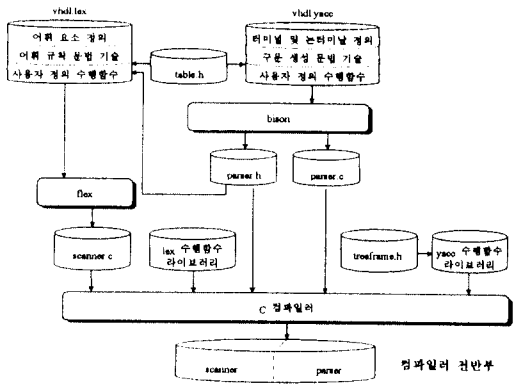


그림 9. 스캐너 및 파서 생성 시스템
Fig. 9. Compiling system of scanner and parser

생성된 파서 프로그램의 "parser.h" 화일은 분석 데이터의 터미날로서 저장될 토큰 문자열 및 고유번호(ID)를 정의해서 어휘 분석기 생성기 flex에 입력된다. Flex는 "table.h" 및 "parser.h" 화일을 포함한 "vhdl.lex" 화일을 입력으로 스캐너 프로그램을 생성한다. VHDL 어휘 문법과 사용자 정의 수행 함수로 구성된 "vhdl.lex" 화일에서 어휘 문법은 정규 표현으로 정의된 VHDL 어휘 요소와 구문의 기본 문자열을 토큰으로 정의한 VHDL 어휘 규칙으로 기술되며, 사용자 정의 함수는 유한 오토마타(finite automata, FA) 스캐너에 대해서 어휘 분석 기능을 지원한다.

일반적인 프로그램 분석 과정에서는 구문 구성의 기본 단위를 입력 소스 코드의 문자열에 대해서 토큰으로 정의한다. 구문 생성의 기본 단위인 토큰을 이용해서 VHDL의 LRM 구문 문법은 확장된 BNF (Backus Naur Form) 표기법으로 기술되어 있다. 실제로 구문 분석기 파서를 생성하기 위하여 bison의 입력 화일 "vhdl.yacc"는 LALR(1) 형태의 문법 기술을 요구하기 때문에, BNF 형식의 VHDL 구문 문법 기술을 변형시키는 것이 요구된다. 일반적으로 문법 기술의 변형은 대치 및 병합, 분리 등의 방법으로 이루어진다. 그러나 이와 같은 문법 기술의 변형에서

사용자는 오류를 범해서 파싱 충돌(conflict)를 발생시키는 경우가 많이 발생한다. 또한 생성된 파서의 구문 분석 과정이 분석 데이터에서 논터미날 및 터미날 정보의 검색 오버헤드를 증가시키는 문제를 발생시키기도 한다. 본 연구에서는 문법의 변형을 최소화시키고 분석 데이터의 검색 오버헤드를 줄이기 위해서, 구문 생성 규칙의 기본 단위인 토큰을 세분화시켰다. 이름 토큰의 세분화를 위해서 VHDL LRM의 BNF 구문 규칙에 이탤릭체로 기술된 객체 정보와 VHDL 프로그램의 선언과 어느 영역에서 이름 토큰이 사용되는 가를 나타내는 구문 정보를 이용한다. 실제로 구문 정보는 토큰에 대해서 선언된 이름과 참조의 이름을 구분해주며, 객체 정보는 다중 정의 가능 여부와 객체에 대해서 신호/변수/상수/타입/서브프로그램의 이름과 기타 이름으로 나누어준다. 이와 같은 객체 및 구분 정보는 VHDL 어의를 나타내며, VHDL 어의를 반영한 이름 토큰의 분류는 표 3의 "어의 토큰"(semantic token)으로써 정의된다.

