CHAPTER _3

문법 요소

3장에서는 델파이 소스를 구성하는 문법 요소들에 대해 설명합니다. 식별자, 예약어, 지시어 등의 요소들과, 표현식, 문장, 블럭 등에 대해 살펴봅니다.

■델파이 문자 셋 ■기본적인 문법 ■주석문과 컴파일러 지시어
■표현식 ■함수 호출 ■집합 생성자 ■인덱스
■타입 캐스트 ■선언과 문장 ■블럭과 유효 범위

델파이 문자 셋

델파이 언어는 유니코드(Unicode) 문자셋(character set)을 사용하며, 여기에는 유니코드 문자들과 영숫자 및 밑줄이 포함됩니다. 대/소문자는 구분하지 않습니다. 스페이스 문자와 ASCII 제어 문자들(리턴 혹은 개행 문자인 ASCII 13을 포함, ASCII 0부터 31까지)은 공백(blank)이라고 부릅니다.

RAD 스튜디오 컴파일러는 파일에 BOM(byte order mark) 헤더가 있을 경우 UCS-2 혹은 UCS-4 인코딩된 파일을 허용합니다. 하지만 UTF-8 이외의 포맷을 사용할 경우 컴파일 속도가 느려질 수 있습니다. UCS-4로 인코딩된 소스 파일의 모든 문자들은 UCS-2에서 서로게이트 페어(surrogate pair) 없이 표현 가능해야 합니다. 서로게이트 페어를 가진 UCS-2 인코딩(GB18030 포함)은 codepage 컴파일러 옵션이 지정된 경우에만 허용됩니다.

기본 문법 요소들은 토큰이라고 불리며, 조합되어 표현식, 선언 및 문장을 구성합니다. 문장은 프로그램 내에서 실행될 수 있는 알고리즘 동작을 기술합니다. 표현식은 문장에서 나타나는 문법 단위로서 값을 나타냅니다. 선언은 표현식과 문장에서 사용할 수 있는 식별자(함수나 변수의 이름 등)를정의하고, 적절한 경우 식별자를 위한 메모리를 할당합니다.

기본적인 문법

가장 단순하게 보자면, 프로그램은 구분자(separator)로 단락지어지는 연속된 토큰들입니다. 토큰은 프로그램에서 의미를 가진 텍스트의 최소 단위입니다. 구분자는 공백이거나 주석(comment)입니다. 엄밀히 말하면, 반드시 두 토큰 사이에 구분자를 두어야 하는 것은 아닙니다. 예를 들어 다음의 코드는,

```
Size:=20;Price:=10;
```

완벽하게 문법에 맞습니다. 하지만, 관례에 따르고 가독성을 높이기 위해 다음과 같이 코드를 작성하시기 바랍니다.

```
Size := 20;
Price := 10;
```

토큰은 특수 기호(symbol), 식별자, 예약어, 지시어, 숫자, 레이블, 문자열로 분류됩니다. 구분자는 토큰이 문자열인 경우에만 토큰의 일부가 될 수 있습니다. 인접한 식별자, 예약어, 숫자, 레이블 사이에는 하나 이상의 구분자가 있어야 합니다.

특수 기호

특수 기호는 문자나 숫자, 혹은 이들의 쌍이 아닌 문자들로서 고정된 의미를 가진 문자들입니다. 다음 단일 문자들은 특수 기호입니다.

$ & ‘ ( ) * + , − . / : ; 〈 = 〉 @ [ ] ^ { }

다음 문자 쌍들도 역시 특수 기호입니다.

(* (. *) .) .. // := 〈= 〉= 〈〉

다음의 표는 같은 의미를 가진 기호들을 나열합니다.

Note

%, ?, \, !, " (쌍따옴표), _ (밑줄), | (파이프), ~ (물결)은 특수 기호가 아닙니다.

표 3.1 동의 기호들

특수 기호	같은 의미의 기호
[	(.
]	.)
{	(*
}	*)

왼쪽 대괄호 [는 왼쪽 괄호와 마침표의 쌍 (.과 같은 의미입니다.
오른쪽 대괄호]는 마침표와 오른쪽 괄호의 쌍.)과 같은 의미입니다.
왼쪽 중괄호 {는 왼쪽 괄호와 별표 (*와 같은 의미입니다.
오른쪽 중괄호 }는 별표와 오른쪽 괄호 *)와 같은 의미입니다.

식별자

식별자(identifier)는 상수, 변수, 필드, 타입, 속성, 프로시저, 함수, 프로그램, 유닛, 라이브러리, 패키지를 표시합니다. 식별자의 길이는 아무런 제한이 없지만 처음 255개의 문자만 의미를 가집니다. 식별자는 문자나 밑줄(_)로 시작해야 하고 공백을 포함할 수 없습니다. 첫 번째 문자 뒤에는 문자, 숫자 및 밑줄이 올 수 있습니다. 예약어는 식별자로 사용될 수 없습니다.
델파이 언어에서는 대소문자를 구분하지 않으므로 CalculateValue와 같은 식별자는 다음 중 어떤 방식으로도 쓸 수 있습니다.

```
CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE
```

유닛 이름은 파일 이름과 같기 때문에 대소문자가 일치하지 않으면 경우에 따라 컴파일에 문제를 일으킬 수도 있습니다. 더 자세히 알아보려면, "유닛 참조와uses 절"을 참고하십시오.

유닛으로 한정된 식별자

둘 이상의 유닛에서 선언된 식별자를 사용할 때, 경우에 따라 식별자를 한정(qualify)해야 할 수도 있습니다. 한정된 식별자에 대한 문법은 다음과 같습니다.

```
identifier1.identifier2
```

여기서, *identifier1*은 *identifier2*를 한정합니다. 예를 들어 두 유닛이 각각 CurrentValue라는 변수를 선언한 경우, 사용자는 다음과 같이 Unit2의 CurrentValue에 액세스하려 한다고 지정할 수 있습니다.

```
Unit2.CurrentValue
```

한정자는 반복해서 사용할 수 있습니다. 예를 들면, 다음과 같습니다.

```
Form1.Button1.Click
```

이 문장은 Form1의 Button1에 있는 Click 메소드를 호출합니다.
식별자를 한정하지 않으면 3장의 "블럭과 유효 범위"절에서 설명하는 범위 규칙에 따라 식별자를 해석합니다.

예약어

다음 예약어(reserved word)는 재정의하거나 식별자로 사용할 수 없습니다.

표 3.2 예약어

add	else	initialization	program	then
and	end	inline	property	threadvar
array	except	interface	raise	to
as	exports	is	record	try
asm	file	label	remove	type
begin	final	library	repeat	unit
case	finalization	mod	resourcestring	unsafe
class	finally	nil	sealed	until
const	for	not	set	uses
constructor	function	object	shl	var
destructor	goto	of	shr	while
dispinterface	if	or	static	with
div	implementation	out	strict private	xor
do	in	packed	strict	protected
downto	inherited	procedure	string	

표 3.2에 있는 예약어 외에, private, protected, public, published 및 automated는 객체 타입 선언 내에서는 예약어로 사용되지만, 그이외의 경우에는 지시어로 간주됩니다. at과 on도 특별한 의미를 가지고 있으므로 예약어로 취급되어야 합니다.

지시어

지시어(directive)는 소스 코드 내의 특정 위치에서 의미를 가진 단어입니다. 지시어는 델파이 언어에서 특별한 의미를 갖지만 예약어와 달리 사용자가 정의하는 식별자가 나타날 수 없는 문맥에만 나타납니다. 따라서, 권장할 만한 방법은 아니지만 지시어처럼 보이는 식별자를 사용자가 정의할 수도 있습니다.

표 3.3 지시어

absolute	export	name	protected	stdcall
abstract	external	near	public	stored
assembler	far	nodefault	published	varargs
automated	forward	overload	read	virtual
cdecl	implements	override	readonly	write
contains	index	package	register	writeonly
default	inline	pascal	reintroduce	
deprecated	library	platform	requires	
dispid	local	pointermath	resident	
dynamic	message	private	safecall	

숫자

정수 상수와 실수 상수는 쉼표나 공백이 없는 일련의 숫자들의 10진수 표기이며, 부호를 표시하기 위해 + 혹은 - 연산자를 앞에 붙일 수 있습니다. 값은 기본적으로 양수이며 (따라서 67258은 +67258과 동일), 선언된 타입의 범위를 넘지 않아야 합니다.

소수점이나 지수가 있는 숫자는 실수이며, 실수가 아닌 숫자는 정수입니다. E 또는 e 문자가 실수에서 사용되면, "10의 거듭제곱"을 의미합니다. 예를 들어, 7E2는 7×10^2를 의미하며, 12.25e+6과 12.25e6는 모두 12.25×10^6을 의미합니다.

선두에 달러($) 기호가 쓰이면 16진수를 나타냅니다. (예를 들면 $8F) 앞에 - 단항 연산자가 없는 16진수는 양수로 간주됩니다.

16진수의 부호는 바이너리 데이터의 가장 왼쪽에 있는 비트(최상위 비트)에 의해서 결정됩니다. 대입 과정에서 16진수 값이 대입되는 타입의 범위를 벗어나면 에러가 발생하지만, Integer(32비트 정수)의 경우 예외적으로 경고(warning)가 발생합니다. 이런 경우에 Integer의 양수 한계를 벗어나는 값은 2의 보수 방식으로 음수로 취급됩니다.

실수 타입과 정수 타입에 대한 자세한 내용은 4장 "데이터 타입, 변수 및 상수"를 참조하십시오. 숫자의 데이터 타입에 대한 내용은 4장의 "순수 상수" 절을 참조하십시오.

레이블

레이블은 표준 델파이 언어 식별자이지만 예외적으로 레입은 숫자로 시작할 수 있습니다. 숫자 레이블은 10자까지만 가질 수 있으므로 0에서 9999999999까지의 숫자가 됩니다.
레이블은 goto 문에서 사용됩니다. goto 문과 레이블에 대한 자세한 내용은 3장의 "goto 문"을 참조하십시오.

