nML 프로그래밍 언어 참고서

⟨**nML** 컴파일러 버전 0.92b⟩

서 선 애
프로그램분석시스템연구단
한국과학기술원, 전산학과

Sunae Seo
Research On Program Analysis System
National Creative Research Initiative Center
Div. CS., KAIST

2002년 12월 20일

목 차

1 장

nML 언어

들어가며(Forward)

이 문서는 nML 언어의 참고서로 제작되었는데, nML 언어의 문법 (syntax)과 의미 (semantics)를 자연어로 설명한다. 이 문서에 설명되는 문법과 의미의 엄밀한 정의는 또 다른 문서 "nML 프로그램 문법, 실행, 기획[1]" 에 잘 나와있다.

이 문서는 nML 언어의 사용법을 설명하는 것이 아니라, nML 언어의 정의를 설명하는 것이므로, 예제 프로그램을 사용하지는 않는다.

표기법(Notation)

문법 구조를 정의할 때는 BNF 형태의 표기법을 따른다. 문법을 정의할 때 사용된 폰트는, 프로그램 텍스트인 경우 "`font`" 폰트를, 프로그램 텍스트 이외의 것에는 "*font*" 폰트를 사용하였다. '[', ']' 괄호는 선택적인 요소를 나타낼 때 사용하였고, '(', ')' 괄호는 그룹을 지을 때 사용하였다. '$()^*$' 는 0번이나 그 이상의 반복을 의미하고, '$()^+$' 는 한번 이상의 반복을 의미한다. 여러 가지 경우를 나타낼 때에는 '|'을 사용하는데, '|' 대신 사용된 † 는 설탕 구조(syntactic sugar)를 의미한다.

1 어휘를 만드는 방법 (Lexical conventions)

주석(Comments)

주석에는 두 가지 형태가 있다. 첫번째는 여러 줄을 한꺼번에 주석 처리하는 것이고, 두번째는 한 줄만을 주석 처리 하는 것이다. 여러 줄을 한꺼번에 주석 처리 할 때에는 주석 문자 (`*` 와 `*`) 을 사용한다. 이 문자들을 사용하면 그것들 사이의 모든 문자열(리턴을 포함해서)들은 무시된다. 이 주석 문자들은 항상 쌍으로 사용되는데 가장 안쪽의 것부터 짝을 이루게 된다. 이때, 가장 바깥쪽 주석 문자 쌍으로 둘러싸인 문자열이 모두 주석 처리된다. `//` 형태의 주석 문자는 `//` 로부터 그 줄의 끝까지를 모두 주석으로 간주한다.

[1] `http://ropas.kaist.ac.kr/n/n.pdf`

이름(Identifiers)

$$
\begin{aligned}
\mathit{lid} &::= (\texttt{a}-\texttt{z} \mid \mathit{hangul})(\texttt{a}-\texttt{z} \mid \texttt{A}-\texttt{Z} \mid \mathit{hangul} \mid \texttt{0}-\texttt{9} \mid \texttt{\_} \mid \texttt{'})^* \\
\mathit{uid} &::= (\texttt{A}-\texttt{Z} \mid \texttt{\_})(\texttt{a}-\texttt{z} \mid \texttt{A}-\texttt{Z} \mid \mathit{hangul} \mid \texttt{0}-\texttt{9} \mid \texttt{\_} \mid \texttt{'})^*
\end{aligned}
$$

프로그램 내의 이름들은 크게 두 가지로 구분된다. 알파벳의 소문자나 한글로 시작되는 이름(*lid*)과, 알파벳의 대문자나 _ 로 시작되는 이름(*uid*) 의 두 가지이다. 이처럼 이름을 구별해서 사용하는 이유는 프로그램을 읽기 쉽도록 하기 위해서이다. 변수 이름과 타입 이름은 *lid*로 표현되고, 모듈, 모듈 함수, 모듈 타입, 데이타 구성자, 예외상황 구성자들은 *uid* 로 표현된다(3절 참조).

정수 (Integer literals)

$$
\begin{aligned}
\mathit{integer} &::= (\texttt{0}-\texttt{9})^+ \\
&\mid (\texttt{0X} \mid \texttt{0x})(\texttt{0}-\texttt{9} \mid \texttt{A}-\texttt{F} \mid \texttt{a}-\texttt{f})^+ \\
&\mid (\texttt{0O} \mid \texttt{0o})(\texttt{0}-\texttt{7})^+ \\
&\mid (\texttt{0B} \mid \texttt{0b})(\texttt{0}-\texttt{1})^+
\end{aligned}
$$

정수(*integer*)는 0 에서 9 까지의 숫자들의 모음이다. 기본적으로 정수는 10진수로 사용되며, 아래 표와 같이 특별히 앞에 붙는 문자(prefix)를 사용하여 다른 진수 표현도 가능하다. nML 에서 음수는 기본 값으로 정의되지 않고, 음의 연산자 기호(-)를 이용해서 정의된다.

일반적으로, 양의 기호(+)/음의 기호(-)는 두 가지 용도로 사용된다. 양/음의 값을 나타내거나, 덧셈/뺄셈을 나타내는 연산자로 사용된다. 우선순위(precedence)는 양/음의 값을 나타내는 기호가 덧셈/뺄셈 기호로 사용되는 경우보다 높다.

Prefix	Radix
0x, 0X	16진수(hexadecimal)
0o, 0O	8진수(octal)
0b, 0B	2진수(binary)

실수(Real number literals)

$$
\mathit{real} ::= (\texttt{0}-\texttt{9})^+\langle\texttt{.}(\texttt{0}-\texttt{9})^+\rangle\langle(\texttt{E} \mid \texttt{e})\langle\texttt{-} \mid \texttt{+}\rangle(\texttt{0}-\texttt{9})^+\rangle
$$

실수(*real*)는 정수 부분과 소수(decimal), 그리고 지수(exponent) 부분으로 나뉜다. 정수 부분은 윗 단락에서 설명한 것과 같다. 소수 부분은 소수점으로 정수 부분과 구분이 되며, 1개 이상의 숫자로 구성된다. 지수 부분은 e 나 E 로 시작되고 지수의 부호를 나타내는 +/-가 붙을 수 있다. 소수 부분과 지수 부분은 생략될 수 있으나, 둘 다 생략될 수는 없다. 왜냐하면 둘 다 생략되면 정수와 다름없기 때문이다. 양의 실수나 음의 실수 정의는 정수에서 양/음을 정의한 것과 같이 연산자 기호를 이용하여 나타낸다.

문자(Character literals)

$$
\begin{aligned}
\mathit{char\_literal} &::= \texttt{'}\mathit{printableChar}\texttt{'} \\
&\mid \texttt{'\textbackslash(\textbackslash'} \mid \texttt{b} \mid \texttt{t} \mid \texttt{n} \mid \texttt{r})\texttt{'} \\
&\mid \texttt{'\textbackslash}(\texttt{0}-\texttt{9})(\texttt{0}-\texttt{9})(\texttt{0}-\texttt{9})\texttt{'}
\end{aligned}
$$

문자는 '(single quote)로 묶이는 한 글자를 말한다. 두 ' 의 안쪽에는 한 글자로 사용 가능한 것 중에 ' 나 \ 를 제외한 문자들과, \ 로 시작되는 다음의 특수 문자를 제외한 문자들을 사용할 수 있다.

특수문자	의미
\\	백 슬레시(\)
\'	작은 따옴표(')
\n	라인피드(LF)
\r	리턴(CR)
\t	수평 탭(TAB)
\b	백 스페이스(BS)
ddd	10진수 ASCII 코드 문자 *ddd*

문자열(String literals)

$$
\begin{array}{rcl}
\textit{string} & ::= & \texttt{"}(\textit{printableChar}|\textit{escapecString}|\textit{ignore})^{*}\texttt{"} \\
\textit{escapecString} & ::= & \texttt{\textbackslash}(\texttt{b}|\texttt{t}|\texttt{n}|\texttt{r}|\texttt{\textbackslash}|\texttt{"}|(\texttt{0}-\texttt{9})(\texttt{0}-\texttt{9})(\texttt{0}-\texttt{9})) \\
\textit{ignore} & ::= & \texttt{\textbackslash}(\textit{newline}|\textit{formfeed}|\textit{newline formfeed})(\textit{space}|\textit{tab})^{+}
\end{array}
$$

문자열은 "(double quote)로 묶인 글자들의 모임이다. 두 " 의 안쪽에는 한 글자로 사용 가능한 것 중에 " 나 \ 를 제외한 문자들과, \ 로 시작되는 다음의 특수 문자를 제외한 문자들을 사용할 수 있다.