사용자가 정의한 VHDL 어의 토큰을 이용한 구문 분석 과정에서는 파서가 다음 토큰을 요청할 때 구문 및 객체 정보를 추출해서 예상되는 토큰의 어의 정보로써 스캐너에 전달하고 스캐너가 보낸 토큰이 예상된 어의를 갖는지를 확인하는 역할을 담당하며, 스캐너는 파서가 보낸 어의 정보를 이용해서 입력 소스의 문자열로부터 추출된 이름 토큰을 분류 및 결정해서 파서로 보내준다. 이를 위해서 "vhdl.yacc" 화일의 구문 규칙 기술은 그림 10과 같이 표현된다. 파서는 구문 생성 과정에서 스캐너에서 토큰을 받은 후에 구문을 환원(shift/reduce)하고 해당 토큰에 기술된 중간 수행 함수를 수행하기 때문에, 다음 토큰의 어의 정보를 파싱 스택으로부터 추출하기 위하여 어의 토큰 (*sene_token*) 앞의 토큰에 중간 수행 함수를 기술하여야 한다. 실제로 어의 토큰을 이용하여 VHDL 구문

```
rule_name  :  token (mid_action) token sem_token token                    (action)
           |  sub_rule_name (mid_action) token sem_token sub_rule_name2    (action)
```

그림 10. 어의 토큰을 이용한 구문 규칙 기술
Fig. 10. Syntax description for semantic token

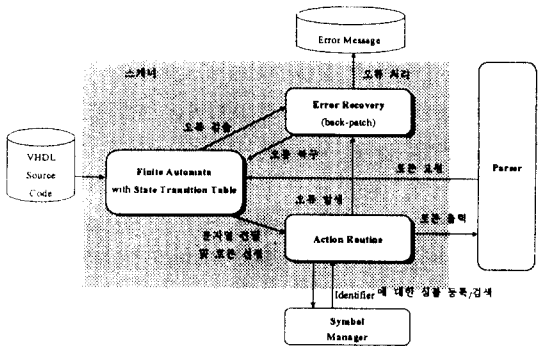
표 3. 구문 및 객체 정보의 어의 토큰
Table 3. Semantic token with syntax and object information

VHDL 어의 정보			이름 토큰의 세분화	VHDL 이름 토큰의
구문 정보	객체 정보			
선언	다중 정의 불가		NEW_IDENTIFER	· design unit/block문/process문/loop문/generate문의 선언 영역 이름 · 신호/변수/상수/자료형 선언 이름 · component 선언 이름 · attribute 선언 이름 · alias 선언 이름 · port/generic/interface 선언 이름 등
	다중 정의 가능		OVERLOADED_IDENTIFIER	· 서브프로그램 선언 이름
참조	특성	신호	SIGNAL_IDENTIFIER	· port_map의 association_list 이름 · sign_assignment의 target 이름
		상수	CONSTANT_IDENTIFIER	· generic_map의 formal designator 이름
		변수	VARIABLE_IDENTIFIER	· variable_assignment의 target 이름
		자료형	TYPE_IDENTIFIER	· type_mark의 타입 이름
		서브프로그램	SUBPROGRAM_IDENTIFIER	· procedure_call/function_call의 서브프로그램 이름
	이름		REFERRED_IDENTIFIER	· 객체 특성을 갖지 않는 기타 이름

표 4. 어의 토큰을 이용한 VHDL 구문 문법의 기술 결과
Table 4. VHDL syntax description results with semantic token

VHDL 항목	VHDL'87	VHDL'93	
		기존 방법	어의 토큰의 방법
토큰 수	113	130	137 (+7)
논터미날	156	241	263 (+22)
구문 규칙	512	670	706 (+36)
구문 상태	1096	1386	1454 (+132)