문자열

문자열은 문자열 리터럴 또는 문자열 상수라고도 불리며, 인용 문자열, 제어 문자열, 또는 인용 문자열과 제어 문자열의 조합으로 구성됩니다. 구분자는 인용 문자열 내에서만 사용할 수 있습니다.
인용 문자열은 확장된 ASCII 문자셋에서 최대 255개의 문자들의 연속으로, 한 줄로 쓰며 작은 따옴표(')로 묶입니다. 작은 따옴표들('') 사이에 아무것도 없는 인용 문자열은 Null 문자열입니다. 인용 문자열 내에서 연속적인 두 개의 작은 따옴표('')는 하나의 작은 따옴표(')를 의미합니다. 예를 들면 다음과 같습니다.

```
'CODEGEAR'      { CODEGEAR }
'You''ll see'   { You'll see }
''''             { ' }
''               { Null 문자열 }
' '              { 스페이스 문자 }
```

제어 문자열은 하나 이상의 연속된 제어 문자들입니다. 각각의 제어 문자는 #기호에 이어 0에서 255까지의 부호 없는 정수 상수(10진수 혹은 16진수)로 구성되고, 해당 ASCII 문자를 나타냅니다. 제어 문자열의 예제는 다음과 같습니다.

```
#89#111#117
```

이 문장은 다음 인용 문자열과 동일합니다.

```
'You'
```

제어 문자열과 인용 문자열을 조합하면 더 큰 문자열을 만들 수 있습니다. 예를 들어 다음 문자열은,

```
'Line 1'#13#10'Line 2'
```

'Line 1'과 'Line 2' 사이에 캐리지 리턴과 라인 피드가 삽입됩니다. 그러나 한 쌍의 연속적인 작은 따옴표('')는 단일 문자로 해석되기 때문에 이러한 방법으로 두 개의 인용 문자열을 연결할 수는 없습니다. 인용 문자열을 연결하려면 3장의 "문자열 연산자"에서 설명된 + 연산자를 사용하거나, 하나의 인용 문자열로 만드십시오.
문자열의 길이는 문자열에 있는 문자의 갯수입니다. 길이에 상관 없이 문자열은 모든 문자열 타입과 PChar 타입과 호환됩니다. 길이가 1인 문자열은 모든 문자 타입과 호환되고, 확장 문법이 활성화(컴파일러 지시자 {$X+})되면 0보다 큰 n 길이의 문자열은 첨자가 0부터 시작하는 n 문자의 배열 및 압축 배열과 호환됩니다. 문자열 타입에 대한 자세한 내용은 4장 "데이터 타입, 변수 및 상수"를 참조하십시오.

주석문과 컴파일러 지시어

주석문(comment)은 컴파일러에서 무시됩니다. 인접한 토큰들 사이의 구분자로서 혹은 컴파일러 지시어로서의 기능을 하는 경우는 예외입니다.
주석문을 만드는 방법은 다음과 같이 몇가지가 있습니다.

```
{  왼쪽 중괄호와 오른쪽 중괄호 사이의 텍스트는 주석문이 됩니다. }
(*  왼쪽 괄호와 별표, 그리고 별표와 오른쪽 괄호 사이의 내용도 주석문이 됩니다.  *)
//  슬래시 두개와 줄끝 문자(개행) 사이의 내용은 주석문이 됩니다.
```

동일한 주석문 기호를 중복해서 주석문을 만들 수는 없습니다. { { } }는 안되지만 (* { } *)는 됩니다. 이것은 주석문을 포함한 코드 영역을 주석 처리하는 데에 유용합니다.
열린 { 또는 (* 바로 뒤에 달러 기호($)가 있는 주석문은 컴파일러 지시어(compiler directive)입니다. 예를 들면 다음과 같습니다.

```
{$WARNINGS OFF}
```

이 문법은 컴파일러가 경고 메시지를 내지 않도록 지정합니다.

표현식

표현식(expression)은 값을 리턴하는 코드 구조입니다. 오른쪽의 표는 델파이 표현식의 예들을 보여줍니다.

가장 간단한 표현식은 변수와 상수 자체입니다. 연산자, 함수 호출, 집합 생성자, 인덱스 및 타입 캐스트를 사용하면 간단한 표현식으로부터 복잡한 표현식을 만들 수 있습니다.

X	변수
@X	변수의 주소
15	정수 상수
InterestRate	변수
Calc(X,Y)	함수 호출
X * Y	X와 Y의 곱
Z / (1 - Z)	Z를 (1 - Z)로 나누기
X = 1.5	부울
C in Range1	부울
not Done	부울의 부정
['a','b','c']	집합
Char(48)	값 타입 캐스트

연산자

연산자(operator)는 델파이 언어에 기본적으로 내장된 함수처럼 동작합니다. 예를 들어, 표현식 (X + Y)는 변수 X와 Y(피연산자라고 부름)와 + 연산자를 사용하여 만들어집니다. 즉, X와 Y가 정수나 실수를 나타내는 경우 (X + Y)는 이 둘의 합을 돌려줍니다. 연산자에는 @, not, ^, *, /, div, mod, and, shl, shr, s, +, -, or, xor, =, 〉, 〈, 〈〉, 〈=, =, in, is가 있습니다.

연산자 @, not 및 ^는 피연산자가 하나인 단항 연산자(unary operator)입니다.다른 모든 연산자들은 피연산자가 두 개인 이항 연산자(binary operator)인데, +와 ?는 단항 연산자와 이항 연산자 모두로 사용할 수 있습니다. 피연산자 뒤에 오는 ^(예를 들어 P^)를 제외한 모든 단항 연산자는 -B 같이 피연산자의 앞에 옵니다. 이항 연산자는 피연산자들 사이에 위치합니다(예를 들어 A = 7).

일부 연산자는 전달된 데이터의 타입에 따라 다르게 동작합니다. 예를 들어, not은 정수 피연산자에 대해서는 비트(bitwise) 부정 연산을 수행하고, 부울 피연산자에 대해서는 논리 부정 연산을 수행합니다. 이런 연산자들은 아래와 같이 다양한 범주에서 나타납니다.

^, is 및 in을 제외한 모든 연산자는 Variant 타입의 피연산자를 사용할 수 있습니다. 자세한 내용은 4장의 "Variant 타입"을 참조하십시오.

다음에 이어지는 절들을 살펴보려면 델파이 데이터 타입에 대해 약간 익숙할 필요가 있습니다. 데이터 타입에 대한 내용은 4장 "데이터 타입, 변수 및 상수"를 참조하십시오.

복잡한 표현식의 연산자 우선 순위에 대한 내용은 4- 2 절의 "연산자 우선 순위"를 참조하십시오.

산술 연산자

산술 연산자는 실수 또는 정수를 다루며, 여기에는 +, -, *, /, div, mod가 있습니다.

표 3.4 이항 산술 연산자

연산자	연산	연산자 타입	결과 타입	예제
+	더하기	정수, 실수	정수, 실수	X + Y
–	빼기	정수, 실수	정수, 실수	Result – 1
*	곱하기	정수, 실수	정수, 실수	P * InterestRate
/	실수 나누기	정수, 실수	실수	X / 2
div	정수 나누기	정수	정수	Total div UnitSize
mod	나머지	정수	정수	Y mod 6

표 3.5 단항 산술 연산자

연산자	연산	연산자 타입	결과 타입	예제
+	부호 동일	정수, 실수	정수, 실수	+7
–	부호 부정	정수, 실수	정수, 실수	–X

다음 규칙은 산술 연산자에 대해 적용됩니다.

- x/y의 값은 x와 y의 타입에 관계 없이 Extended 타입입니다. 다른 산술 연산자에서는 피연산자 중 하나가 실수이면 결과는 Extended 타입입니다. 피연산자 중 하나가 Int64 타입이면, 결과는 Int64 타입입니다. 그외에는 정수 타입입니다. 피연산자의 타입이 정수 타입의 부분 범위인 경우 정수 타입처럼 처리됩니다.
- x div y 값은 x/y를 가장 가까운 정수로 내림한 값, 즉 몫입니다.
- mod 연산자는 피연산자 나눗셈의 나머지를 리턴합니다. 다시 말하면 다음과 같습니다.
  x mod y = x – (x div y) * y
- x/y, x div y 또는 x mod y 같은 표현식에서 y가 0이면 런타임 에러가 발생합니다.

부울 연산자

부울 연산자(boolean operator)인 not, and, or, xor는 부울 타입의 피연산자를 받고 부울 타입의 값을 돌려줍니다.

표 3.6 부울 연산자

연산자	연산	연산자 타입	결과 타입	예제
not	부정	부울	부울	not (C in MySet)
and	논리곱	부울	부울	Done and (Total>0)
or	논리합	부울	부울	A or B
xor	배타적 or	부울	부울	A xor B

이러한 연산들은 부울 로직의 표준 규칙에 따릅니다. 예를 들어, x와 y 모두가 True인 경우에만 x and y 표현식이 True입니다.

■완전 부울 계산과 단축 부울 계산

컴파일러는 and 및 or 연산자에 대한 계산에 두가지 모드가 있는데, 완전 계산(complete evaluation)과 단축 계산(short-circuit evaluation)입니다. 완전 계산은 전체 표현식의 결과가 이미 정해졌더라도 각각의 논리곱이나 논리합이 계산된다는 것을 의미합니다. 부분 계산은 왼쪽에서부터 오른쪽으로 계산해나가다가 전체 표현식의 결과가 정해지는 즉시 중단됩니다. 예를 들어, A가 False인 상태에서 단축 모드로 A and B 표현식이 계산되는 경우, 컴파일러는 B를 계산하지 않습니다. 컴파일러는 A를 계산한 직후에 전체 표현식이 False라는 것을 알기 때문입니다.

단축 계산이 실행 시간을 최소화하고 또 대부분의 경우 코드 크기를 최소화하기 때문에 일반적으로 더 많이 사용됩니다. 한 피연산자가 함수이고 프로그램의 실행을 바꿀 수 있는 부작용이 있는 경우에는 완전 계산이 편리합니다.

또한 단축 계산은 경우에 따라 잘못된 런타임 연산을 피할 수 있게 해 줍니다. 예를 들어, 다음 코드는 문자열 S에서 첫 번째 쉼표를 만날 때까지 반복됩니다.

```
while (I <= Length(S)) and (S[I] <> ',') do
begin
  ...
  Inc(I);
end;
```

만약 S에 쉼표가 없으면, 마지막 반복에서 I 값이 증가하여 S의 길이보다 큰 값을 가지게 됩니다. while 조건문이 다음 조건을 테스트하려 하면 완전 계산은 S[I]를 읽으려고 시도하게 되어 런타임 에러를 일으킬 수 있습니다. 반대로, 단축 계산에서는 while의 첫번째 조건이 실패하므로 두 번째 조건(S[I] 〈〉 ',')는 계산되지 않습니다.