특수문자	의미
\\	백 슬레시(\)
\"	큰 따옴표(")
\n	라인피드 (LF)
\r	리턴 (CR)
\t	수평 탭(TAB)
\b	백 스페이스(BS)
ddd	10진수 ASCII 코드 *ddd*

앞에 붙는 기호와 새치기 연산자 기호(Prefix and infix symbols)

$$
\begin{array}{rcl}
\textit{prefixid} & ::= & (\texttt{!}\,|\,\texttt{?}\,|\,\texttt{\~{}})\textit{sym}^{*} \\
\textit{infixid} & ::= & (\texttt{\%}\,|\,\texttt{\&}\,|\,\texttt{\$}\,|\,\texttt{\#}\,|\,\texttt{+}\,|\,\texttt{-}\,|\,\texttt{/}\,|\,\texttt{:}\,|\,\texttt{<}\,|\,\texttt{=}\,|\,\texttt{>}\,|\,\texttt{@}\,|\,\texttt{\textbackslash}\,|\,\texttt{`}\,|\,\texttt{\^{}}\,|\,\texttt{|}\,|\,\texttt{*})\textit{sym}^{*} \\
\textit{sym} & ::= & \texttt{!}\,|\,\texttt{\%}\,|\,\texttt{\&}\,|\,\texttt{\$}\,|\,\texttt{\#}\,|\,\texttt{+}\,|\,\texttt{-}\,|\,\texttt{/}\,|\,\texttt{:}\,|\,\texttt{<}\,|\,\texttt{=}\,|\,\texttt{>}\,|\,\texttt{?}\,|\,\texttt{@}\,|\,\texttt{\textbackslash}\,|\,\texttt{\~{}}\,|\,\texttt{`}\,|\,\texttt{\^{}}\,|\,\texttt{|}\,|\,\texttt{*} \\
\textit{opid} & ::= & \textit{prefixid}\,|\,\textit{infixid}
\end{array}
$$

기호 연산자 이름(*opid*)은 기호가 앞에오는(prefix) 연산자 이름과 새치기(infix) 연산자 이름으로 구성된다. 각각의 이름들은 기호가 앞에오는 연산자나 새치기 연산자로 사용된다. 기호 연산자 중에서 특히 -> 나 !! 과 같이 연속된 연산 문자가 하나의 기호 연산자로 사용되기도 한다. 위의 기호 중에 기호 연산자로 사용되지 않은 경우에는 사용자가 연산자로 정의해서 사용할 수 있다.

키워드(Keywords)와 예약어(Reserved Words)

아래에 나타나는 단어나 기호들은 키워드나 예약어로서 사용된다. 따라서 다른 곳에서는 이 단어나 기호들을 사용할 수 없다.

```
and             as              case            do              else
end             exception       fn              for             fun
functor         handle          if              in              include
let             local           of              open            raise
rec             ref             sig             signature       struct
structure       then            type            val             where
while


:=    !    ->    <-    =>    |    :    ;     {     }     [     ]
[|    |]    ,     (     )     .


array           bool            char            exn             int
list            real            ref             string          unit


andalso         orelse
false           False           nil             Nil             not
true            True


+    -    *    /    %   **    ++    --    +=    -=    *=    /=    <<
>>  &&   ||   =    <>   <     >     <=    >=    ::    @     ^
```

2 값(Values)

이 절에서는 nML 에서 사용할 수 있는 여러 종류의 값에 대하여 설명한다. 값에는 기본 타입과 관련된 기본 값, 리스트를 포함하는 데이타 구성자에 의해 생성된 값, 예외상황 값, 레코드 값, 배열, 튜플 값, 메모리 주소형 값, 함수 값과 기본 연산자 값들이 있다.

2.1 기본 값(Base values)

정수(Integer numbers)

정수 값은 -2^{30} 에서 $2^{30}-1$ 의 값을 가질 수 있다. 즉, -1073741824 에서 1073741823 까지의 숫자가 정수 값이 될 수 있다.

실수**(Real numbers)**

실수 값은 앞절에서 설명한 것과 같이 소수점이나 지수로 구성된 숫자이다. 현재의 구현은 53 bits의 밑과 (-1022) - (1023) 의 사이의 지수로 구성되는 IEEE 754 표준에 맞춰져 있다.

문자**(Characters)**

문자의 값은 0에서 255 사이의 8-bit 정수 값으로 표현된다. 0 에서 127 사이의 문자 코드는 ASCII 표준를 따라 사용되고, 나머지는 ISO 8859-1 표준에 따라 해석이 된다. nML 에서는 한글 문자를 사용할 수 있다. nML에서 사용되는 한글 문자는 KSX1001(a.k.a. KSC5601 이나 eur-kr)과 KSX1005-1(a.k.a. KSC5700, unicode 혹은 ISO/IEC10646-1)를 따르고 있다.

문자열**(Character strings)**

문자열 값은 문자들의 한정된 모임이다. 현재의 구현은 $2^{24} - 6$ 개의 문자를 포함하는 문자열 값을 지원할 수 있다(16777210 개).

2.2 데이타 구성자에 의해 생성된 값(Values from constructors)

데이타 구성자에 의해서 생성되는 값들은 크게 두 가지가 있다. 첫번째는 인수없이 생성되는 붙박이(built-in) 상수 값이고 두번째는 여러 개의 인수를 이용해서 생성되는 값이다.

다음의 표에 있는 값들은 다시 정의될 수 없이 붙박이 상수 값으로 사용되는 값들이다.

상수	의미하는 값	타입
`false`, `False`	거짓 값	`bool`
`true`, `True`	참 값	`bool`
`()`	"유닛" 값	`unit`
`[]`, `nil`, `Nil`	빈 리스트	`list`

2.3 리스트(List)

리스트(list)는 붙박이(built-in)로 정의된 데이타 구성자에 의해 생성된 값이다. 리스트는 같은 타입을 가지는 값들의 모임으로 다른 모임 값들(튜플, 배열, 레코드)과 달리 크기가 정해져 있지 않다. 리스트는 일반적으로 $[v_1, \ldots, v_n]$ 로 쓰이는데, v_1 에서 v_n 까지의 값은 같은 타입을 가진다.

nML 에서 제공하는 여러가지 연산자에 의해 리스트가 다루어질 수 있는데, 이러한 연산들에 대해서는 6.3절에서 다루고 있다.

2.4 예외상황(Exceptions)

예외상황 값은 예외상황 타입으로 정의되어 사용되는 값을 말한다. 예외상황 값이 타입과 함께 선언되어서 사용되는 방법은 7.3절에서 설명한다.

2.5 레코드(Records)

레코드[2]는 각 원소에 이름을 붙여서 묶은 것이다. 레코드 값은 {lab_1=v_1, ..., lab_n=v_n} 과 같이 쓰이며, 각 v_i 들은 레코드 레이블(label)인 lab_i 라는 이름을 갖게 된다. 레코드를 다루는 연산자에 관해서는 6.3절에서 설명한다.

2.6 배열(Arrays)

배열 값은 [| v_1, ..., v_n |] 으로 쓰이며, 같은 타입의 값을 가진다. nML 에서 배열에 대한 연산은 메모리 영역 위에서 직접 이루어지므로, 배열의 참조나 갱신은 메모리 참조나 갱신으로 구현된다. 배열을 위한 메모리 참조 및 갱신 연산자는 6.3절에서 소개된다.

2.7 튜플(Tuples)

튜플 값은 (v_1, ..., v_n) 으로 쓰이며, v_1에서 v_n 까지의 값을 가지는 n-튜플을 의미한다(튜플은 이름이 숫자로 되어 있는 레코드로 정의된다. 튜플은 내부적으로는 레코드 값의 설탕 구조로 표현된다). 튜플을 다루는 방법은 6.3절에서 소개된다.

2.8 메모리 주소형 값(Reference)

메모리 주소형 값은 메모리 내의 주소를 값으로 가진다. 이 주소에는 실제 데이타가 들어있는 다른 주소가 저장되어 있다. 메모리 주소형 값은 `ref` v 의 형태를 갖는데, 이 값은 v 값을 저장한 메모리 주소이다. nML 에서 메모리 주소를 가지고 연산하는 것은 금지된다.

2.9 함수(Functions)

함수 값은 값에서 값으로의 매핑이다. 함수 정의(6.1절 참조)를 통해서 함수 값을 정의할 수 있다. nML 에서 함수는 값으로 취급되기 때문에, 프로그램의 인자나 결과로 사용될 수 있다.

2.10 기본 연산자(Basic operators)

nML 에서 기본 연산자는 값으로 정의된다. 하지만, 일반적으로 기본 연산자는 프로그램식으로 사용되기 때문에 기본 연산자에 대한 설명은 프로그램식(6.4절)에서 하도록 한다.

[2]nML의 정의에 따르면 타입 선언없이 레코드를 값으로 사용할 수 있으나, ocaml을 기반으로 하는 현재의 nML 구현에서는 타입 정의 없이 레코드를 사용할 수 없다. 왜냐하면 ocaml에서 타입 정의없이 레코드를 사용할 수 없기 때문이다.

3 이름(Names)

$$
\begin{array}{rcl}
\mathit{varid} & ::= & \mathit{lid} \\
\mathit{opid} & ::= & \mathit{prefixid} \mid \mathit{infixid} \\
\mathit{tyid} & ::= & \mathit{lid} \\
\mathit{tyvar} & ::= & \texttt{'}\mathit{lid} \\
\mathit{conid} & ::= & \mathit{uid} \\
\mathit{lab} & ::= & (0-9)(0-9)^* \mid \mathit{lid} \\
\mathit{strid} & ::= & \mathit{uid} \\
\mathit{sigid} & ::= & \mathit{uid} \\
\mathit{fctid} & ::= & \mathit{uid} \\
\mathit{varlongid} & ::= & \mathit{varid} \mid \mathit{strid}\,.\,\mathit{varlongid} \\
\mathit{oplongid} & ::= & \mathit{opid} \mid \mathit{strid}\,.\,\mathit{oplongid} \\
\mathit{tylongid} & ::= & \mathit{tyid} \mid \mathit{strid}\,.\,\mathit{tylongid} \\
\mathit{conlongid} & ::= & \mathit{conid} \mid \mathit{strid}\,.\,\mathit{conlongid} \\
\mathit{strlongid} & ::= & \mathit{strid} \mid \mathit{strid}\,.\,\mathit{strlongid}
\end{array}
$$

여러 종류의 언어 요소들을 구분하는 이름들은 다음과 같다.