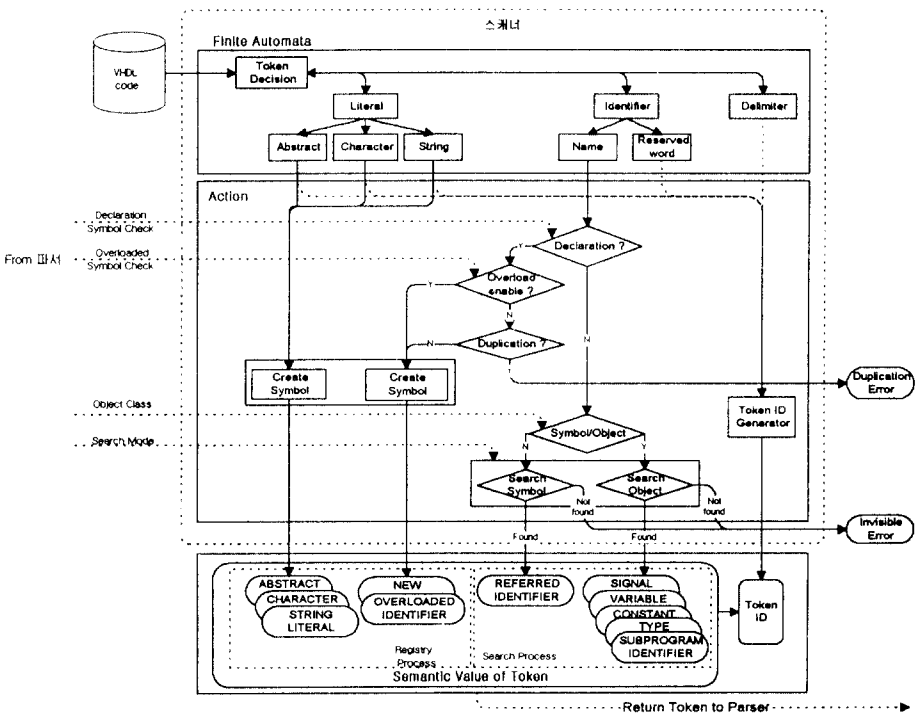
문법을 LALR(1) 형식으로 기술한 결과가 표 4에 비교되어 있다.