참고

피연산자에 Vairant 타입이 관련된 경우에는 컴파일러는 {$B-} 모드에서도 항상 완전 계산을 수행합니다.

$B 컴파일러 지시어를 사용하면 계산 모드를 지정할 수 있습니다. 기본 상태는 {$B-}로서 단축 계산 모드입니다. 코드 일부에서 완전 계산 모드로 지정하려면 코드에 {$B+} 지시어를 추가하십시오. 프로젝트 전체를 완전 계산 모드로 지정하려면, Compiler Option 다이얼로그에서 Compiler Boolean Evaluation을 선택하면 됩니다.

논리(비트 단위) 연산자

다음의 논리 연산자들은 정수 피연산자를 비트 단위(bitwise)로 조작합니다. 예를 들어, X에 저장된 값이 001101이고 Y에 저장된 값이 100001일 경우 다음의 문장은,

```
Z := X or Y;
```

Z에 값 101101을 대입합니다.

표 3.7 논리 (비트 단위) 연산자

연산자	연산	연산자 타입	결과 타입	예제
not	비트 부정	정수	정수	**not** X
and	비트 and	정수	정수	X **and** Y
or	비트 or	정수	정수	X **or** Y
xor	비트 xor	정수	정수	X **xor** Y
shl	왼쪽 비트 시프트	정수	정수	X **shl** 2
shr	오른쪽 비트 시프트	정수	정수	Y **shr** I

비트 단위 연산자에는 다음과 같은 규칙이 적용됩니다.

- not 연산의 결과는 피연산자와 같은 타입이 됩니다.
- and, or 또는 xor 연산의 피연산자가 둘 다 정수인 경우 그 결과는 두 타입의 모든 가능한 값을 포함하는 가장 작은 범위의 이미 정의된 정수 타입을 가집니다.
- x shl y와 x shr y 연산은 x의값을 왼쪽 혹은 오른쪽으로 y 비트만큼 시프트하는데, 결과 값은 x를 2 로 곱하거나 나눈 것과 동일합니다. 그 결과는 x와 같은 타입을 가집니다. 예를 들어, N이 01101 (십진수 13) 값을 저장하고 있는 경우, N shl 1은 11010 (십진수 26)을 리턴합니다.

문자열 연산자

관계 연산자 =, 〈〉, 〈, 〉, 〈= 및 〉=는 모두 문자열 피연산자를 가질 수 있습니다(3장의 "관

계 연산자" 참조). + 연산자는 두 문자열을 연결합니다.

표 3.8 문자열 연산자

연산자	연산	연산자 타입	결과 타입	예제
+	연결	문자열, packed 문자열, 문자	문자열	S + '.'

문자열 연결에는 다음과 같은 규칙이 적용됩니다.

- +의 피연산자는 문자열, packed 문자열(Char 타입의 압축 배열) 또는 문자일 수 있습니다. 그러나, 하나의 피연산자가 WideChar 타입인 경우, 다른 피연산자는 긴 문자열(AnsiString이나 WideString)이어야 합니다.
- + 연산의 결과는 모든 문자열 타입과 호환됩니다. 그러나, 피연산자가 모두 짧은 문자열 혹은 문자이고 합한 길이가 255보다 클 경우, 결과는 앞의 255개 문자까지 잘린 값입니다.

포인터 연산자

관계 연산자 〈, 〉, 〈= 및 〉=는 PChar와 PWideChar 타입의 피연산자를 가질 수 있습니다(3장의 "관계 연산자" 참조). 다음 연산자도 피연산자로서 포인터를 가질 수 있습니다. 포인터에 대한 자세한 내용은 4장의 "포인터와 포인터 타입(Pointer type)"을 참조하십시오.

표 3.9 문자-포인터 연산자

연산자	연산	연산자 타입	결과 타입	예제
+	포인터 더하기	문자 포인터, 정수	문자 포인터	P + I
-	포인터 빼기	문자 포인터, 정수	문자 포인터, 정수	P - Q
^	포인터 역참조	포인터	포인터의 기본 타입	P^
=	동등	포인터	부울	P = Q
〈〉	부등	포인터	부울	P 〈〉 Q

^ 연산자는 포인터를 역참조합니다. 모든 타입의 포인터가 ^ 연산자의 피연산자가 될 수 있는데, 일반 Pointer는 예외로서 역참조하기 전에 타입 캐스트되어야 합니다.

P == Q는 P와 Q가 같은 주소를 가리키는 경우에만 True입니다. 그렇지 않은 경우에 P 〈〉 Q가 True입니다.

+와 - 연산자를 사용하여 문자 포인터의 오프셋을 증가시키고 감소시킬 수 있습니다. 또한 -를 사용하여 두 문자 포인터의 오프셋 차이를 계산할 수 있습니다. 다음과 같은 규칙이 적용됩니다.

- I가 정수이고 P가 문자 포인터인 경우, P + I는 P의 주소에 I를 더합니다. 즉, P 주소에서 I 문자만큼 더한 주소의 포인터를 리턴합니다. (표현식 I + P는 P + I와 동일합니다.) P – I는 P 주소에서 I를 뺍니다. 즉, P 주소에서 I 문자만큼 뺀 주소의 포인터를 리턴합니다. 이것은 P가 PChar 포인터일 경우이고, P가 PWideChar 포인터일 경우에 P + I는 P에 SizeOf(WideChar)를 더합니다.
- P와 Q가 모두 문자 포인터인 경우 P – Q는 P 주소(상위 주소)와 Q 주소(하위 주소) 간의 차이를 계산합니다. 즉, P와 Q 사이의 문자 수를 나타내는 정수를 리턴합니다. P + Q는 허용되지 않습니다.

집합 연산자

다음 연산자는 피연산자로서 집합(set)을 가집니다.

표 3.10 집합 연산자

연산자	연산	연산자 타입	결과 타입	예제
+	합집합	set	set	Set1 + Set2
–	차집합	set	set	S – T
*	교집합	set	set	S * T
<=	일부 집합	set	부울	Q <= MySet
>=	포함 집합	set	부울	S1 >= S2
=	동등	set	부울	S2 = MySet
<>	부등	set	부울	MySet <> S1
in	속함	서수, set	부울	A in Set1

+, -, *에는 다음과 같은 규칙이 적용됩니다.

- 서수(ordinal) O가 X나 Y(또는 모두)에 속하면, O는 X + Y에 속합니다. O가 X에 속하고 Y에 속하지 않는 경우, O는 X – Y에 속합니다. O가 X와 Y에 모두 속하는 경우, O는 X * Y에 속합니다.
- +, –, * 연산의 결과는 set of A..B 타입인데 A는 결과 집합에서 가장 작은 순서값이고 B는 가장 큰 순서값입니다.

다음 규칙은 <=, >=, =, <> 및 in에 적용됩니다.

- X가 Y에 속한 경우에만 X <= Y는 True입니다. 즉, Z >= W는 W <= Z와 동일합니다. U와 V의 요소가 정확히 같은 경우에만 U = V는 True입니다. 그렇지 않은 경우 U <> V가 True입니다.
- 서수 O와 집합 S에서는 O가 S에 속하는 경우에만 O in S가 True입니다.

관계 연산자

관계 연산자는 두 개의 피연산자를 비교하는 데 사용됩니다. 연산자 =, <>, <= 및 >=는 집합에도 적용됩니다(3장의 "집합 연산자" 참조). 즉, = <>는 포인터에도 적용됩니다(3장의 "포인터 연산자" 참조).

표 3.11 관계 연산자

연산자	연산	연산자 타입	결과 타입	예제
=	동등	단순 타입, 클래스, 클래스 참조, 인터페이스, 문자열, packed 문자열	부울	I = Max
<>	부등	단순 타입, 클래스, 클래스 참조, 인터페이스, 문자열, packed 문자열	부울	X <> Y
<	작다	단순 타입, 문자열, packed 문자열, PChar	부울	X < Y
>	크다	단순 타입, 문자열, packed 문자열, PChar	부울	Len > 0
<=	작거나 같다	단순 타입, 문자열, packed 문자열, PChar	부울	Cnt <= I
>=	크거나 같다	단순 타입, 문자열, packed 문자열, PChar	부울	I >= 1

대부분의 단순 타입에서는 비교는 직관적입니다. 예를 들어, I와 J가 같은 값을 가지는 경우에만 I = J가 True이고, 그렇지 않은 경우 I <> J가 True입니다. 관계 연산자에는 다음과 같은 규칙이 적용됩니다.

- 피연산자들은 호환 가능한 타입이어야 하지만, 예외적으로 실수와 정수는 비교가 가능합니다.
- 문자열은 해당 문자열을 구성하는 문자들의 번호 값에 따라 비교됩니다. 문자 타입은 길이가 1인 문자열로 처리됩니다.
- 두 packed 문자열을 비교하려면 컴포넌트 수가 같아야 합니다. n개의 컴포넌트가 있는 packed 문자열을 문자열과 비교할 때 packed 문자열은 길이가 n인 문자열처럼 처리됩니다.
- 두 포인터가 같은 문자 배열 내를 가리키는 경우에만 연산자 <, >, <=, >=가 PChar 피연산자에 적용됩니다.
- 연산자 =와 <>는 클래스 타입과 클래스 참조 타입의 피연산자를 가질 수 있습니다. 클래스 타입의 피연산자에 대해 =와 <>는 포인터에 적용되는 다음 규칙에 따라 계산됩니다. C와 D가 같

은 인스턴스 객체를 가리키는 경우에만 C = D가 True입니다. 그렇지 않은 경우 C 〈〉 D가 True입니다. 클래스 참조 타입의 연산자로는, C와 D가 같은 클래스를 나타내는 경우에만 C = D는 True입니다. 그렇지 않은 경우 C 〈〉 D가 True입니다. 포인터에 대한 자세한 내용은 6장 "클래스와 객체"를 참조하십시오.