변수 이름 **:** 변수 이름은 값을 가리키는 이름이며, 다음과 같이 두 가지 변수 이름이 사용된다.

- 문자와 숫자로 된 변수 이름(*varid*) : 소문자와 한글로 시작되는 단어들이 사용될 수 있다.
- 기호로 된 변수 이름 : (*opid*) 의 형태를 갖는 변수 이름으로 *opid*에는 연산자 이름이 온다. () 를 포함하여 정의된 이와같은 이름은 연산자가 아니라 변수 이름으로 사용된다.

타입 이름(*tyid*) **:** 특정 타입을 나타내는 이름으로 소문자와 한글로 시작되는 단어들이 가능하다.

타입 변수 이름(*tyvar*) **:** 다변형(polymorphic) 타입을 나타낼 때 쓰이는 타입 변수의 이름으로, ‘’’ 가 앞에 붙어 있는 소문자와 한글로 시작되는 단어들이 가능하다.

ex) `'a list` : 모든 가능한 리스트를 나타내는 타입

데이타 구성자 이름(*conid*) **:** 사용자 정의 상수를 생성할 수 있는 데이타 구성자를 나타내는 이름으로, 대문자나 _ 로 시작되는 이름만이 가능하다.

레이블 이름(*lab*) **:** 레코드의 각 필드(field)를 나타내는 이름으로, 소문자와 한글로 시작되는 단어들과 부호 없는 정수가 가능하다.

모듈 이름 (*strid*, *sigid*, *fctid*) **:** 모듈(`structure`), 모듈 타입(`signature`), 모듈 함수(`functor`)들을 나타내는 이름으로 대문자나 _ 로 시작되는 이름만이 가능하다.

연산자 이름(*opid*) **:** 연산자를 나타내는 이름으로 일부 예약어를 제외한 기호 문자로 이루어진 단어이다. *opid* 는 위에서 언급한 것과 같이 변수 이름으로도 사용될 수 있다.

긴 이름(*varlongid, oplongid, tylongid, conlongid, strlongid*) : 긴 이름은 모듈, 모듈 타입, 모듈 함수 등의 안쪽에 정의된 이름을 가리키는 이름이다.

ex) `Mod.operand` : 이름이 `Mod` 인 모듈 속에 정의된 이름 `operand` 를 말한다.

사용되는 각 이름들에 대해 사용할 수 있는 알파벳 종류를 정리하면 다음과 같다.

이름 종류	이름의 첫 단어에 대한 조건
문자열 변수 이름	소문자나 한글
타입 이름	소문자나 한글
데이타 구성자 이름	대문자나 _
레이블 이름	소문자나 한글 혹은 부호없는 정수
모듈 이름	대문자나 _

4 타입식(Type expressions)

ty	::=	*tyvar*	
	|	`{`*tyrow*`}`	
	|	*tyseq tylongid*	
	|	ty_1 `->` ty_2	
	|	`(`*ty*`)`	
	†	ty_1 `*` ty_2	
tyrow	::=	*lab* `:` *ty* ⟨`,`*tyrow*⟩	
tyseq	::=	*ty*	
	|		empty sequence
	|	`(`$ty_1, \cdots, ty_k$`)`	$k \geq 1$
tyvarseq	::=	*tyvar*	
	|		empty sequence
	|	`(`$tyvar_1, \cdots, tyvar_k$`)`	$k \geq 1$
tyvar	::=	`'`*lid*	

타입식은 데이타 타입 정의에서 사용되고, 패턴이나 프로그램식에서 타입을 제약할 때 사용된다.

다음 표는 타입식에서 사용되는 연산자들의 우선 순위를 나타낸다. 표에서 위쪽에 오는 연산자들일수록 높은 우선 순위를 가지고 있다. 표의 가장 위쪽에 나오는 타입 구성자 적용하기 (Type constructor application)는 아래에 나오는 타입 구성자 (Constructed types) 에 인자로 타입을 넘기는 연산을 말한다.

연산자	방향성
타입 구성자 적용하기 (Type constructor application)	–
`*` (튜플 타입 (Tuple type))	–
`->` (함수 타입 (Function type))	오른쪽
`where` (모듈 타입 제약식 (Signature constraint))	왼쪽

타입 변수(Type variables)

타입식 *'lid*는 *lid*를 이름으로 하는 타입 변수를 의미한다.

레코드 타입(Record types)

타입식 {*tyrow*}는 내부에 이름이 있는 원소를 가지고 있는 레코드 타입을 나타낸다. 내부의 원소는 *tyrow* 와 같이 여러 개의 레이블과 타입 쌍으로 나타난다.

타입 구성자(Constructed types)

타입 구성자는 인자로 타입을 받아서 결과 타입을 만드는 일종의 타입을 만드는 함수이다. 인자(arguments)를 가지고 있지 않은 타입 구성자는 그 자체가 타입이 된다. *ty tyid* 는 타입 구성자 *tyid* 에 하나의 인자인 *ty*를 넘긴 결과 타입을 의미한다. $(ty_1, ..., ty_k)$ *tyid* 는 k 개의 인자를 가지는 타입 구성자 *tyid* 에 k 개의 인자를 넘긴 결과 타입을 의미한다.

함수 타입(Function types)

ty_1 -> ty_2 는 ty_1 타입을 갖는 인자를 받아 ty_2 타입의 결과를 주는 함수 타입을 나타낸다.

괄호가 있는 타입(Parenthesized types)

타입식 (*ty*) 은 *ty* 과 같은 타입을 나타낸다.

튜플 타입(Tuple types)

타입식 ty_1* ... * ty_n 은 각각의 원소들이 $ty_1, ..., ty_n$ 의 타입에 속하는 튜플 타입을 말한다. 튜플 타입은 레코드 타입의 설탕 구조이다.

패턴 구성자	방향성
`ref`	–
데이타 구성자 패턴	왼쪽
커리형 패턴	–
`::`	오른쪽
`|`	–
`:`	–
`as`	–

표 1.1: 패턴의 연산자 사이의 우선 순위

5 패턴(Pattern)

pat	::=	`_`	wild pattern
	|	*integer* | *string* | *character*	constant pattern
	|	`{` *patrow* `}` | `{}`	record pattern
	|	`[|`*pat* (`,` *pat*)*`|]` | `[||]`	array pattern
	|	`(` *pat* `)`	
	|	`ref` *pat*	
	|	*conlongid* ⟨*pat*⟩	datacon pattern
	|	*varid*	pattern variable
	|	`(` *opid* `)`	operator pattern variable
	|	*pat* `:` *ty*	pattern with type
	|	(*varid* | `(` *opid* `)`) ⟨`:` *ty*⟩ `as` *pat*	as pattern
	|	pat_1 `|` pat_2	or pattern
	|	*pat* `::` *pat*	cons pattern
	†	(*pat* (`,` *pat*)$^+$)	tuple pattern
	†	`[`*pat* (`,` *pat*)*`]` † `[]`	list pattern
	†	`()`	unit pattern

patrow	::=	`...`	
	|	*lab* `=` *pat* ⟨`,` *patrow*⟩	
	†	*lid*⟨`,` *patrow*⟩	

패턴은 주어진 값을 데이타 구조에 맞추어 선택하거나, 이름을 데이타 구조를 가진 값과 묶을 때 사용되는 틀이다. 이 선택을 위한 연산을 패턴 맞추기(pattern matching)라고 부른다: 패턴 맞추기의 결과는 "주어진 값이 이 패턴과 맞지 않는다" 이거나, "주어진 값이 패턴에 맞고 해당하는 이름과 값이 묶인다(binding)" 의 두 가지이다.

표 1.1는 상대적인 우선 순위를 나타낸다. 가장 높은 우선 순위를 가지고 있는 생성자가 제일 위에 온다.

임의 패턴(Wild pattern)

패턴 _ 은 어떠한 값과도 어울리지만, 이름과는 묶이지(binding) 않는다.

상수 패턴(Constant patterns)

상수로 되어있는 패턴 (*integer*|*string*|*character*|`()`)은 값이 주어진 상수와 같은지 비교한다.

레코드 패턴(Record patterns)

패턴 `{}` 는 빈 레코드값이 들어 오면 패턴 맞추기가 성립한다[3]. 패턴 $\{lab_1{=}pat_1, ..., lab_n{=}pat_n\}$ 은 비교되는 값이 레코드 값을 나타낼 경우에 패턴 비교가 수행된다. 그 레코드가 갖는 레이블이 패턴의 레이블 $lab_i (i = 1, ..., n)$과 일치하는지 검사한다. 그리고 레이블이 같으면 각 레이블에 정의된 값과 위의 패턴 $pat_i (i = 1, ..., n)$이 일치하는지 비교한다.

레코드 패턴 맞추기에는 특별히 임의 패턴을 위해 `...` 이 사용된다[4]. 패턴 `...` 은 레코드의 나머지 부분에 대해 비교하지 않을 때 쓰인다. 즉, 패턴 $\{lab_1{=}pat_1, \texttt{...}\}$ 은 비교되는 값이 레코드값이고, 그 레코드값 내의 레이블 lab_1 이 가지는 값이 pat_1 과 패턴이 맞으면 레코드 내의 다른 레이블들과 관계없이 패턴 맞추기가 성공적으로 수행된다.