(a) 스캐너의 구성

3.2 Front-end의 구성 및 동작

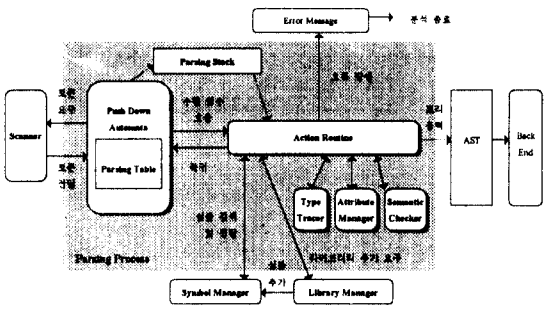
컴파일러 자동화 도구에 의해서 생성된 VHDL 스



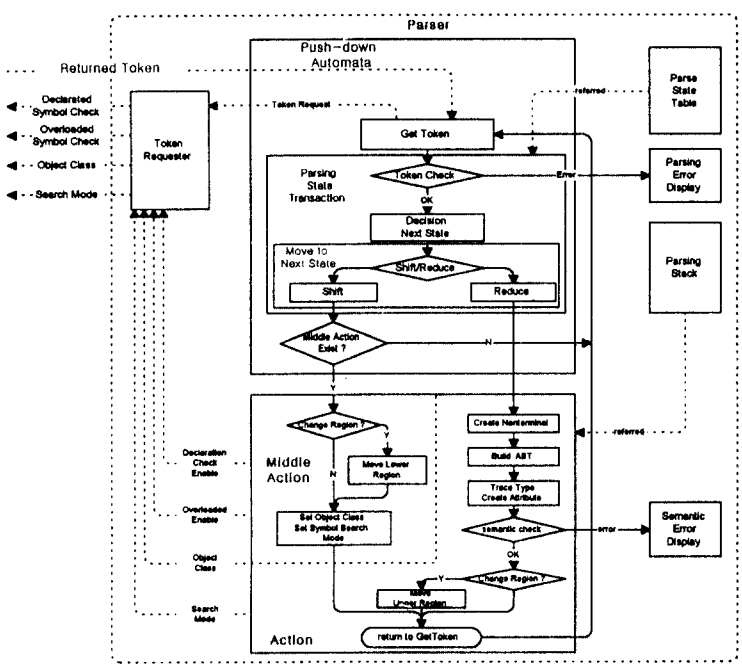
(b) 스캐너의 동작

그림 11. VHDL 스캐너의 구성 및 동작
Fig. 11. Configuration and operations of VHDL scanner

캐너와 파서는 VHDL 입력 소스 프로그램을 어휘 및 구문 분석하여 컴파일러 후반부의 VHDL-to-C 사상을 위한 입력 데이터를 구성한다. VHDL 스캐너의 구성 및 동작이 그림 11에 나타나 있다. 스캐너는 기본적으로 유한 오토마타 어휘 분류기와 오류 복구기 및 수행 함수로 그림 11(a)와 같이 구성된다. Flex가 VHDL 어휘의 정규 표현에서 생성한 유한 오토마타는 내부 상태 천이표(state transition table)를 이용해서 VHDL 입력 소스에서 토큰 문자열을 구분하여 VHDL 토큰의 분류, 등록 등의 처리를 수행한다. 오류 복구기는 VHDL 어휘 분석 과정에서 type_mark의 속성을 위한 구분자 토큰과 character 리터럴의 구분자 토큰이 동일한 특수 문자 apostrophe(')를 해결하는 문제를 flex의 back-patch 기능을 통해서 복구하



(a) 파서의 구성



(b) 파서의 동작

그림 12. VHDL 파서의 구성 및 동작
Fig. 12. Configuration and operations of VHDL parser

는 기능을 담당한다. 수행 함수에서는 유한 오토마타가 구분한 토큰을 파서에서 예상한 구문 및 객체에 관한 어의 정보를 이용하여 다시 토큰을 결정하고 고유 번호를 부여하며, 토큰을 분석 데이터의 터미날을 구성하는 심볼 테이블 및 트리에 저장 및 검색하는 과정을 처리한다. 어의 정보에 따른 토큰의 구분, 등록 및 고유 번호의 부여를 처리하는 동작이 그림 11 (b)의 스캐너 동작에 나타나 있다. 유한 오토마타가 추출한 abstract/character/string의 리터럴 토큰은 심볼 테이블에 등록되고 토큰의 고유 번호를 받는다. 식별자의 이름 토큰은 파서에서 보낸 선언/참조, 다중 정의 가능 여부 및 객체 정보 등을 이용하여 표 3의 어의 토큰으로 분류되며, 각 토큰에 대한 수행 함수에서는 심볼 테이블 및 트리에 등록하거나 저장된 심볼 정보를 검색해서 토큰의 고유 번호와 함께 파서로 전달된다. 또한 수행 함수는 추출된 예약어 및 구분자 토큰에 고유 번호를 부여해서 파서로 보낸다.

스캐너에서 추출된 토큰을 기본 단위로 구문 생성을 수행하는 파서는 그림 12의 구성 및 동작을 가진다. VHDL 파서는 기본적으로 푸쉬다운 오토마타의 구문 생성기 및 파싱 스택, 중간 수행 함수와 수행 함수로 구성된다. Bison이 VHDL 구문의 CFG 표현으로부터 생성한 푸쉬다운 오토마타는 스캐너가 전달한 토큰을 확인하고 다음 구문 상태를 결정한 후에 shift 및 reduce의 구문 생성을 수행한다. 해당 토큰의 확인은 파싱 상태표를 참조해서 VHDL 구문에서 허용되는 토큰인지를 검사하며, 또한 확인된 토큰에 따라서 구문 생성의 다음 상태가 결정된다. 이 과정에서 예상된 토큰을 스캐너가 보내지 않으면 파싱 오류가 발생되는데, 토큰에 관한 구문 및 객체 정보가 자동 검사되기 때문에 부분적인 어의 검사가 동시에 이루어진다. 결정된 다음 구문 상태로 이동하는 shift/reduce 과정에서 중간 수행 함수/수행 함수가 각각 호출되어 다음 토큰의 어의 정보를 추출하고 분석 데이터의 심볼 테이블에서 선언 영역을 이동시키며, 환원된 논터미날을 분석 데이터에 저장하고 관련된 어의를 검사하는 등의 처리를 수행한다. 실제로 구문 생성의 shift에 대해서 실행되는 중간 수행(middle action) 함수는 다음의 두 가지 기능을 담당한다.