클래스 연산자

연산자 as와 is는 클래스와 인스턴스 객체를 피연산자로 가집니다. as는 인터페이스에 대해서도 동작합니다. 자세한 내용은 6장 "클래스와 객체" 및11장 "인터페이스"를 참조하십시오. 관계 연산자 =와 〈〉도 클래스에 대해 연산합니다. 바로 앞의 "관계 연산자" 부분을 참조하십시오.

@ 연산자

@ 연산자는 변수의 주소나 함수, 프로시저, 메소드의 주소를 리턴합니다. 즉, @는 피연산자에 대한 포인터를 알아냅니다. 포인터에 대한 자세한 내용은 4장의 "포인터와 포인터 타입"을 참조하십시오. 다음 규칙은 @에 적용됩니다.

- X가 변수인 경우 @X는 X의 주소를 리턴합니다. X가 프로시저 변수일 때에는 특별한 규칙이 적용됩니다. 4장의 "프로시저 타입" 절에서 "문장 및 표현식의 프로시저 타입"을 참조하십시오. 기본값 {$T-} 컴파일러 지시어가 적용된 상태에서 @X의 타입은 포인터입니다. {$T+} 상태에서 X의 타입이 T이면 @X는 ^T 타입을 가집니다. (이 차이는 대입 호환성과 관련하여 중요합니다. 4장의 "대입 호환성"을 참고하십시오.)
- F가 루틴(함수나 프로시저)인 경우, @F는 F의 엔트리 포인트 주소 값을 리턴합니다. @F의 타입은 항상 포인터입니다.
- @가 클래스에서 정의된 메소드에 적용될 때, 메소드 식별자는 클래스 이름이 지정되어야 합니다. 예를 들면, 다음과 같습니다. @TMyClass.DoSomething이 문장은 TMyClass의 DoSomething 메소드를 가리킵니다. 클래스와 메소드에 대한 자세한 내용은 6장 "클래스와 객체"를 참조하십시오.

연산자 우선 순위

복잡한 표현식에서는 연산자 우선 순위 규칙에 따라 연산이 수행되는 순서가 정해집니다.

표 3.12 연산자 우선 순위

연산자	우선 순위
@, not	첫번째 (가장 높음)
*, /, div, mod, and, shl, shr, as	두번째
+, −, or, xor	세번째
=, <>, <, >, <=, >=, in, is	네번째 (가장 낮음)

우선 순위가 높은 연산자는 우선 순위가 낮은 연산자보다 먼저 계산되고, 우선 순위가 같은 연산자는 왼쪽에서 오른쪽 순으로 계산됩니다. 따라서 표현식이 다음과 같은 경우,

```
X - Y * Z
```

Y와 Z를 곱한 후 그 결과에 X를 더합니다. 즉, *가 +보다 우선 순위가 높기 때문에 먼저 수행됩니다. 그러나 다음의 경우,

```
X - Y + Z
```

먼저 X에서 Y를 뺀 후 그 결과에 Z를 더합니다. 즉, -와 +는 우선 순위가 같기 때문에 왼쪽에 있는 연산이 먼저 수행됩니다.

괄호를 사용하여 이러한 우선 순위 규칙보다 우선하여 처리되도록 할 수 있습니다. 괄호 내에 있는 표현식이 먼저 계산되고, 계산된 결과는 하나의 피연산자로 간주됩니다. 예를 들면, 다음과 같습니다.

```
(X + Y) * Z
```

이 문장은 X와 Y의 합에 Z를 곱합니다.

언뜻 보면 괄호가 필요하지 않은 것처럼 보이는 곳에 괄호가 필요한 경우도 있습니다. 예를 들어, 다음의 표현식을 생각해 봅시다.

```
X = Y or X = Z
```

이 문법의 원래의 의도는 분명히 다음과 같습니다.

```
(X = Y) or (X = Z)
```

그러나 괄호가 없다면 컴파일러는 연산자 우선 순위에 따라 다음과 같이 읽습니다.

```
(X = (Y or X)) = Z
```

이 경우 Z가 부울 값이 아니라면 컴파일 에러가 발생합니다.
필요하지 않은 경우라도 괄호를 많이 사용하는 것이 코드의 작성이나 읽기를 쉽게 하는 경우가 많습니다. 따라서 위의 첫 번째 예제는 다음과 같이 쓸 수 있습니다.

```
X + (Y * Z)
```

여기서 괄호는 (컴파일러에게) 불필요하지만, 프로그래머와 코드를 읽어보는 사람들에게는 연산자 우선 순위를 생각해보지 않아도 되게 해줍니다.

함수 호출

함수는 값을 리턴하므로, 함수 호출도 표현식입니다. 예를 들어, 두 개의 정수 인수를 가지고 있고 정수를 리턴하는 Calc라는 함수를 정의했다면, 함수 호출 Calc(24, 47)은 정수 표현식입니다. I와 J가 정수 변수인 경우 I + Calc(J, 8) 역시 정수 표현식입니다. 함수 호출의 예제는 다음과 같습니다.

```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Radius, Height)
GetValue
TSomeObject.SomeMethod(I, J);
```

함수에 대한 자세한 내용은 5장 "프로시저 및 함수"를 참조하십시오.

집합 생성자

집합 생성자는 집합 타입의 값을 나타냅니다. 예를 들면, 다음과 같습니다.

```
[5, 6, 7, 8]
```

이 문장은 요소가 5, 6, 7, 8인 집합을 나타냅니다. 다음의 집합 생성자는,

```
[ 5..8 ]
```

위의 집합과 같은 집합을 나타냅니다.
집합 생성자에 대한 문법은 다음과 같습니다.

```
[ item1, ..., itemn ]
```

여기서 각각의 *item*은 집합의 기본 타입의 서수(*ordinal*)를 나타내는 표현식이거나 이런 표현식 두개 사이에 두 개의 마침표(..)가 있는 것입니다. *item*의 형태가 x..y인 경우 x부터 y 범위에 있는 모든 수를 간단하게 표기한 것입니다. 그러나 x가 y보다 큰 경우, [x..y]는 아무것도 나타내지 않고 공집합이 됩니다. 집합 생성자 [ ]는 공집합을 나타내며, [x]는 x의 값을 유일한 요소로 가진 집합을 나타냅니다.
집합 생성자의 예는 다음과 같습니다.

```
[red, green, MyColor]
[1, 5, 10..K mod 12, 23]
['A'..'Z', 'a'..'z', Chr(Digit + 48)]
```

집합에 대한 자세한 내용은 4장의 "구조 타입" 절에서 "집합"을 참조하십시오.

인덱스

문자열, 배열, 배열 속성, 문자열이나 배열에 대한 포인터는 인덱스를 사용할 수 있습니다. 예를 들어, FileName이 문자열 변수인 경우 표현식 FileName[3]은 FileName 문자열에서

세 번째 문자를 리턴하고, FileName[I + 1]은 I번째 인덱스의 다음 문자를 리턴합니다. 문자열에 대한 내용은 4장의 "문자열 타입"절을 참조하십시오. 배열과 배열 속성에 대한 내용은 4장의 "구조 타입" 절 "배열"과 6장의 "속성" 절 "배열 속성"을 참조하십시오.

타입 캐스트

표현식이 다른 타입인 것처럼 다루는 것이 유용할 경우가 있습니다. 이를 위해 타입 캐스트(typecast)을 이용하여 표현식의 타입을 일시적으로 변화시킬 수 있습니다. 예를 들어, Integer('A')는 A를 정수로 변환합니다.
타입 캐스트의 문법은 다음과 같습니다.

```
typeIdentifier(expression)
```

표현식이 변수인 경우에는 변수 타입 캐스트라고 부릅니다. 그 외의 경우는 값 타입 캐스트입니다. 변수 타입 캐스트와 값 타입 캐스트의 문법은 동일하지만, 두 종류의 타입 캐스트에는 서로 다른 규칙이 적용됩니다.

값 타입 캐스트

값 타입 캐스트(value typecast)에서는, 타입 식별자와 타입 캐스트 표현식은 모두 서수(ordinal) 타입이거나 포인터 타입이어야 합니다. 값 타입 캐스트의 예는 다음과 같습니다.

```
Integer('A')
Char(48)
Boolean(0)
Color(2)
Longint(@Buffer)
```

결과 값은 괄호 안의 표현식을 변환함으로써 얻습니다. 지정된 타입의 크기가 표현식의 타입 크기와 다른 경우 잘리거나 늘어날 수도 있습니다. 표현식 부호는 항상 유지됩니다.
문법이 다음과 같을 경우

```
I := Integer('A');
```

이 문법은 Integer('A')의 값, 즉 65를 변수 I에 대입합니다.
값 타입 캐스트 뒤에는 한정자가 올 수 없고, 대입문의 왼쪽에 나타날 수 없습니다.

변수 타입 캐스트

어떤 변수든 어떤 타입으로도 타입 캐스트할 수 있지만, 크기가 동일하고 정수와 실수를 함께 사용하지 않아야 합니다. (숫자 타입을 타입 캐스트하려면 Int와 Trunc 같은 표준 함수를 사용하십시오.) 변수 타입 캐스트의 예는 다음과 같습니다.

```
Char(I)
Boolean(Count)
TSomeDefinedType(MyVariable)
```

변수 타입 캐스트는 대입문의 어느 쪽이든 사용할 수 있습니다. 따라서 다음과 같은 경우,

```
var MyChar: char;
...
Shortint(MyChar) := 122;
```

이 문장은 MyChar에 z 문자(ASCII 122)를 지정합니다.
변수를 프로시저 타입으로 변환할 수도 있습니다. 예를 들어, 다음과 같이 선언한 경우,

```
type Func = function(X: Integer): Integer;
var
  F: Func;
  P: Pointer;
  N: Integer;
```

다음과 같은 대입문들이 가능합니다.

```
F := Func(P);            { P의 프로시저 값을 F에 대입 }
Func(P) := F;            { F의 프로시저 값을 P에 대입 }
@F := P;                 { P의 포인터 값을 F에 대입 }
P := @F;                 { F의 포인터 값을 P에 대입 }
N := F(N);               { F의 함수를 호출 }
N := Func(P)(N);         { P의 함수를 호출 }
```

다음의 예와 같이 변수 타입 캐스트를 한정자 와 함께 사용할 수 있습니다.

```
type
  TByteRec = record
    Lo, Hi: Byte;
  end;
  TWordRec = record
    Low, High: Word;
  end;
  PByte = ^Byte;
var
  B:Byte;
  W:Word;
  L:Longint;
  P:Pointer;
begin
  W := $1234;
  B := TByteRec(W).Lo;
  TByteRec(W).Hi := 0;
  L := $01234567;
  W := TWordRec(L).Low;
  B := TByteRec(TWordRec(L).Low).Hi;
  B := PByte(L)^;
end;
```

이러한 예에서, TByteRec는 워드의 하위 및 상위 바이트를 액세스하는 데 사용되고, TWordRec은 longint의 하위 및 상위 워드를 액세스하는 데 사용됩니다. 같은 목적으로 델파이에 내장된 함수 Lo와 Hi를 호출할 수도 있지만, 변수 타입 캐스트는 대입문의 왼쪽에서 사용할 수 있는 잇점이 있습니다.