배열 패턴(Array patterns)

패턴 `[||]` 는 빈 배열값이 들어오는 경우 패턴 맞추기가 이루어진다. 패턴 $[|pat_1, ..., pat_n|]$ 은 배열 값과 비교된다. 우선, 원소의 갯수가 n 인 배열이 이 패턴과 맞추기가 이루어질 수 있다. 배열의 원소들이 $v_i (i = 1, ..., n)$의 값을 갖는다면, 각각의 i 에 대해 각 v_i 들은 $pat_i (i = 1, ..., n)$과 패턴 맞추기가 일어난다.

괄호가 있는 패턴(Parenthesized patterns)

패턴의 앞뒤에 괄호가 사용되는 것은 우선 순위를 위한 것이다. 패턴 (*pat*)은 *pat* 이 패턴 맞추기에 사용된다.

메모리 주소 패턴(Reference patterns)[5]

패턴 `ref` *pat* 은 메모리 주소 값과 *pat* 이 일치하는지 비교한다.

데이타 구성자 패턴(Data Constructor patterns)

패턴 *conlongid* ⟨*pat*⟩ 은 먼저 주어진 값의 데이타 구성자가 *conlongid*인지 비교한다. 만일 데이타 구성자 *conlongid* 가 인자를 가지지 않으면 패턴 맞추기는 여기서 끝난다. 하지만, 인자를 가지면 패턴의 인자인 ⟨*pat*⟩ 과 인자 값 간의 패턴 맞추기가 수행된다.

[3] `()`는 `{}`의 설탕구조이다.

[4] 레이블과 값의 쌍이 없는 패턴 `{...}`은 현재 nML에서 사용할 수 없다.

[5] 메모리 주소 패턴은 ocaml 을 기반으로 하는 현재의 nML 에서는 사용할 수 없다. 왜냐하면 ocaml에서 메모리 주소 패턴을 지원하지 않기 때문이다.

변수 패턴(Variable patterns)

패턴 *varid*는 어떤 값과도 맞추기가 이루어진다. 임의 패턴 _ 이 변수의 값정의가 일어나지 않는 것과는 달리, 변수 패턴은 변수 이름 *varid* 에 대한 값정의가 패턴 맞추기 때 발생한다. 즉, 비교되는 값은 변수 이름 *varid* 의 값으로 정의된다. 하나의 패턴 내부에는 같은 변수가 여러번 나타날 수 없다.

연산자 패턴 변수(Operator pattern variables)

패턴 (*opid*)은 연산자를 이용하여 패턴 변수를 정의하는 연산자 패턴 변수이다. 이것은 변수 패턴과 마찬가지로 패턴 맞추기를 수행하는 값이 변수 이름 (*opid*) 에 묶여서 사용된다. 하나의 패턴 내부에는 같은 변수가 여러번 나타날 수 없다.

타입 제약식이 있는 패턴

패턴 뒤에는 타입 제약식이 붙을 수 있다. $pat : ty$ 와 같은 패턴은 이 패턴과 비교되는 값이 ty과 어울리는 적합한(compatible) 타입을 갖는지를 검사할 때 사용된다.

'As' 패턴('As' pattern)

패턴 *varid* as *pat* 은 *pat* 과 값을 비교한다. 패턴 맞추기가 성공적으로 이뤄진 경우, 맞추어진 값은 변수 이름 *varid*에 묶여서 사용된다.

무더기 패턴('Or' pattern)

패턴 pat_1 | pat_2 는 두 패턴에 대한 논리적 'or'를 의미한다. 즉, 이 패턴에 맞추어진 값은 pat_1과 일치하거나, pat_2과 일치한다. 이때 두 하위 패턴인 pat_1과 pat_2에서 사용하는 변수 이름은 같아야 한다. 예를 들어, 패턴 pat_1 에 사용된 변수가 세 가지이고 그 이름이 'x, y, z' 라면, 패턴 pat_2 에도 이름이 똑같은 변수 'x, y, z' 가 모두 사용되고 있어야 한다.

리스트 패턴(List pattern)

패턴 [] 은 비어 있는 리스트와 맞추어 진다. 패턴 pat_1 :: pat_2는 비어있지 않은 리스트와 비교된다. 비교되는 리스트가 v_1 :: v_2 인 경우, 리스트의 머리(head)인 v_1 은 패턴의 머리 pat_1 과 일치하는지 비교되고, v_2는 패턴의 나머지(tail)인 pat_2와 비교된다. 패턴 [pat_1 ,..., pat_n] 은 리스트의 각 원소들이 pat_1, ..., pat_n과 일치하는 길이 n 인 리스트와 맞추어 진다. 이 패턴은 pat_1 :: ... :: pat_n :: [] 과 같다.

튜플 패턴(Tuple patterns)

패턴 (pat_1,...,pat_n) 은 튜플 값과 패턴 맞추기가 이루어진다. n개의 원소를 갖는 튜플 값에 대해, 튜플의 각 원소들이 순서대로 pat_1 에서 pat_n까지의 패턴과 일치하는 경우 전체 패턴 맞추기가 성공적으로 이루어지게 된다. 즉, 이 패턴은 각 pat_i 들에 대하여 일치하는 v_i 들을 원소로 갖는 튜플 ($v_1, ..., v_n$) 과 맞추어 진다(이때, $i = 1, ..., n$).

6 프로그램식(Expressions)

e	::=	*const*	
	|	*varlongid*	
	|	*conlongid*	
	|	`(` *oplongid* `)`	
	|	`{` ⟨*labrow*⟩ `}`	record
	|	e_1 `.` *lab*	record field
	|	`[|` ⟨*elmtrow*⟩ `|]`	array
	|	e_1 `.` `[` e_2 `]`	array field
	|	e_1 `.` `[` e_2 `]` `<-` e_3	array update
	|	`let` *valdec* `in` e `end`	binding
	|	`(` e `)`	
	|	e_1 e_2	application
	|	e `:` *ty*	
	|	e `handle` *match*	
	|	`raise` e	
	|	`fn` *match*	function
	†	`fn` *curmatch*	curried function
	|	e_1 `:=` e_2	assignment
	|	`!` e	dereference
	|	`ref` e	reference
	†	*prefixid* e	prefix application
	†	e_1 *infixid* e_2	infix application
	†	e_1 `{` *lab* `<-` e_2 `}`	new record
	†	e_1 `;` e_2	sequence
	†	`case` e `of` *match*	
	†	`if` e_1 `then` e_2 `else` e_3	
	†	`if` e_1 `then` e_2	
	†	`(` e `,` *elmtrow* `)`	tuple
	†	`[` ⟨*elmtrow*⟩ `]`	list
	†	`while` e_1 `do` e_2 `end`	
	†	`for` *varid* `=` e_1`;` e_2`;` e_3 `do` e_4 `end`	
	†	`()`	

const	::=	*integer* | *real* | *string* | *character*
curmatch	::=	pat^+ `=>` e ⟨`|` *curmatch*⟩
match	::=	*pat* `=>` e ⟨`|` *match*⟩
labrow	::=	*lab* `=` e ⟨`,` *labrow*⟩
elmtrow	::=	e ⟨`,` *elmtrow*⟩

$$
\begin{array}{rcl}
\mathit{valdec} & ::= & \texttt{val}\ \mathit{tyvarseq}\ \mathit{valbind} \\
& \dagger & \texttt{fun}\ \mathit{funbind} \\
& | & \mathit{valdec}\ \langle \texttt{;} \rangle\ \mathit{valdec}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{funrule} & ::= & (\mathit{varid} \mid \texttt{(}\ \mathit{opid}\ \texttt{)})\ \mathit{pat}^{+} \langle \texttt{:}\ \mathit{ty} \rangle\ \texttt{=}\ e \\
\mathit{funbind} & ::= & \mathit{funrule}\ (\texttt{|}\mathit{funrule})^{*}\ \langle \texttt{and}\ \mathit{funbind} \rangle
\end{array}
$$

아래의 표는 연산자들의 상대적인 우선 순위와 방향성을 나타낸다. 표의 위쪽에 있을수록 높은 우선 순위를 가진다. 새치기 연산자 기호나 앞에 붙는 연산자 기호들에 대해서 *··· 과 같은 형식으로 표기 하였는데, 이것은 *로 시작되는 모든 기호들을 뜻한다.