- 어의 토큰 처리
- 심볼 트리에서 하위 선언 영역으로 이동

또한 구문 생성에서 논터미날이 환원(reduce)되면 수행(action) 함수를 통해서 다음과 같은 처리가 이루어진다.

- 논터미날 구조체 생성 및 AST의 상향식 구성
- 분석 데이터의 터미날 및 논터미날 정보 관리
- 논터미날에 관한 어의 검사
- 분석 데이터의 심볼 트리에서 상위 선언 영역으로 복원
- VHDL-to-C 사상을 위한 기타 전처리 과정

Ⅳ. 실험 및 결과

본 연구에서는 VHDL-to-C 사상을 위한 VHDL 컴파일러 전반부를 설계 및 구현하였다. VHDL'87과 VHDL'93 LRM에 대해서 개발된 컴파일러 전반부는 VHDL 설계 프로그램의 어휘 및 구문 문법을 분석하고, 정적 어의 문법을 검사하며, 컴파일러 후반부의 VHDL-to-C 사상을 위한 전처리 과정을 수행한다. 설계 및 구현된 VHDL 컴파일러 전반부의 연구 결과를 평가하기 위해서 VHDL LRM에 정의된 어휘, 구문 및 어의 문법의 분석 범위를 실험하였으며, 본 연구에서 설계 및 구현한 계층적 심볼 트리 구조, 어의 토큰 및 AST 어의 전파 등의 성능 개선 효과를 알아보았다.

VHDL 컴파일러 전반부의 VHDL 설계 프로그램에 대한 분석 범위를 평가하기 위해서 VHDL Validation Suite[13]를 이용하였다. VHDL 표준화 작업이 IEEE std.1076-1987과 IEEE/ANSI std.1076-1993으로 진행됨에 따라서 Validation Suite를 개발하기 위한 노력이 진행되고 있으며, 현재 VHDL Technology Group에서 Suite의 관리 및 개선을 추진하고 있다. 본 연구에서 설계 및 구현된 VHDL 컴파일러 전반부가 VHDL LRM에 정의된 어휘, 구문 및 어의 문법을 어느 정도 분석할 수 있는가를 Validation Suite를 통해서 평가한 결과가 표 5에 요약되어 있다.

- VHDL'87 Validation Suite에 대해서 컴파일러 전반부는 3225개 테스트 프로그램에서 2764개를 분석 및 검사하여 85.7%의 coverage를 처리하고

표 5. VHDL 컴파일러 전반부 실험 결과
Table. 5. Experimental results of VHDL compiler front-end

		VHDL '87			VHDL '93			compiling errors	
check		suites	success	failure	suites	success	failure	'87	'93
Lexical	no error	59	59	0	64	64	0	0 /290	0 /290
	compiler error	231	231	0	226	226	0		
	runtime error								
Syntactic	no error	183	183	0	189	189	0	0 /593	0 /593
	compiler error	410	410	0	404	404	0		
	runtime error								
Semantic	no error	1437	1345	92	1470	1379	91	461 /2342	443 /2342
	compiler error	882	532	350	849	516	333		
	runtime error	23	4	19	23	4	19		
total suites		3225	2764	461	3225	2782	443	461 /3225	443 /3225

표 6. VHDL 분석 데이터 모델의 성능 개선
Table. 6. Performance improvement of analyzed data model in VHDL compiler front-end

VHDL 테스트 프로그램	본 연구의 분석 데이터 모델	기본적인 분석 데이터 모델	성능 개선 효과
hierarchy.vhd	18 msec	25 msec	계층적 선언 영역의 표현
visibility.vhd	28 msec	32 msec	심볼 트리를 이용한 가시성 확인
concurrent.vhd	13 msec	16 msec	병행 구문의 심볼 관리
sym_prop.vhd	8 msec	12 msec	심볼 전파
type_prop.vhd	8 msec	13 msec	타입 전파

있다. 어휘 및 구문 문법의 테스트 프로그램은 100% 분석해서 VHDL 설계 프로그램의 오류를 확인시킬 수 있으며, 어의 문법에서는 80%의 검사 능력을 갖고 있다.