타입 캐스트 포인터에 대한 내용은 4장의 "포인터와 포인터 타입" 절을 참조하십시오. 클래스와 인터페이스 타입 캐스트에 대한 내용은 6장의 "클래스 참조" 절의 "as 연산자"와 11장의 "인터페이스 참조" 절의 "인터페이스의 타입 캐스트"을 참조하십시오.

선언과 문장

uses 절과 유닛의 영역을 구분하는 implementation 같은 예약어들을 제외하면, 프로그램은 선언(declaration)과 문장(statement)으로 이루어지며, 블럭(block)으로 묶여 구성됩니다.

선언

변수, 상수, 타입, 필드, 속성, 프로시저, 함수, 프로그램, 유닛, 라이브러리, 패키지의 이름을 식별자(identifier)라고 합니다. 26057과 같은 숫자 상수는 식별자가 아닙니다. 식별자는 사

용하기 전에 먼저 선언되어 있어야 합니다. 단, 컴파일러가 자동으로 이해하는 몇몇 이미 정의된 타입/루틴/상수들, 그리고 함수 블럭에서 나타나는 변수 Result와 메소드 구현에서 나타나는 변수 Self는 예외입니다.
선언은 식별자를 정의하고 적절한 곳에 식별자에 대해 메모리를 할당합니다. 예를 들면,

```
var Size: Extended;
```

위의 문장은 Extended(실수) 값을 가지고 있는 Size라는 변수를 선언하며,

```
function DoThis(X, Y: string): Integer;
```

위 문장은 인수로 두 개의 문자열을 가지고 정수를 리턴하는 DoThis라는 함수를 선언합니다. 각 선언은 세미콜론으로 끝납니다. 여러 변수, 상수, 타입이나 레이블을 동시에 선언하는 경우에는 적절한 예약어를 다음과 같이 한 번만 사용하면 됩니다.

```
var
  Size: Extended;
  Quantity: Integer;
  Description: string;
```

선언의 문법과 위치는 사용자가 정의하는 식별자의 종류에 따라 다릅니다. 일반적으로 선언은 블럭의 시작 부분 혹은 유닛의 인터페이스나 임플먼테이션 섹션의 시작 부분(uses 절 뒤에)에만 나타날 수 있습니다. 변수, 상수, 타입, 함수 등의 선언에 대한 특수한 문법은 각 장의 해당 항목에 설명되어 있습니다.

■힌트 지시어

"힌트" 지시어인 platform, deprecated 및 library는 어떤 선언에도 추가될 수 있습니다. 이 지시어들은 컴파일 때 경고를 내도록 합니다. 힌트 지시어는 타입 선언, 변수 선언, 클래스, 인터페이스, 구조 선언, 클래스나 레코드 내의 필드 선언, 함수와 메소드 선언, 유닛 선언에 적용될 수 있습니다.
힌트 지시어가 유닛 선언에 있으면 해당 힌트가 유닛 전체에 적용된다는것을 의미합니다. 예를 들어, 윈도우 3.1 스타일 OleAuto.pas 유닛은 사용하지 말 것을 권합니다(deprecated). 해당 유닛이나 그 안의 심볼에 대한 참조는 비추천(deprecation) 메시지를 내게 됩니다.

심볼 혹은 유닛에 대한 platform 힌트 지시어는 다른 플랫폼들에서 존재하지 않거나 다르게 구현되었을 수 있다는 것을 의미합니다. 심볼이나 유닛에 대한 library 힌트 지시어는 다른 라이브러리 아키텍처에서 해당 코드가 존재하지 않거나 다르게 구현되었을 수 있다는 것을 의미합니다.

platform과 library 지시어는 특정 플랫폼이나 라이브러리를 지정하지는 않습니다. 만약 플랫폼에 독립적인 코드를 작성하려는 것이 목적이라면, 심볼이 특정되는 플랫폼이 어떤 것인지 알 필요가 없습니다. 호환성이라는 목적에 관련하여 문제를 일으킬 수도 있다는 점을 알 수 있도록 심볼이 '어떤' 플랫폼에 특정된다는 것을 표시하는 것만으로 충분합니다.

프로시저나 함수의 경우에는, 세미콜론(;)으로 힌트 지시어를 다른 선언들과 구분해야 합니다. 예를 들면,

```
procedure SomeOldRoutine; stdcall; deprecated;
var
   VersionNumber: Real library;
type
   AppError = class(Exception)
     ...
end platform;
```

소스 코드가 {$HINTS ON} {$WARNINGS ON} 모드로 컴파일되면, 이러한 지시어들로 선언된 식별자를 참조하는 곳마다 해당하는 힌트나 경고를 냅니다. Windows 같은 특정 운영 체제에 고유한 항목을 표시하려면 platform을 사용하고, 더 이상 쓰이지 않거나(obsolete) 항목이 하위 호환성을 위해서만 지원된다는 점을 나타내려면 deprecated를 사용하며, 특정 라이브러리나 컴포넌트 프레임워크에 의존한다는 점을 표시하려면 library를 사용합니다.

문장

문장(statement)은 프로그램 내의 알고리즘 동작을 정의합니다. 대입문과 프로시저 호출문 같은 단순문을 조합하여 순환문(loop), 조건문(conditional statement) 및 기타 구조문(structured statement)을 만들 수 있습니다.

블럭 내의 문장들과 유닛의 이니셜라이제이션 섹션이나 파이널라이제이션 섹션에서 사용된 문장들은 세미콜론으로 구분됩니다.

단순문

단순문(simple statement)은 다른 문장을 포함하지 않습니다. 단순문에는 대입문, 프로시저와 함수 호출, goto 문이 포함됩니다.

■ 대입문

대입문(assignment statement)의 형식은 다음과 같습니다.

```
variable := expression
```

여기서 *variable*은 어떤 변수 참조 형태이든 가능한데, 여기에는 변수, 변수 타입 캐스트, 역참조(dereference) 포인터, 구조(structured) 변수의 컴포넌트 등이 있습니다. *expression*은 대입이 가능한 모든 표현식이 될 수 있습니다. (함수 블럭 안에서 해당 함수 자체의 이름을 *variable*로 쓸 수도 있습니다. 5장 "프로시저 및 함수"를 참조하십시오.) := 기호를 대입 연산자라고 부르기도 합니다.

대입문은 *variable*의 현재 값을 *expression*의 값으로 바꿉니다. 예를 들면,

```
I := 3;
```

이 문장은 변수 I에 값 3을 대입합니다. 대입 연산자의 왼쪽에 있는 변수 참조가 오른쪽의 표현식에 사용될 수도 있습니다. 예를 들면,

```
I := I + 1;
```

이 문장은 I의 값을 증가시킵니다. 다른 대입문의 예는 다음과 같습니다.

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
Hue1 := [Blue, Succ(C)];
I := Sqr(J) - I  * K;
Shortint(MyChar) := 122;
TByteRec(W).Hi := 0;
MyString[I] := 'A';
SomeArray[I + 1] := P^;
TMyObject.SomeProperty := True;
```

■ 프로시저 및 함수 호출

프로시저 호출은 프로시저 이름(한정자가 지정될 수도 있습니다)과 (필요한 경우) 파라미터 리스트로 구성됩니다. 다음과 같은 예들이 있습니다.

```
PrintHeading;
Transpose(A, N, M);
Find(Smith, William);
Writeln('Hello world!');
DoSomething();
Unit1.SomeProcedure;
TMyObject.SomeMethod(X,Y);
```

확장 문법이 활성화된 상태에서는({$X+}) 함수 호출은 프로시저 호출과 마찬가지로 문장으로서 사용될 수 있습니다.

```
MyFunction(X);
```

함수 호출을 이런 방식으로 사용할 경우 리턴 값은 버려집니다.
프로시저와 함수에 대한 자세한 내용은 5장 "프로시저 및 함수"를 참조하십시오.

■ Goto 문

goto 문의 형식은 다음과 같습니다.

```
goto label
```

이 문장은 프로그램 실행을 지정한 레이블이 표시된 문장으로 이동합니다. 문장에 레이블을 표시하려면 먼저 레이블을 선언해야 합니다. 그런 다음, 레이블로 표시하려는 문장 앞에 다음과 같이 레이블과 콜론을 써넣습니다.

```
label: statement
```

레이블은 다음과 같이 선언합니다.

```
label label;
```

다음과 같이 여러 개의 레이블을 한 번에 선언할 수도 있습니다.

```
label label1, ..., labeln
```

레이블은 유효한 식별자 또는 0과 9999 사이의 숫자를 사용할 수 있습니다.
레이블 선언, 레이블이 표시된 문장, 그리고 goto 문은 같은 블럭에 있어야 합니다.(3장의 "블럭과 유효 범위" 절을 참조) 따라서 프로시저나 함수의 안이나 밖으로 이동하는 것은 불가능합니다. 한 블럭에서 둘 이상의 문장을 같은 레이블로 표시하지 마십시오.
예를 들면, 다음과 같습니다.

```
label StartHere;
...
StartHere: Beep;
goto StartHere;
```

이 문장은 Beep 프로시저를 반복적으로 호출하는 무한 루프를 만듭니다.
또, try-finally나 try-except 문의 안이나 바깥으로 이동하는 것은 불가능합니다.
일반적으로 구조적 프로그램에서는 goto 문을 사용하지 않는 것이 좋습니다. 그러나 다음 예제처럼 중첩된 루프로부터 빠져나가는 방법으로 사용되는 경우도 있습니다.