프로그램식 구성자	방향성
`ref`, *prefixop*, `++` (postfix), `--` (postfix)	오른쪽
`.`, `.[`	왼쪽
새 레코드	왼쪽
데이타 구성자 적용하기	왼쪽
함수 호출식	왼쪽
`+` (prefix), `-` (prefix)	오른쪽
`**`···	오른쪽
`*`···, `/`···, `%`···	왼쪽
`+`···, `-`···	왼쪽
`::`	오른쪽
`@`··, `^`···	오른쪽
다른 새치기 연산자들	왼쪽
`not`	오른쪽
`andalso`, `&&`	오른쪽
`orelse`, `||`	오른쪽
`:=`, `+=`, `-=`, `*=`, `/=`, `<-`	오른쪽
`,`	왼쪽
`:`	왼쪽
`raise`	오른쪽
`if`	오른쪽
`;`	오른쪽
`case`, `fn`, `handle`	오른쪽

6.1 기본 프로그램식 (Basic expressions)

긴 이름(Long identifiers)

긴 이름은 다른 모듈에 정의된 값을 쓰고자 할 때 사용한다. 모듈 안에 모듈이 정의될 수 있으므로 여러 개의 모듈 이름을 사용해야 할 경우도 있다. 모듈 내의 이름들을 외부에서 사용하고자 할 경우에는 모듈 이름 뒤에 '.'을 붙여서 표기한다. 예를 들어, 모듈 M 내에 정의된 이름 *varid* 를

모듈 외부에서 사용할 때에는, $M.varid$ 과 같이 사용한다. 모듈 타입(*signature*)에 명시되어 있는 이름들은 모두 이런 긴 이름을 이용하여 모듈 외부에서 사용될 수 있다. 단지 다른 모듈에서 정의된 기호로 된 이름(*opid*)을 사용할 때에는 전체의 긴 이름을 ()로 둘러 싸야 한다. 예를 들어, 어떤 모듈 M 에 정의된 이름 (*opid*)이 모듈 M의 외부에서 사용된다면, ($M.opid$) 과 같이 사용되어야 한다($M.(opid)$ 는 옳치 못하다).

Boolean 연산자

프로그램식 $expr_1$ `andalso` $expr_2$ 은 두 프로그램식이 모두 `true` 값을 결과로 가질 때 `true` 값을 갖고, 다른 모든 경우에는 `false` 값을 결과로 갖는다. 만일 첫번째 프로그램식의 결과가 `false`가 나오면, 두번째 프로그램 식을 수행하지 않고 전체 결과를 `false`라고 낸다. 즉, $expr_1$ `andalso` $expr_2$ 프로그램식은 아래의 프로그램처럼 동작한다.

`if` $expr_1$ `then` $expr_2$ `else false`

키워드 `andalso` 는 연산자 기호 `&&` 와 같다.

프로그램식 $expr_1$ `orelse` $expr_2$ 은 두 프로그램식이 모두 `false` 값을 결과로 가질 때 `false` 값을 갖고, 다른 모든 경우에는 `true` 값을 결과로 갖는다. 만일 첫번째 프로그램식의 결과가 `true`가 나오면, 두번째 프로그램 식을 수행하지 않고 전체 결과를 `true`라고 낸다. 즉, $expr_1$ `orelse` $expr_2$ 프로그램식은 아래의 프로그램처럼 동작한다.

`if` $expr_1$ `then true else` $expr_2$

키워드 `orelse` 는 연산자 기호 `||` 와 같다.

함수 호출식**(Function application)**

함수 호출식은 함수로 계산되는 프로그램식 뒤에 인자로 계산되는 프로그램식이 오는 형태를 가지고 있다. 이 식의 계산 순서는 먼저 앞의 식을 함수로 계산하고 그 다음에 뒤의 식을 계산한다. 그리고, 뒤의 식으로부터 나온 값을 앞선 식의 인자로 넘겨서 함수를 계산한다.

함수식이 먼저 나오는 것이 함수 호출식의 일반적인 형태이지만, 기본 연산자로 정의된 몇 가지 연산자 기호(예를 들어, `+`, `-` 등)들은 인자들의 가운데에 오거나, 인자들의 뒤에 오는 경우가 있다. 이런 기본 연산자들은 6.4절에서 자세히 다루고 있다.

함수 정의식**(Function definition)**

함수의 정의에는 두 가지 형태가 있다. 첫번째 형태는 아래와 같이 하나의 패턴 인자를 가지는 것이다. 이 형태를 기반으로 해서 두 번째 형태의 함수 정의와 키워드 `fun`를 이용하는 함수 정의(7.1절)가 모두 설탕 구조로 정의되게 된다.

$$\begin{array}{lll} \texttt{fn} & pat_1 & \texttt{=>} \quad expr_1 \\ \mid & \ldots & \\ \mid & pat_n & \texttt{=>} \quad expr_n \end{array}$$

이 프로그램식은 하나의 인자를 갖는 함수 정의식이다. 이 함수가 어떤 값 v를 인자로 해서 호출되면, 그 값은 pat_1 에서 pat_n 까지의 각 패턴에 일치하는지 검사된다. 각 패턴 중 하나가 일치

하면, 즉 v가 어떤 한 패턴 pat_i (어떤 i에 대해서) 에 대해서 일치하면, 해당하는 함수 몸체(body) $expr_i$ 이 계산된다. 이때, 패턴 pat_i 의 패턴 맞추기 때에 새로 정의된 이름들은 $expr_i$을 계산할 때에만 사용된다. $expr_i$ 의 계산 결과는 전체 함수 호출식의 결과값이 된다.

만일 여러 개의 패턴이 일치하면, 그 중에서 제일 먼저 일치(함수 정의 상, 제일 위에 있는)하는 패턴이 선택된다. 어떠한 패턴도 인자 값에 대해서 일치하지 않으면 예외상황 `Match` 가 발생한다.

다른 형태의 함수 정의는 아래와 같은 형태이다.

`fn` pat_1 ... pat_n `=>` $expr$

이 프로그램식은 다음의 프로그램식과 같다.

`fn` pat_1 `=>` ... `=>` `fn` pat_n `=>` $expr$

이 형태의 함수 프로그램식은 n 개의 인자를 가지는 커리형(curried) 함수로 계산이 된다: 즉, 이 함수가 $v_1, ..., v_n$ 의 n 개의 값을 인자로 호출되면, 각 v_i 들은 pat_i 에 대해서 패턴 맞추기가 일어난다(단, $i = 1, ..., n$). 패턴 맞추기가 성공적으로 이루어지면, 각 패턴 맞추기에 의해서 새롭게 정의된 이름들이 $expr$ 의 수행 시에 사용된다. 최종적으로 계산된 $expr$ 의 값이 전체 함수 호출식의 결과가 된다. 만일 하나의 패턴이라도 패턴 맞추기가 이루어지지 않으면 예외상황 `Match`가 발생한다.

지역적인 정의를 가지는 프로그램식**(Local definitions)**

지역적인 정의는 `let` $valdec$ `in` e `end` 을 통해서 이루어 진다. 이 프로그램식은 $valdec$ 을 통해서 이름이 붙여진 값들의 유효 범위를 프로그램식 e의 내부로 제한한다. 여기서 값의 이름을 정하는 $valdec$ 부분의 정의는 선언 부분에서 사용할 수 있는 정의와 동일하다. 따라서, $valdec$ 은 선언에 대한 절인 7.1절에 설명된 것을 따른다.

지역적인 정의를 가지는 프로그램식은 이밖에도, 선언(7절)에서 사용되는 `local` dec_1 `in` dec_2 `end` 이 있다. 이 프로그램식에 대한 설명은 7.4절에서 다루고 있다.

6.2 실행 순서 구조(Control structures)

나열식**(Sequences)**

프로그램식 $expr_1$`;`$expr_2$ 는 $expr_1$ 을 먼저 계산하고, $expr_2$ 를 그 다음에 수행하여 $expr_2$ 의 결과를 가지는 프로그램식이다. 나열식을 사용할 때 주의할 점은 조건문의 안쪽에서는 나열식을 괄호로 묶어야 한다는 점이다. 나열식 기호 ';' 의 우선순위가 `if then else` 보다는 낮기 때문이다.

조건식**(Conditional)**

`if` 를 사용하는 조건식에는 두 가지 형태가 있다. 첫번째는 `if` $expr_1$ `then` $expr_2$ `else` $expr_3$ 의 프로그램 식이다. 이 프로그램식은 우선 $expr_1$ 을 계산하여, 그 값이 `true`가 나올 경우 $expr_2$ 의 값을 결과로 하고, `false`가 나올 경우 $expr_3$ 의 값을 결과로 한다.

두번째 형태의 조건식은 `else` 부분을 생략한 것이다. 프로그램식 `if` $expr_1$ `then` $expr_2$ 이 그것인데, 이 경우 $expr_1$ 의 결과가 `true`가 나올 경우 $expr_2$ 의 값을 계산하여 결과로 낸다. 이때, $expr_2$은 `unit` 타입의 결과값을 가져야 한다. $expr_1$ 의 결과가 `true`가 아닐 경우에는 `()` 을 결과로 낸다.

선택식**(Case expressions)**

프로그램식

$$\begin{array}{lll} \texttt{case} & expr & \\ \quad \texttt{of} & pat_1 & \texttt{=>} \quad expr_1 \\ \quad\quad | & ... & \\ \quad\quad | & pat_n & \texttt{=>} \quad expr_n \end{array}$$

은 $expr$ 의 값을 각 패턴 $pat_1, ..., pat_n$ 과 비교하여 `=>` 뒤의 프로그램식을 선택적으로 계산하는 프로그램식이다.

만일 패턴 pat_i 가 일치한다면, 해당하는 프로그램식 $expr_i$ 가 계산되고, 그 결과값이 `case` 문 전체의 값이 된다. 일치된 패턴에 의한 새 이름들은 관련된 프로그램식의 수행시에만 사용된다. 여러 개의 패턴이 일치하는 경우에는 제일 먼저 일치(가장 위에 있는)하는 패턴이 선택된다. 일치하는 패턴이 없는 경우에는 예외상황 `Match`가 발생한다.