- VHDL 컴파일러 전반부는 VHDL'93 Validation Suite에 대해서 86.5%의 coverage를 가진다. 어휘 및 구문 문법의 테스트 프로그램은 100% 분석 및 검사가 이루어지며, 어의 문법에 대해서는 81.1%의 coverage를 보이고 있다.

컴파일러 전반부에서는 VHDL의 계층적 선언 영역 및 가시성, 다중 정적 스택 구조를 효과적으로 표현하기 위해서 심볼 트리 구조를 구현하였으며, 분석 과정에서 데이터 검색의 오버헤드를 줄이기 위해서 구문 및 객체 정보를 이용해 어의 토큰을 터미날 심볼로써 등록하고 심볼 및 타입 등의 어의 정보를 AST에서 전파시키는 분석 데이터 모델을 구성하였다. 이 같은 분석 데이터의 설계 및 구현 방법이 컴파일러 전반부의 분석 과정에서 어느 정보의 성능 개선을 가져오는가를 확인하기 위해서 VHDL의 계층성(hierarchy), 가시성(visibility), 병행성(concurrent), 심볼 및 타입 전파(symbol & type-propagation) 등을 나타내는 테스트 프로그램을 설계하고 VHDL 컴파일러의 성능을 실험하였다. 표 6은 본 연구에서 개선된 분석 데이터 모델을 이용한 경우와 기본적인 분석 데이터 모델을 이용한 경우를 비교해서 VHDL 컴파일러 전반부의 성능을 나타내고 있다. 실험 결과는 대

략 25% 정도의 성능 개선 효과를 보이고 있다.

이와 같은 성능 개선을 가져오는 분석 데이터 모델의 주요 특징은 1)심볼 트리, 2)어의 토큰 및 어의 전파의 두 가지로 요약된다. 1)심볼 트리는 VHDL 선언 영역 및 가시성의 처리 과정을 개선시킬 것으로 예상되며, 2)어의 토큰 및 어의 전파는 VHDL 어의 검사 과정의 오버헤드를 감소시킬 수 있을 것으로 예측된다. 두 가지 방법의 성능 개선 효과를 각각 확인하기 위해서 VHDL 테스트 프로그램(semantics.vhd, symbol.vhd)을 통해서 실험하였다. 테스트 프로그램 semantics.vhd는 30개의 어의 검사 항목을 포함하도록 설계되었으며, symbol.vhd는 VHDL 프로그램에서 많은 심볼이 계층적으로 중첩 및 확장되어 선언되고 참조되는 과정을 포함하도록 구성되었다. 표 7은 VHDL 컴파일러 전반부에서 개선된 분석 데이터 모델과 기본적인 분석 데이터 모델을 이용한 경우로 나누어서 semantics.vhd 및 symbol.vhd의 테스트 프로그램에 대해서 성능을 비교해서 나타내고 있다. 분석 데이터 모델의 두 가지 개선 방법을 통해서 VHDL 컴파일러 전반부는 각각 30% 이상의 성능 향상 효과를 얻을 수 있다.

표 7. VHDL 심볼 트리, 어의 토큰 및 전파의 성능 개선 효과
Table 7. Performance improvement of symbol tree, semantic token and propagation in VHDL compiler front-end

테스트 프로그램	개선된 분석 데이터 모델	기본적인 분석 데이터 모델	성능 개선 효과
semantics.vhd	326 msec	528 msec	어의 토큰 처리
symbol.vhd	194 msec	285 msec	심볼 트리 검색

V. 결 론

본 연구실에서는 VHDL 설계 모델을 C 실행 모델로 변환시키는 VHDL 컴파일러를 개발하고 있다. 본 논문은 VHDL 프로그램을 분석하고 VHDL-to-C 사상의 전처리 과정을 수행하는 컴파일러 전반부의 설계 및 구현에 대해서 기술하였다.