```
procedure FindFirstAnswer;
var X, Y, Z, Count: Integer;
label FoundAnAnswer;
begin
  Count := SomeConstant;
  for X := 1 to Count do
    for Y := 1 to Count do
      for Z := 1 to Count do
        if ... { X, Y, Z가 관계된어떤 조건 } then
          goto FoundAnAnswer;
  ... {답을 찾지 못했을 경우 실행할 코드 }
  Exit;

  FoundAnAnswer:
  ... {답을 찾았을 경우 실행할 코드 }
end;
```

중첩 루프를 빠져나가기 위해 goto 문을 사용했다는 점을 눈여겨보십시오. 루프나 구조문의

안으로 이동해서는 안됩니다. 예기치 못한 결과를 일으킬 수 있기 때문입니다.

구조문

구조문(structured statement)은 다른 문장들로 이루어집니다. 다른 문장들을 차례대로 또는 어떤 조건 하에, 또는 반복적으로 실행하고자 하는 경우 구조문을 사용합니다.

- 복합문 또는 with 문은 단순히 구성 문장들의 순서대로 실행합니다.
- 조건문 즉, if 또는 case 문은 지정된 조건에 따라 구성 문장들 중 하나가 실행되거나 전혀 실행되지 않습니다.
- repeat, while 및 for 등의 순환문(loop statement)은 구성 문장들의 순서대로 반복해서 실행합니다.
- raise, try...except 및 try...finally 등의 특수한 문장들은 예외(exception)를 생성하고 처리합니다. 예외 생성과 처리에 대한 내용은 6장의 "예외" 절을 참조하십시오.

■ 복합문

복합문(compound statement)은 다른 타입의 문장들(단순문 또는 구조문)의 연속으로서, 작성된 순서대로 실행됩니다. 복합문은 예약어 begin 및 end로 묶이고, 복합문 내의 문장들은 세미콜론으로 구분합니다. 예를 들면, 다음과 같습니다.

```
begin
  Z := X;
  X := Y;
  Y := Z;
end;
```

end 앞의 마지막 세미콜론은 옵션입니다. 그러므로 다음과 같이 쓸 수 있습니다.

```
begin
  Z := X;
  X := Y;
  Y := Z
end;
```

복합문은 델파이 문법에서 하나의 문장이 필요한 상황에서 필수적입니다. 프로그램, 함수나

프로시저 블럭 외에도, 조건문이나 순환문과 같은 구조문 내에 나타날 수 있습니다. 예를 들면, 다음과 같습니다.

```
begin
  I := SomeConstant;
  while I > 0 do
  begin
    ...
    I := I - 1;
  end;
end;
```

내부의 문장이 단 하나인 복합문을 작성할 수도 있습니다. 복잡한 문장에서의 괄호와 마찬가지로 begin과 end로 명확하게 표시함으로써 코드의 가독성을 높일 수 있습니다. 또한 빈 복합문을 사용하여 내용이 없는 블럭을 만들 수도 있습니다.

```
begin
end;
```

■with 문

with 문은 레코드의 필드나 객체의 필드, 속성, 메소드를 참조하기 위한 간단 표기 방식입니다. with 문의 문법은 다음과 같습니다.

```
with obj do statement
```

또는,

```
with obj1, ..., objn do statement
```

여기서, *obj*는 레코드 참조나 객체 인스턴스, 클래스 인스턴스, 인터페이스나 클래스 타입(메타클래스) 인스턴스를 나타내는 변수 참조이고, *statement*는 단순문이나 구조문입니다. *statement* 내에서 한정자 없이 식별자만으로 *obj*의 필드, 속성, 메소드를 참조할 수 있습니다.

예를 들어, 다음과 같이 선언했다고 가정해 보십시오.

```
type TDate = record
  Day: Integer;
  Month: Integer;
  Year: Integer;
end;

var OrderDate: TDate;
```

다음과 같은 with 문을 작성할 수 있습니다.

```
with OrderDate do
  if Month = 12 then
  begin
    Month := 1;
    Year := Year + 1;
  end
  else
    Month := Month + 1;
```

이 문장은 다음과 동일합니다.

```
if OrderDate.Month = 12 then
begin
  OrderDate.Month := 1;
  OrderDate.Year := OrderDate.Year + 1;
end
else
  OrderDate.Month := OrderDate.Month + 1;
```

*obj*의 해석에 배열 인덱싱이나 포인터 역참조가 포함되면, 이러한 동작은 *statement*가 실행되기 전에 한 번 수행됩니다.

따라서 with 문은 간결할 뿐만 아니라 더 효율적이 됩니다. 또한 with 문을 실행하는 동안은 *statement* 내에서 변수에 대한 대입 동작은 *obj*의 해석에 영향을 미지 않는다는 것을 의미합니다.

with 문 안의 각각의 변수 참조나 메소드 이름은 가능한 경우 지정된 객체나 레코드의 멤버로서 해석됩니다. with 문 안에서 액세스하려는 것과 같은 이름을 가진 변수나 메소드가 있을 경우, 다음 예제에서와 같이 한정자를 앞에 덧붙여야 합니다.

```
with OrderDate do
begin
  Year := Unit1.Year
  ...
end;
```

with 뒤에 객체나 레코드가 여러 개 있으면, 중첩된 with 문처럼 처리됩니다. 따라서,

```
with obj1, obj2, ..., objn do statement
```

이 문장은 다음과 같습니다.

```
with obj1 do
  with obj2 do
    ...
    with objn do
      statement
```

이 경우에, *statement*에 있는 각 변수 참조 이름이나 메소드 이름은 가능하면 obj_n의 멤버로 해석됩니다. 그렇지 않은 경우 가능하면 obj_{n-1}의 멤버로 해석되며, 그렇지 않은 경우 다시 더 상위의 with 문으로 반복해서 해석을 시도해나갑니다. obj 자신들을 해석하는 데에도 같은 규칙이 적용됩니다. 예를 들어 obj_n가 obj_1과 obj_2 모두의 멤버인 경우, $obj_2.obj_n$으로 해석됩니다.

■if 문

if 문에는 if...then과 if...then...else의 두가지 형태가 있습니다. if...then 문장의 문법은 다음과 같습니다.

```
if expression then statement
```

여기서, *expression*은 부울 값을 리턴하는 표현식입니다. *expression*이 True인 경우 *statement*가 실행되고 그렇지 않은 경우 실행되지 않습니다. 예를 들면,

```
if J <> 0 then Result := I/J;
```

if...then...else 문의 문법은 다음과 같습니다.

```
if expression then statement1 else statement2
```

여기서, *expression*은 부울 값을 리턴합니다. *expression*이 True인 경우 *statement1*이 실행되고 그렇지 않은 경우 *statement2*가 실행됩니다. 예를 들면,

```
if J = 0 then
  Exit
else
  Result := I / J;
```

then과 else 절은 각각 하나의 문장을 포함하지만, 이 각 문장은 구조문이 될 수도 있습니다. 예를 들면,

```
if J <> 0 then
begin
  Result := I/J;
  Count := Count + 1;
end
else if Count = Last then
  Done := True
else
  Exit;
```

then 절과 else 사이에 세미콜론이 없다는 사실에 주의하십시오. 블럭 내에서 if 문을 다음 문장과 구분하기 위해 전체 if 문 뒤에 세미콜론을 사용할 수 있지만, then과 else 절 사이에는 공백이나 캐리지 리턴 외에 다른 것은 필요 없습니다. if 문의 else 바로 앞에 세미콜론을 사용하는 것은 흔한 프로그래밍 실수입니다.

연속으로 중첩된 if 문을 사용할 때에는 특별한 주의가 필요합니다. 일부 if 문에는 else 절이 있고, 다른 if 문에서는 else 절이 없기 때문에 이러한 문제가 발생합니다. if 문에 비해 else 절이 적은 연속된 중첩 조건문에서는, 어떤 else 절이 어떤 if에 속하는지 명확하게 보이지 않을 수도 있습니다.

```
if expression1 then if expression2 then statement1 else statement2;
```

이를 해석하는 방법에는 두가지 방법이 있을 것입니다.

```
if expression1 then [ if expression2 then statement1 else statement2 ];
if expression1 then [ if expression2 then statement1 ] else statement2 ;
```

컴파일러는 항상 첫 번째 방법으로 해석합니다. 즉, 실제 코드에서 다음 문장은,

```
if ...{ expression1 } then
  if ...{ expression2 } then
    ... { statement1 }
  else
    ... { statement2 } ;
```

다음 문장과 동일합니다.

```
if ...{ expression1 } then
begin
  if ...{ expression2 } then
    ... { statement1 }
  else
    ... { statement2 }
end;
```

중첩된 if 문을 해석할 때의 규칙은, 가장 안쪽에 있는 조건문부터 시작하여 각각의 else를 왼쪽 방향으로 가장 가까운 if로 묶으면서 해석한다는 것입니다. 컴파일러가 위의 두 번째 방식으로 이해하도록 하려면 다음과 같이 명시적으로 작성해야만 합니다.

```
if ...{ expression1 } then
begin
  if ...{ expression2 } then
    ... { statement1 }
end
else
  ... { statement2 } ;
```

■case 문

case 문은 복잡하게 중첩된 if 조건문에 비해 더 이해하기 쉬운 대안이 될 수 있습니다. case 문의 형식은 다음과 같습니다.

```
case selectorExpression of
  caseList1: statement1;
  ...
  caseListn: statementn;
end
```

여기서, *selectorExpression*은 서수(ordinal) 타입(문자 타입은 사용할 수 없음) 표현식이고, *caseList*는 다음 중 하나입니다.

- 선언된 숫자 상수이거나 컴파일러가 프로그램을 실행하지 않고 계산할 수 있는 기타 표현식. *selectorExpression*과 호환되는 서수 타입이어야 합니다. 따라서 7, True, 4 + 5 * 3, 'A' 및 Integer('A')는 모두 *caseList*로 사용할 수 있지만, 변수와 대부분의 함수 호출은 사용할 수 없습니다. (Hi와 Lo 같은 몇몇 내장 함수는 *caseList*에 사용할 수 있습니다. 4장의 "선언된 상수" 절의 "상수 표현식"을 참조)
- *First..Last* 형식의 부분 범위(subrange). 여기서, *First*와 *Last*는 위의 조건을 모두 만족해야 하며 First는 Last보다 작거나 같아야 합니다.
- *item*1, ..., *item*n 형식의 리스트. 여기서, 각 *item*은 위의 조건들 중 하나를 만족해야 합니다.

*caseList*가 나타내는 각각의 값은 case 문에서 고유해야 합니다. 즉, 부분 범위와 리스트가 겹칠 수 없습니다. case 문의 마지막에는 else 절이 올 수 있습니다.