반복문**(Loops)**

프로그램식 `while` $expr_1$ `do` $expr_2$ `end` 은 $expr_1$ 이 `true` 가 나올 때까지 반복적으로 $expr_2$ 를 수행한다. 반복문의 조건식 $expr_1$ 은 반복문 전체를 되풀이 수행할 때마다 수행되고 검사된다. 전체 프로그램식 `while` ... `end` 의 결과는 반드시 `()`이 되어야 한다.

프로그램식 `for` $varid$ `=` $expr_1$`;` $expr_2$`;` $expr_3$ `do` $expr_4$ `end` 은 지역 변수를 사용한 반복문이다. 프로그램식 $expr_1$ 의 값은 지역 변수인 $varid$ 의 초기값을 결정한다. $expr_2$ 는 반복문의 조건식으로써 이 프로그램식의 계산 결과가 `true` 가 되면 `for` 반복문을 계속 수행하고, `false`가 되면 `for` 반복문을 종료한다. 일반적으로 조건식 $expr_2$ 의 결과는 지역 변수 $varid$에 의존하게 된다. $expr_3$ 은 다음 반복 수행시 $varid$에 묶이는 값을 의미한다.

`for` $varid$ `=` $expr_1$`;` $expr_2$`;` $expr_3$ `do` $expr_4$ `end` 문의 수행은 다음과 같이 이루어진다. 우선 $expr_1$ 의 값을 계산하여 $varid$ 에 묶어 두고, 조건식 $expr_2$를 계산한다. 만일 $expr_2$ 의 수행 결과가 `false`가 되면 `for` 반복문을 종료한다. 이때 프로그램식의 결과는 `()` 이어야 한다. 만일 $expr_2$ 의 수행 결과가 `true` 가 되면 $expr_4$ 를 계산한다. 그리고 $expr_3$ 의 값을 계산하여 그 값을 다시 $varid$ 에 묶은 뒤 반복문의 수행을 계속한다.

예를 들어, 다음과 같은 C 언어의 반복문이 있을 때,

`for (` i`=0;` i`<10;` i`=`i`+1) {` $expr$`;}`

같은 역할을 하는 nML 언어의 반복문은 다음과 같다.

`for` i`=0;` i`<10;` i`+1 do` $expr$ `end`

C 언어와 비교할 때 주의할 점은 nML에서 반복문의 지역 변수 i 의 유효 범위(scope)는 반복문 내부로 한정되며, i+1 의 자리에 i++ 이 올 수 없다는 점이다(nML 에서 i++ 은 메모리 반응(memory-refernce) 을 가지는 프로그램식이다; 6.4절 참조).

`for` ... `end` 반복문 역시 설탕 구조로써 `let` ... `in` ... `end` 를 사용하여 다음과 같이 표현할 수 있다.

```
for varid = e1; e2; e3 do e4 end
  ≡  let val rec f = fn varid =>
                 if e2 then (e4; f(e3)) else ()
     in f e1 end
```

예외상황의 발생과 처리**(Exception raise and handling)**

프로그램식

`raise` *expr*

은 예외상황을 발생시킨다. 이때 발생하는 예외상황은 이전에 `exception` 타입으로 미리 선언되어 있어야 한다. 즉, `raise` 뒤에 나타나는 *expr*은 미리 선언되어 있던 `exception` 타입의 데이타 구성자로서, 발생한 예외상황의 이름이 된다. 이 이름을 보고 해당하는 예외상황 처리기가 발생한 예외상황을 처리하게 된다.

프로그램식

expr `handle` pat_1 `=>` $expr_1$
`|` ...
`|` pat_n `=>` $expr_n$

은 우선 *expr* 의 값을 계산하여 정상적으로 수행이 이루어지면 *expr*의 계산된 값을 전체 프로그램식의 결과로 한다. 만일 *expr* 의 계산 도중에 `raise` 프로그램식에 의해서 예외상황이 발생하면, 발생된 예외상황 값을 pat_1 에서 pat_n 까지의 패턴들과 비교한다. 만일 패턴 pat_i이 발생한 예외상황과 일치한다면, $expr_i$를 계산하여 그 값을 결과로 낸다. 여기서, 패턴 pat_i의 패턴 맞추기 때에 새롭게 정의된 이름들은 $expr_i$을 계산할 때만 사용된다. 여러 개의 패턴이 일치하는 경우에는 제일 먼저 일치(가장 위에 있는)하는 패턴이 선택된다. 일치하는 패턴이 없는 경우에는 발생했던 예외상황 값을 다시 예외상황으로 발생시킨다.

6.3 데이타 구조에 대한 연산들(Operations on data structures)

튜플**(Tuples)**

프로그램식 $(expr_1, ..., expr_n)$ 은 $expr_1$ 에서 $expr_2$ 의 값으로 구성된 n 개의 원소를 가지는 튜플 값으로 계산된다. 튜플 내부의 값을 참조할 때에는 튜플 패턴을 이용한 패턴 맞추기를 사용한다(5절 참조). 튜플은 내부적으로 이름이 숫자로 되어있는 레코드의 설탕 구조로 구현되었기 때문에, 레코드와 유사하게 튜플 내부의 원소 하나를 참조할 수 있다. 예를 들어, 튜플 `(1,2,3)` 에서 첫번째 원소인 `1` 을 참조하는 방법은 `(1,2,3).0` 이다.

리스트**(Lists)**

리스트에 대해서는 몇 가지 설탕 구조 연산자가 제공된다. 프로그램식 $expr_1$`::`$expr_2$ 는 생성자 '`::`' 에 두 개의 인자인 $expr_1$, $expr_2$ 를 적용한 것과 같다. 따라서 이 프로그램식은 머리가 $expr_1$ 의 값이고, 꼬리가 $expr_2$ 의 값을 갖는 리스트로 계산된다. 따라서, 프로그램식 `[`$expr_1, ..., expr_n$`]` 은 $expr_1$`::...::`$expr_n$`::[]` 과 동일한 식이다. 리스트 내의 값을 참조할 때에는 리스트 패턴을 이용한다. 리스트 패턴은 위에서 설명한 몇가지 리스트에 대한 연산자를 사용해서 정의된다(5절 참조).

레코드(**Records**)

프로그램식 {lab_1=$expr_1$, ..., lab_n=$expr_n$} 각 레이블 lab_i에 exp_i의 값이 묶여 있는 레코드가 된다. 레코드는 레코드 타입을 정의해서 사용할 수도 있고, 타입을 정하지 않고 사용할 수도 있다. 레이블의 순서는 정해져 있지 않지만, 한 레이블은 반드시 레코드 안에서 한번만 나타날 수 있다. 레코드 내의 값을 참조할 때에는 레코드 패턴을 이용(5절 참조) 하거나, 아래에 나타나는 레코드 연산을 위한 프로그램식을 이용한다.

프로그램식 $expr$.lab 은 우선 레코드 타입을 가지는 식 $expr$을 계산하고, 해당 레코드 내의 레이블 lab 에 관련된 값을 넘겨 준다.

프로그램식 $expr_1${lab<-$expr_2$} 은 레코드 타입을 가지는 프로그램식 $expr_1$ 을 계산하고, 해당 레코드 내의 레이블 lab 에 $expr_2$ 을 계산한 값을 묶는다. 이 프로그램식을 이용하면 레코드 $expr_1$ 과 다른 레이블의 값은 모두 동일하고 lab 의 값만 $expr_2$ 의 값으로 바뀐 새로운 레코드가 생성된다. 만일 정의되지 않은 레이블이 참조 되거나 값 정의를 하는 프로그램식이 존재하면 예외상황 `Bound`가 발생한다.

배열(**Arrays**)

프로그램식 [|$expr_1$, ..., $expr_n$|] 은 n 개의 원소를 가지는 배열로 계산된다. 배열의 각 원소들은 $expr_1$ 에서 $expr_n$ 까지의 값들로 초기화된다. 배열 내의 값을 참조할 때에는 배열 패턴을 사용(5절 참조)하거나, 아래에 설명된 배열 연산을 위한 프로그램식을 이용한다.

프로그램식 $expr_1$.[$expr_2$] 은 $expr_1$ 로부터 계산되는 배열의 $expr_2$ 번째 값을 넘겨 준다. 배열의 색인(index)은 0 부터 시작된다; 배열의 첫번째 원소는 $expr_1$.[0] 으로 나타낼 수 있고, 마지막 원소는 $expr_1$.[$n-1$] 으로 나타낼 수 있다(크기가 n인 배열에서). 만일 위의 색인 범위를 넘어가면 예외상황 `Bound` 가 발생한다.

프로그램식 $expr_1$.[$expr_2$] <- $expr_3$ 은 배열의 값을 갱신한다. 즉 $expr_1$ 로부터 계산되는 배열의 $expr_2$ 번째 값을 $expr_3$ 의 값으로 바꾸어 준다. 이 프로그램식은 레코드와 달리 새로운 배열을 생성시키지는 않고 원래의 배열에서 해당되는 값만을 변화시킨다. $expr_2$ 의 값이 배열의 접근 범위를 벗어난 경우에는 예외상황 `Bound`가 발생한다. 이 프로그램식의 결과 값은 () 이다.

6.4 기본 연산자와 기본 타입(Operators and base types)

기본적인 연산자들의 타입과 그 의미를 정리해 보면 표 6.4과 같다. nML 에는 다음과 같은 기본 타입들이 있다.