본 연구에서는 VHDL의 계층적 구조체, 선언 영역 및 가시성, 다중 정의와 동형 이의어, 병행적 다중 스택 구조 등을 표현하기 위해서 분석 터미널 데이터에 심볼 트리 구조를 구성하였으며, VHDL 고유의 객체, 타입 및 서브타입, 속성과 연산자를 나타내기 위한 구조체 및 처리 함수를 설계하였고, VHDL의 병행문/순차문, 행위/구조 기술, 동기 메커니즘 등을 분석하여 VHDL-to-C 사상에 필요한 정보를 분석 데이터에 구축하였다. 또한 컴파일러 전반부의 어의 검사와 후반부의 VHDL-to-C 사상 과정을 효과적으로 수행하기 위해서 객체 및 구문의 어의 정보를 이용한 어의 토큰 심볼을 터미널 정보로써 등록시키고 논터미널 데이터간의 어의 전파 기능을 AST 데이터에 포함시켜서 분석 데이터의 어의 정보를 저장 및 검색하기 위한 오버헤드를 감소시켰다. 또한 실제로 구현된 VHDL 컴파일러 전반부의 분석 데이터 모델은 컴파일러 후반부의 VHDL-to-C 사상을 통해서 효용성 및 적합성이 확인되었다.

참 고 문 헌

1. *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076-1987, IEEE, Mar. 1988.
2. *IEEE Standard VHDL Language Reference Manual*, ANSI/IEEE Std. 1076-1993, IEEE, Jun. 1994.
3. Michael Carroll, "VHDL-panacea or hype?," *IEEE Spectrum*, pp. 34-37, Jun. 1993.
4. Jean-Michel Berg , Alain Fonkoua, Serge Maginot and Jacques Rouillard, *VHDL Designer's Reference*, Kluwer Academic Publishers, 1992.
5. *Leapfrog VHDL Simulator Reference Manual 1.0*, Cadence, Jun. 1993.
6. *Vantage SpreadSheet User's Guide Volume 1*, Vantage Analysis System Inc., Dec. 1990.
7. *V-System/Windows User's Manual*, Model Technology, Mar. 1993.
8. 최기영, "IVDT : VHDL Tool 개발 환경," *Proceedings of the 1st Korea VHDL User's Group Workshop*, pp. 1-30, 1993년 5월.
9. 이영희, 황선영, "VHDL 시뮬레이터의 설계와 구현," 대한 전자공학회지, Vol. 22 No. 8, pp. 90-924, Aug. 1995.
10. Vern Paxson, *Flex : A Fast Scanner Generator Ver-*

sion 2.5, GNU, Mar. 1995.

11. Charles Donnelly, Richard Stallman, *Bison: The YACC-compatible Parser Generator Version 1.25*, GNU, Nov. 1995.
12. 공진흥, 박성주, 박찬원, "어의 문법의 테이블 기술을 이용한 VHDL 정적 어의 검사기 구현," 전자공학회 논문지, Vol. 33, No. 9-A, 1996.
13. *VHDL Validation Testsuite Users Manual*, VHDL Technology Group, 1995.

공 진 흥(Jin-Hyeung Kong) 정회원

1976년 3월~1980년 2월 : 서울대학교 전자공학 공학사

1980년 3월~1982년 2월 : 한국과학기술원 통신공학 공학석사

1986년 9월~1989년 12월 : 텍사스 주립대학교 컴퓨터공학 공학박사

1979년 11월~1986년 8월 : 삼성전자(반도체 연구소) 과장

1989년 9월~현재 : 광운대학교 컴퓨터공학과 부교수

고 형 일(Hyung-Il Goh) 정회원

1991년 3월 : 광운대학교 전자계산기공학과 입학

1995년 2월 : 동대학교 컴퓨터공학과 졸업

1995년 3월 : 동대학교 대학원 전자계산기공학과 석사과정 입학

1997년 2월 : 동대학교 대학원 동학과 졸업

1997년 3월 : 동대학교 대학원 컴퓨터공학과 박사과정 입학