```
case selectorExpression of
  caseList1: statement1;
  ...
  caseListn: statementn;
else
  statements;
end
```

여기서, 각 *statement*는 세미콜론으로 구분된 여러 문장들입니다. case 문이 실행되면 *statement*1 ... *statement*n 중에 하나의 문장만 실행되거나 전혀 실행되지 않습니다.

*selectorExpression*과 같은 값의 *caseList*가 어느 *statement*가 실행될지를 결정합니다. *caseList* 중에서 *selectorExpression*과 같은 값이 없을 경우, else 절이 있으면 else 절의 문장이 실행됩니다.
다음 case 문은

```
case I of
  1..5: Caption := 'Low';
  6..9: Caption := 'High';
  0, 10..99: Caption := 'Out of range';
else
  Caption := '';
end;
```

다음의 중첩된 조건문과 동일합니다.

```
if I in [1..5] then
  Caption := 'Low'
  else if I in [6..10] then
    Caption := '';
    else if (I = 0) or (I in [10..99]) then
      Caption := 'Out of range'
      else
        Caption := '';
```

case 문의 다른 예제는 다음과 같습니다.

```
case MyColor of
  Red: X := 1;
  Green: X := 2;
  Blue: X := 3;
  Yellow, Orange, Black: X := 0;
end

case Selection of
  Done: Form1.Close;
  Compute: CalculateTotal(UnitCost, Quantity);
else
  Beep;
end
```

■ 순환문

순환문(루프, loop)는 언제 실행을 중단시킬지를 결정할 제어 조건이나 변수를 이용하여 일련의 문장들을 반복적으로 실행할 수 있게 해줍니다. 델파이에는 repeat 문, while 문 및 for 문의 세가지 제어 루프(control loop)가 있습니다.

표준 Break와 Continue 프로시저를 사용하여 repeat, while, for 문의 흐름을 제어할 수 있습니다. Break는 Break가 있는 곳에서 문장을 종료하며, Continue는 순환문 내에서 그 이후의 문장들을 무시하고 순환문의 다음 반복을 실행합니다. 이러한 프로시저에 대한 자세한 내용은 온라인 헬프를 참조하십시오.

■ repeat 문

repeat 문의 문법 형식은 다음과 같습니다.

```
repeat statement1; ...; statementn; until expression
```

여기서, *expression*은 부울 값을 리턴합니다. until 앞에 있는 마지막 세미콜론(;)은 옵션입니다. repeat 문은 repeat 문 내에 있는 문장들을 차례로 실행하고, 매번 반복이 끝난 후에 *expression*을 테스트합니다. *expression*이 True를 리턴하면 repeat 문은 종료됩니다. 첫번째 반복이 끝나기 전까지는 *expression*을 계산하지 않기 때문에 repeat 내에 있는 문장들은 최소한 한 번 이상 실행됩니다.

repeat 문의 예제는 다음과 같습니다.

```
repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;

repeat
  Write('Enter a value (0..9): ');
  Readln(I);
until (I >= 0) and (I <= 9);
```

■ while 문

while 문은 repeat 문과 유사하지만, while 문 내에 있는 문장들을 실행하기 전에 제어 조건을 계산한다는 점이 다릅니다. 따라서 조건이 False면 while 문 내에 있는 문장들은 전혀 실

행되지 않습니다.
while 문의 문법 형식은 다음과 같습니다.

```
while expression do statement
```

여기서, *expression*은 부울 값을 리턴하며, *statement*는 복합문일 수 있습니다. while 문은 그 안의 *statement*를 반복적으로 실행하고, 매번 반복하기 전에 *expression*을 검사합니다. *expression*이 True를 리턴하는 한, while 문은 계속 실행됩니다.
while 문의 예는 다음과 같습니다.

```
while Data[I] <> X do I := I + 1

while I > 0 do
begin
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end

while not Eof(InputFile) do
begin
  Readln(InputFile, Line);
  Process(Line);
end;
```

■ for 문
repeat 또는 while 문과 달리, for 문은 루프의 반복 횟수를 명시적으로 지정해야 합니다. for 문의 문법 형태는 다음과 같습니다.

```
for counter := initialValue to finalValue do statement
```

또는,

```
for counter := initialValue downto finalValue do statement
```

여기서,

- *counter*는 for 문을 포함한 블럭에서 선언된 서수 타입의 지역 변수이며, 한정자가 없습니다.
- *initialValue*와 *finalValue*는 *counter*에 대입 호환 가능한 표현식입니다.
- *statement*는 *counter*의 값을 변경하지 않는 단순문 또는 구조문입니다.

주의!
반복 값 counter는 루프 진행중에 변경되어서는 안됩니다. 여기에는 대입이나 프로시저의 var 파라미터로 전달하는 경우도 포함됩니다. counter를 변경하는 동작은 컴파일 중에 경고를 일으킵니다.

for 문은 *initialValue* 값을 *counter*에 대입한 다음, *statement*를 반복해서 실행하면서 각각의 반복 후에 *counter*를 증가 혹은 감소시킵니다. (for...to 문법은 counter를 증가시키고 for...downto문법은 감소시킵니다.) *counter*가 *finalValue*와 같은 값을 리턴하면 *statement*가 한 번 더 실행되고 for 문이 종료됩니다. 다시 말해, *statement*는 *initialValue*부터 *finalValue*까지의 범위에 있는 모든 값에 대해 한 번씩 실행됩니다. initialValue가 *finalValue*와 같을 경우, *statement*는 단 한 번만 실행됩니다. *initialValue*가 for...to 문에서 *finalValue*보다 크거나, for...downto 문에서 *finalValue*보다 작으면, *statement*는 전혀 실행되지 않습니다. for 문이 종료된 후에(Break나 Exit에 의해 강제 종료된 경우가 아니라면) *counter*의 값은 정의되지 않습니다. 표현식 *initialValue*와 *finalValue*는 루프의 실행을 제어하기 위해 루프가 시작되기 전에 단 한 번만 계산됩니다. 따라서 for...to 문은 다음의 while 문법과 (완전히는 아니지만) 거의 동일합니다.

```
begin
  counter := initialValue
  while counter <= finalValue do
  begin
    statement;
    counter := Succ(counter);
  end
end
```

이 문법과 for...to 문의 차이는, while 루프는 각각의 반복 이전에 *finalValue*를 다시 계산한다는 것입니다. *finalValue*가 복잡한 표현식인 경우 이 때문에 현저하게 느려질 수 있고, 이는 *statement* 내에서 *finalValue* 값을 변경하면 루프의 실행에 영향을 줄 수 있다는 것을 의미합니다.

for 문의 예는 다음과 같습니다.

```
for I := 2 to 63 do
  if Data[I] > Max then
    Max := Data[I];

for I := ListBox1.Items.Count - 1 downto 0 do
  ListBox1.Items[I] := UpperCase(ListBox1.Items[I]);
```

```
for I := 1 to 10 do
  for I := 1 to 10 do
  begin
    X := 0;
    for I := 1 to 10 do
    X := X + Mat1[I, K] * Mat2[K, J];
    Mat[I, J] := X;
  end;

for C := Red to Blue do Check(C);
```

■ for 문으로 컨테이너 반복

델파이 컴파일러는 컨테이너에 대해 for-element-in-collection 스타일의 반복을 지원합니다. 델파이 컴파일러는 다음과 같은 컨테이너 반복 패턴들을 인식할 수 있습니다.

```
for Element in ArrayExpr do statement;
for Element in StringExpr do statement;
for Element in SetExpr do statement;
for Element in CollectionExpr do statement;
for Element in Record do statement;
```

주의!

반복 값 counter는 루프 진행중에 변경되어서는 안됩니다. 여기에는 대입이나 프로시저의 var 파라미터로 전달하는 경우도 포함됩니다. counter를 변경하는 동작은 컴파일 중에 경고를 일으킵니다.

반복 변수 *Element*는 컨테이너가가지고 있는 타입과 일치해야 합니다. 루프의 각 반복 때마다 반복 변수는 컬렉션 멤버의 현재 값을 가집니다. 일반적인 for 루프와 마찬가지로, 반복 변수는 for 문과 같은 블럭에서 선언되어 있어야 합니다.

배열 표현식은 1차원 혹은 다차원, 고정 길이, 동적 배열일 수 있습니다. 배열 요소들에 대한 탐색은 증가 방향으로 진행되므로, 배열의 가장 작은 인덱스 요소부터 시작해서 배열 크기 빼기 1로 끝납니다. 다음의 코드는 1차원 배열, 다차원 배열, 동적 배열에 대한 예를 보여줍니다.

```
type
  TIntArray        = array[0..9] of Integer;
  TGenericIntArray = array of Integer;
var
  IArray1: array[0..9] of Integer   = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
  IArray2: array[1..10] of Integer  = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
  IArray3: array[1..2] of TIntArray = ((11, 12, 13, 14, 15, 16, 17, 18, 19, 20),
                                       (21, 22, 23, 24, 25, 26, 27, 28, 29, 30));
  MultiDimTemp: TIntArray;
  IDynArray: TGenericIntArray;
  I: Integer;
```

```
begin
  for I in IArray1 do
  begin
    // I에 대해 어떤 작업을 실행...
  end;

  // 인덱스는 배열의 최저 요소인 1에서 시작합니다.
  for I in IArray2 do
  begin
    // I에 대해 어떤 작업을 실행...
  end;

  // 다차원 배열에 대한 반복
  for MultiDimTemp in IArray3 do   // Indexing from 1..2
    for I in MultiDimTemp do       // Indexing from 0..9
    begin
      // I에 대해 어떤 작업을 실행...
    end;

  // 동적 배열에 대한 반복
  IDynArray := IArray1;
  for I in IDynArray do
  begin
    // I에 대해 어떤 작업을 실행...
  end;
```

다음은 문자열 표현식에 대한 반복의 예입니다.