`unit`	(유닛 타입)
`bool`	(boolean 값에 대한 타입)
`int`	(정수 값에 대한 타입)
`real`	(실수 값에 대한 타입)
`string`	(문자열 값에 대한 타입)
`char`	(문자 값에 대한 타입)
`list`	(리스트에 대한 타입)
`ref`	(메모리를 가리키는 포인터 값에 대한 타입)
`exn`	(예외상황에 대한 타입)
`array`	(배열에 대한 타입)

연산자	타입	의미
+,-,*,/	(infix) int → int → int	정수 덧셈/뺄셈/곱셈/나눗셈
	(infix) real → real → real	실수 덧셈/뺄셈/곱셈/나눗셈
+	(prefix) int → int	항등원
	(prefix) real → real	항등원
-	(prefix) int → int	정수 부호 바꾸기
	(prefix) real → real	실수 부호 바꾸기
%	(infix) int → int → int	정수 나머지 구하기
**	(infix) int → int → int	정수 지수 만들기
	(infix) real → real → real	실수 지수 만들기
++, --	(postfix) int ref → unit	정수 1 더하기/1 빼기 결과를 메모리에 쓰기
	(postfix) real ref → unit	실수 1 더하기/1 빼기 결과를 메모리에 쓰기
+=,-=,*=,/=	(infix) int ref → int → unit	정수 덧셈/뺄셈/곱셈/나눗셈 결과 메모리에 쓰기
	(infix) real ref → real → unit	실수 덧셈/뺄셈/곱셈/나눗셈 결과 메모리에 쓰기
<<, >>	(infix) int → int → int	비트 단위 왼쪽/오른쪽 산술적 옮기기
andalso,&&	(infix) bool → bool → bool	불린 '그리고' 연산
orelse,||	(infix) bool → bool → bool	불린 '또는' 연산
not	(prefix) bool → bool	불린 역으로 바꾸기 연산
=	(infix) 'a → 'a → bool	'같다' 연산
<>	(infix) 'a → 'a → bool	'다르다' 연산
<,<=,>,>=	(infix) 'a → 'a → bool	정수 비교
@	(infix) 'a list → 'a list → 'a list	두 리스트 붙이기
^	(infix) string → string → string	두 문자열 붙이기

표 1.2: 기본 연산자

이 기본 타입과 표 6.4에 나타나는 기본 연산자들은 다시 정의될 수 없다.

표 6.4에서 infix 로 정의된 연산자들은 인자로 받는 두 프로그램식의 가운데에 올 수 있다. prefix 로 정의된 연산자들은 인자로 받는 프로그램식의 앞부분에 올 수 있고, postfix 로 정의된 연산자들은 인자로 받는 프로그램식의 뒷부분에 올 수 있다. 이런 연산자 기호들은 특별한 함수들로써 해석이 된다. 즉, 프로그램식 $expr_1$ *infixid* $expr_2$ 은 함수 (*infixid*) 와 두개의 인자들이 모인 함수 호출식으로 (*infixid*) $expr_1$ $expr_2$ 와 같이 해석된다.

표 6.4의 기본 연산자들 중에는 메모리 효과를 이용하는 연산자들이 있다. 예를 들어, 연산자 ++ 은 int ref(혹은 real ref) 를 받아서 unit 을 되돌린다. 프로그램식 v++ 은 ref 타입의 변수 v 에 의해 메모리에 저장된 정수값(혹은 실수값)을 하나 증가시킨 뒤, 같은 메모리 영역에 다시 저장한다. 즉, v++ 은 다음과 같다.

$$v \texttt{ := !}v \texttt{ + 1}$$

프로그램식 v+=n 은 ref 타입의 변수 v 에 의해 메모리에 저장된 정수값(혹은 실수값)을 상수 n 만큼 증가시킨 뒤, 같은 메모리 영역에 다시 저장한다. 즉, v+=2 은 다음과 같다.

$$v \texttt{ := !}v \texttt{ + 2}$$

메모리 효과를 이용하는 다른 연산자들도 같은 방식으로 사용된다.

7 선언(Declarations)

선언들은 모듈 내부나 가장 윗 단계(top-level)에서 사용되는 프로그램식을 말한다.

$$
\begin{array}{rcl}
\mathit{dec} & ::= & \mathit{valdec} \\
& | & \texttt{type}\ \mathit{typbind} \\
& | & \texttt{exception}\ \mathit{exnbind} \\
& | & \texttt{local}\ \mathit{dec}_1\ \texttt{in}\ \mathit{dec}_2\ \texttt{end} \\
& | & \mathit{dec}_1\ \langle\texttt{;}\rangle\ \mathit{dec}_2 \\
& \dagger & \texttt{open}\ \mathit{strlongid}
\end{array}
$$

이 선언들과 모듈에 관련된 프로그램식을 제외한 어떠한 다른 프로그램식도 가장 윗 단계 선언 자리에 올 수 없다. Standard ML 에서는 `let` 을 이용해서 안쪽의 프로그램식에서도 타입 정의나 예외상황 정의를 할 수 있도록 하였으나, 이런 허용이 유용한 경우는 매우 드물기 때문에 nML에서는 그같은 프로그램식을 불가능하게 제약하였다.

7.1 값 선언(Value declarations)

$$
\begin{array}{rcl}
\mathit{valdec} & ::= & \texttt{val}\ \mathit{tyvarseq}\ \mathit{valbind} \\
& \dagger & \texttt{fun}\ \mathit{funbind} \\
& | & \mathit{valdec}\ \langle\texttt{;}\rangle\ \mathit{valdec}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{valbind} & ::= & \mathit{pat}\ \texttt{=}\ e\ \langle\texttt{and}\ \mathit{valbind}\rangle \\
& | & \texttt{rec}\ \mathit{valbind}
\end{array}
$$

값 선언은 변수 선언과 함수 선언이 있다. 두 가지 선언들은 `let` *valdec* `in` e_2 `end`에서 나타나는 *valdec*과 같다.

변수 이름 선언(Variable declarations)

선언식 `val` *tyvarseq pat* `=` *expr* 는 프로그램식 *expr*를 계산하고 그 값을 패턴 *pat* 과 비교해서 일치하는 경우, *pat* 내에 나타나는 이름들의 값정의를 하는 식이다. *tyvarseq* 는 다변형(polymorphic) 타입을 제한하는 경우에 추가되는 타입 변수들이다.

변수 이름의 자기 호출적(recursive)인 정의가 필요할 때에는 `val rec` 을 사용한다. 즉, `val rec` *pat* `=` *expr* 에서 패턴 *pat* 내의 이름이 프로그램식 *expr* 에서 사용되는 경우, 키워드 `rec` 이 필요하다. 이 `rec`을 사용해서 자기 호출적인 정의로 사용하는 것은 자기 호출 함수이다. 즉, 일반적으로 `val rec`은 아래와 같이 쓰인다.

$$
\texttt{val rec}\ \mathit{funid}\quad \texttt{=}\quad \texttt{fn}\ \mathit{varid}\ \texttt{=>}\ \cdots
$$

`val rec`을 대신하여 `fun` 을 사용할 수도 있다.

함수 이름 선언**(Function declarations)**

선언식 `fun` *funbind* 는 함수 프로그램식을 선언하는 식이다. 함수 및 인자들의 이름과 함수의 몸체를 정의하는 *funbind* 는 아래와 같다.

$$\begin{aligned} \mathit{funbind} \quad ::= \quad & (\mathit{fun\_id}\ \mathit{pat}^+ : \langle \mathit{ty} \rangle\ \texttt{=}\ \mathit{expr}) \\ & (\texttt{|}\mathit{fun\_id}\ \mathit{pat}^+ : \langle \mathit{ty} \rangle\ \texttt{=}\ \mathit{expr})^* \\ & \langle \texttt{and}\ \mathit{funbind} \rangle \end{aligned}$$

일반적으로, 이 형태의 함수 정의를 '|' 를 이용해서 풀어쓰면 다음과 같다.

$$\begin{aligned} & \texttt{fun}\ \mathit{fun\_id}\ \mathit{pat}_{00}\ \cdots\ \mathit{pat}_{0m}\texttt{:}\ \mathit{ty}\ \texttt{=}\ \mathit{expr}_0 \\ & \quad \cdots \\ & \texttt{|}\ \mathit{fun\_id}\ \mathit{pat}_{n0}\ \cdots\ \mathit{pat}_{nm}\texttt{:}\ \mathit{ty}\ \texttt{=}\ \mathit{expr}_n \end{aligned}$$

이 형태의 함수 정의는 함수의 이름 *fun_id* 에 m 개 만큼의 인자를 받아들이는 함수 값을 묶는다. 함수 호출식을 통해서 이 함수가 사용될 때에는, 함수의 인자로 들어오는 값들과 각 패턴들과 패턴 맞추기가 수행된다. m 개의 인자들이 패턴들과 맞추기를 해서 어떤 패턴들 $\mathit{pat}_{i0} \cdots \mathit{pat}_{im}$과 일치했다면, 해당하는 함수의 몸체인 expr_i 가 수행된다. 이때, $\mathit{pat}_{i0} \cdots \mathit{pat}_{im}$ 들의 패턴 맞추기에 의해 값이 새로 정의된 이름들은 expr_i 의 수행시에만 사용된다. 만일, 여러 개의 패턴이 일치하면, 제일 위에 있는 패턴이 선택되어 수행되고, 만일 일치하는 패턴이 없으면, 예외상황 `Match` 가 발생한다.