```
var
  C: Char;
  S1, S2: String;
  Counter: Integer;
  OS1, OS2: ShortString;
  AC: AnsiChar;
begin
  S1 := 'Now is the time for all good men to come to the aid of their country.';
  S2 := '';

  for C in S1 do
    S2 := S2 + C;

  if S1 = S2 then
    WriteLn('SUCCESS #1');
  else
    WriteLn('FAIL #1');

  OS1 := 'When in the course of human events it becomes necessary to dissolve...';
```

```
  OS2 := '';

  for AC in OS1 do
    OS2 := OS2 + AC;

  if OS1 = OS2 then
    WriteLn('SUCCESS #2');
  else
    WriteLn('FAIL #2');
end.
```

다음은 집합(set) 표현식에 대한 반복의 예입니다.

```
type
  TMyThing = (one, two, three);
  TMySet   = set of TMyThing;
  TCharSet = set of Char;
var
  MySet:   TMySet;
  MyThing: TMyThing;

  CharSet: TCharSet;
  C: Char;

begin

  MySet := [one, two, three];
  for MyThing in MySet do
  begin
    // MyThing에 대해 작업을 실행...
  end;

  CharSet := [#0..#255];
  for C in CharSet do
  begin
    // C에 대해 작업을 실행...
  end;

end.
```

클래스나 인터페이스에서 for-in 루프를 사용하려면, 규정된 컬렉션 패턴을 구현해야 합니다. 컬렉션 패턴을 구현하는 타입은 다음과 같은 특성을 가져야 합니다.

- 해당 클래스 혹은 인터페이스는 퍼블릭 인스턴스 메소드 GetEnumerator()를 포함해야 합니다.

GetEnumerator() 메소드는 클래스, 인터페이스나 레코드 타입을 리턴해야 합니다.

- GetEnumerator()에 의해 리턴된 클래스, 인터페이스나 레코드는 퍼블릭 인스턴스 메소드 MoveNext()를 가져야 합니다. MoveNext() 메소드는 부울 값을 리턴해야 합니다.
- GetEnumerator()에 의해 리턴된 클래스, 인터페이스나 레코드는 퍼블릭 인스턴스 읽기 전용(read-only) 속성 Current를 가져야 합니다. Current 속성의 타입은 해당 컬렉션이 포함하는 타입이어야 합니다.

GetEnumerator()에 의해 리턴된 열거자(enumerator) 타입이 IDisposable 인터페이스를 구현하는 경우, 컴파일러는 루프가 종료될 때 그 타입의 Dispose 메소드를 호출하게 됩니다. 다음의 코드는 델파이에서 열거형 컨테이너에 대해 반복을 하는 예제입니다.

```
type
  TMyIntArray = array of Integer;

  TMyEnumerator = class
    Values: TMyIntArray;
    Index:  Integer;
  public
    constructor Create;
    function GetCurrent: Integer;
    function MoveNext: Boolean;
    property Current: Integer read GetCurrent;
  end;

  TMyContainer = class
  public
   function GetEnumerator: TMyEnumerator;
  end;

constructor TMyEnumerator.Create;
begin
  inherited Create;
  Values := TMyIntArray.Create(100, 200, 300);
  Index := -1;
end;

function TMyEnumerator.MoveNext: Boolean;
begin
  if Index < High(Values) then
  begin
    Inc(Index);
    Result := True;
  end
  else
```

```
    Result := False;
end;

function TMyEnumerator.GetCurrent: Integer;
begin
  Result := Values[Index];
end;

function TMyContainer.GetEnumerator: TMyEnumerator;
begin
  Result := TMyEnumerator.Create;
end;

var
  MyContainer: TMyContainer;
  I: Integer;

  Counter: Integer;

begin
  MyContainer := TMyContainer.Create;

  Counter := 0;
  for I in MyContainer do
    Inc(Counter, I);

  WriteLn('Counter = ', Counter);
end.
```

다음의 클래스들과 그 자손 클래스들은 for-in 문법을 지원합니다.

- TList
- TCollection
- TStrings
- TInterfaceList
- TComponent
- TMenuItem
- TCustomActionList
- TFields
- TListItems
- TTreeNodes
- TToolBar

블럭과 유효 범위

선언과 문장으로 구성되는 블럭은 레이블과 식별자에 대한 지역적인 네임스페이스, 또는 유효 범위(scope)를 정의합니다. 블럭은 변수 이름 등의 식별자는 프로그램의 다른 곳에서 다른 의미를 가질 수 있게 해줍니다. 모든 블럭은 프로그램, 함수나 프로시저의 선언의 일부이며, 각 프로그램, 함수 또는 선언은 하나의 블럭을 가집니다.

블럭

블럭(block)은 일련의 선언들과 그 뒤를 이은 복합문으로 구성됩니다. 모든 선언은 블럭의 시작 부분에 함께 나타나야 합니다. 그러므로 블럭의 형식은 다음과 같습니다.

```
declarations
begin
  statements
end;
```

declarations 영역에는 변수, 상수(리소스 문자열 포함), 타입, 프로시저, 함수, 레이블 등을 순서에 상관없이 선언합니다. 프로그램 블럭에서는 *declarations* 섹션이 하나 혹은 둘 이상의 exports 절을 포함할 수도 있습니다(10장 "DLL과 패키지" 참조).
예를 들어, 다음과 같은 함수 선언에서

```
function UpperCase(const S: string): string;
var
  Ch: Char;
  L: Integer;
  Source, Dest: PChar;
begin
  ...
end;
```

선언의 첫 번째 행은 함수 헤더 부분이며 나머지 행들이 블럭을 구성합니다. Ch, L, Source, Dest는 지역 변수로서, 그 선언은 오직 UpperCase 함수 블럭에만 적용되며, 이 블럭에서는 프로그램 블럭이나 유닛의 인터페이스/임플먼테이션 섹션에서 선언된 동일 이름의 식별자들이 있더라도 오버라이드, 즉 다른 선언들을 무시하고 우선하게 됩니다.

유효 범위

변수나 함수 이름과 같은 식별자는 선언의 유효 범위(scope) 내에서만 사용될 수 있습니다. 유효 범위는 선언의 위치에 따라 결정됩니다. 프로그램, 함수, 프로시저의 선언 내에서 선언된 식별자는 선언된 블럭에만 제한되는 유효 범위를 가집니다. 유닛의 인터페이스 섹션에서 선언된 식별자의 유효 범위는 그 유닛을 사용(uses)하는 다른 유닛 혹은 프로그램까지 포함됩니다. 작은 유효 범위를 가지는 식별자, 특히 함수와 프로시저에서 선언된 식별자를 지역(local)이라고 하고, 넓은 범위를 가지는 식별자를 전역(global)이라고 합니다.
식별자 범위를 정하는 규칙은 다음과 같이 요약할 수 있습니다.

식별자의 선언 위치	식별자의 유효 범위
프로그램, 함수, 프로시저의 선언 부분	선언된 지점부터 현재 블럭의 끝까지, 그리고 해당 유효 범위 내의 모든 블럭들
유닛의 인터페이스 섹션	선언된 지점부터 유닛의 끝까지, 그리고 해당 유닛을 사용(uses)하는 모든 유닛/프로그램 (2장 "프로그램과 유닛" 참조)
유닛의 임플먼테이션 섹션 (함수/프로시저 블럭이 아님)	선언된 지점부터 유닛의 끝까지. 식별자는 해당 유닛의 모든 함수/프로시저에서 사용 가능 (initialization/finalization 섹션 포함)
레코드 타입의 정의 (즉, 식별자가 레코드의 필드 이름)	선언된 지점부터 레코드 타입 정의의 끝까지 (4장 "레코드" 참조)
클래스의 정의 (즉, 식별자가 클래스의 데이터 필드 속성/메소드의 이름)	선언된 지점부터 클래스 타입 정의의 끝까지, 그리고 클래스의 자손들과 클래스의 모든 메소드들, 그 자손들. (6장 "클래스와 객체" 참조)

이름 충돌

하나의 블럭이 다른 블럭을 둘러싸는 경우, 둘러싸는 블럭을 외부 블럭(outer block)이라 하고 둘러싸인 블럭을 내부 블럭(inner block)이라고 합니다. 외부 블럭에서 선언된 식별자가 내부 블럭에서 다시 선언되는 경우, 내부 블럭의 영역 안에서는 내부 선언이 외부 블럭보다 우선하며 식별자의 의미를 결정합니다. 예를 들어, 사용자가 유닛의 인터페이스 섹션에서 MaxValue라는 변수를 선언하고, 그 유닛 내의 함수 선언에서 같은 이름으로 새로운 변수를 선언하는 경우, 그 함수 블럭에서는 한정자가 없는 MaxValue는 지역 선언인 두 번째 변수로 인식됩니다. 마찬가지로, 다른 함수 내에서 선언된 함수는, 외부 함수가 사용하는 식별자가 로컬로 다시 선언될 수 있는 새로운 내부 유효 범위(inner scope)가 됩니다.
유닛을 여러 개 사용하면 유효 범위의 정의가 더 복잡해집니다. uses 절에 나열된 각 유닛은 그 뒤의 유닛 리스트와 그 uses 절을 포함하는 프로그램이나 유닛을 포함하는 새로운 유효 범위를 만듭니다. uses 절에서 첫 번째 유닛은 가장 외부 범위를 나타내고, 그 다음의 유닛

들은 각각 이전 범위 내부의 새로운 범위를 나타냅니다. 두 개 이상의 유닛이 각각의 인터페이스 섹션에서 같은 식별자를 선언하는 경우, 식별자에 대해 한정되지 않은 참조는 가장 내부 유효 범위(innermost scope) 즉, 참조 자체가 나타나는 유닛에서 선언을 선택하거나, 그 유닛이 식별자를 선언하지 않는 경우 식별자에 대한 uses 절의 유닛 리스트에서 해당 식별자를 선언한 마지막 유닛이 지정됩니다.

System 유닛과 SysInit 유닛은 자동으로 모든 프로그램과 유닛에서 사용(uses)됩니다. System 유닛의 선언들은 이미 정의된 타입, 루틴, 컴파일러가 자동으로 이해하는 상수들와 함께 항상 가장 외부 범위를 가집니다.

한정된 식별자("식별자" 절의 "한정된 식별자"를 참조) 또는 with 문("선언과 문장" 절의 "with 문" 참조)을 사용하면 이러한 유효 범위에 관한 규칙들을 뛰어넘어서 내부 선언을 무시할 수 있습니다.