`and`은 함수들이 서로가 서로를 호출하는(mutual recursive) 관계에 있을 때 사용된다. 예를 들어, 어떤 숫자가 짝수인지 홀수인지를 구분하는 함수를 서로가 서로를 호출하는 함수 관계를 이용해서 작성하면 다음과 같다.

```
fun even n = if n=0 then true
             else if n=1 then false
             else odd (n-1)
and odd  n = if n=0 then false
             else if n=1 then true
             else even (n-1)
```

`fun` 을 이용한 함수 정의는 `val rec`과 `fn` 을 이용한 커리형 함수를 통해서 아래와 같이 정의할 수 있다(설탕구조).

$$\begin{aligned} & \texttt{fun}\ \mathit{fun\_id}\ \ \mathit{pat}_{00}\ \cdots\ \mathit{pat}_{0m}\ \texttt{:}\ \mathit{ty}\ \texttt{=}\ e_0 \\ & \quad \cdots \\ & \texttt{|}\ \mathit{fun\_id}\ \ \mathit{pat}_{n0}\ \cdots\ \mathit{pat}_{nm}\ \texttt{:}\ \mathit{ty}\ \texttt{=}\ e_n \\ & \equiv\ \ \texttt{val rec}\ \mathit{fun\_id}\ \texttt{=}\ \texttt{fn}\ x_0\ \texttt{=>}\ \cdots\ \texttt{=>}\ \texttt{fn}\ x_m\ \texttt{=>} \\ & \qquad \texttt{(case}\ (x_0, \cdots, x_m) \\ & \qquad\ \texttt{of}\ (\mathit{pat}_{00}, \cdots, \mathit{pat}_{0m})\texttt{=>}e_0 \\ & \qquad\quad \cdots \\ & \qquad\ \texttt{|}\ (\mathit{pat}_{n0}, \cdots, \mathit{pat}_{nm})\texttt{=>}e_n)\ \texttt{:}\ \mathit{ty} \end{aligned}$$

7.2 타입 정의(Type definitions)

타입 정의는 타입 구성자와 데이타 타입을 묶는 역할을 한다

$$\begin{array}{rcl} \textit{typedef} & ::= & \texttt{type}\ \textit{tybind} \\ \textit{typbind} & ::= & \textit{tyvarseq}\ \textit{tyid}\ \texttt{=}\ \textit{ty}\ \langle \texttt{and}\ \textit{typbind} \rangle \\ & | & \textit{tyvarseq}\ \textit{tyid}\ \texttt{=}\ \textit{conbind}\ \langle \texttt{and}\ \textit{typbind} \rangle \\ \textit{conbind} & ::= & \textit{conid}\ \langle \texttt{of}\ \textit{ty} \rangle\ \langle \texttt{|}\ \textit{conbind} \rangle \end{array}$$

타입 정의는 키워드 type 으로 시작되어 and 로 묶일 수 있는 여러 개의 간단한 타입 정의나 자기 참조(혹은 상호 참조)적인 타입 정의로 이루어 진다. 각각의 간단한 타입 정의는 하나의 타입 구성자를 정의한다.

간단한 타입 정의는 소문자 이름으로 구성되며, 경우에 따라서 한 개 혹은 여러 개의 타입 인자들이 앞에 붙을 수 있다. 또 이름의 뒤에는 '=' 로 연결된 두 가지 종류의 타입식이 올 수 있다.

첫번째는

$$\textit{typbind} \quad ::= \quad \textit{tyvarseq}\ \textit{tyid}\ \texttt{=}\ \textit{ty}$$

형태의 타입 정의이다. 타입 구성자 *tyid* 뒤에 '=' 로 연결되어 나타나는 *ty*는 4절에서 정의한 타입식들을 의미한다. 이 타입 정의는 오른편에 있는 타입식에 왼편과 같은 타입 이름을 정해 준다.

두번째는

$$\textit{typbind} \quad ::= \quad \textit{tyvarseq}\ \textit{tyid}\ \texttt{=}\ \textit{conid}\ \texttt{of}\ \textit{ty}$$

형태의 타입 정의이다. 이 타입 정의는 데이타 구성자와 데이타 타입을 묶어서 새로운 타입을 정의한 것이다. 여러 개의 데이타 구성자가 존재하는 경우에는 '|' 로 연결하여 나타낸다.

타입이 모듈 타입(signature)에 나타날 경우, 이름 뒤의 '=' 을 포함한 타입식들이 생략되기도 한다. 이 경우, 관련된 타입의 타입 구성자나 인수의 갯수 등 모든 정보가 가려지게 되며, 다른 어떠한 타입과도 동일하게 사용될 수 없다.

7.3 예외상황 정의(Exception definitions)

$$\textit{exndef} \quad ::= \quad \texttt{exception}\ \textit{conid}\ \langle \texttt{of}\ \textit{ty} \rangle\ \langle \texttt{and}\ \textit{conid...} \rangle$$

예외상황 정의[6]는 새로운 데이타 구성자를 붙박이(built-in) 타입인 exn 의 값으로 추가시켜 주는 것이다. 다음 표에 나타나는 예외상황들은 기본적인 예외상황들로써 재정의가 불가능하다.

예외상황	의미
Match[7]	맞는 패턴 없음
Zero[8]	0 으로 나눔
Overflow[9]	넘침 발생
Bound[10]	배열/레코드의 범위를 넘김
Equality[11]	의미없는 인수가 들어옴

[6] 현재의 구현에서는 ocaml에 정의된 예외상황이 발생한다.
[7] ocaml에서는 Match_failure
[8] ocaml에서는 Division_by_zero
[9] ocaml에서는 Invalid_argument
[10] ocaml에서는 해당하는 예외상황이 없다.
[11] ocaml에서는 Invalid_argument

7.4 지역적인 선언(Local declarations)

선언식 `local` dec_1 `in` dec_2 `end` 은 선언식 dec_1 의 유효 범위를 dec_2 로 한정하는데 사용된다. 이것의 역할은 프로그램식 `let` ... `end` 와 유사하나, `let` ... `end` 이 선언 단계에서 올 수 없듯이 `local` ... `end` 도 내부의 프로그램식 단계에 나타날 수 없다는 차이점이 있다.

7.5 모듈 열기(Opening a module path)

선언식 `open` *module-path* 는 컴포넌트를 새롭게 생성하거나 새로운 값 정의를 추가 하는 등의 일을 하지 않고, 단지 다음에 나타나는 모듈 내의 원소들의 파싱에 영향을 미칠 뿐이다. 즉 *module-path.simpe-name* 대신 *simple-name* 으로 사용할 수 있도록 한다. `open` 의 유효 범위는 `open` 프로그램식이 씌여진 부분에서 부터 `open` 프로그램식을 포함하는 모듈의 끝까지 이다.

8 모듈식(Modules)

모듈식은 모듈 타입(`signature`), 모듈(`structure`), 모듈 함수(`functor`)를 포함하는 모든 모듈 형식의 프로그램식들을 말한다.

8.1 모듈 타입(Signatures)

sigdec ::= `signature` *sigid* `=` *sig*

sig ::= *sigid*
| `sig` ⟨*spec*⟩ `end`
| *sig* `where type` *tyvarseq tylongid* `=` *ty* ⟨`and` *longtypbind*⟩

spec ::= `val` (*varid*|`(`*opid*`)`) `:` *ty* ⟨`and` ..⟩
| `type` *typdesc*
| `exception` *conid* ⟨`of` *ty*⟩ ⟨`|` *condesc*⟩
| `include` *sig*
| `structure` *strid* `:` *sig*
| *spec* ⟨`;`⟩ *spec*

typdesc ::= *tyvarseq tyid* ⟨`and` *typdesc*⟩
| *tyvarseq tyid* `=` *ty* ⟨`and` *typdesc*⟩
| *tyvarseq tyid* `=` *conid* ⟨`of` *ty*⟩ ⟨`|` *conid* ...⟩ ⟨`and` *typdesc*⟩

모듈 타입은 모듈에 대한 타입 접속 방안이다. 모듈 타입의 정의는 `signature` *sigid* `=` 을 통해서 이루어진다. 모듈 타입 이름 *sigid* 에 정의되는 값은 `sig` ... `end` 로 묶인 접속 방안들의 모음이다. 즉, 값의 이름이나 타입 이름, 예외상황 이름, 모듈 및 모듈 타입 이름들에 대한 타입 접속 방안이 `sig` ... `end` 사이에 나타나게 된다. 어떤 모듈이 모듈 타입(signature) 내의 모든 이름들에 대해서 값정의를 가지고 있고 타입 요구 조건에 대해서 만족하면, 그 모듈은 모듈 타입과 맞게 된다.

값 접속 방안(**Value specifications**)

모듈 타입 내에 있는 값에 대한 접속 방안은 `val` *varid* `:` *ty* 으로 정의되어 사용된다. 여기서 *varid* 는 값의 이름이고 *ty* 는 그 값의 타입이다. (*opid*) 는 문자가 아닌 기호로 구성된 이름이다. `and` 를 사용해서 여러 개의 값 접속 방안을 정의할 수 있다.

타입 접속 방안(**Type specifications**)

모듈 타입 내부의 타입 요소들의 접속 방안은 `type` *typdesc* 을 통해서 이루어진다. 타입식을 정의하는 *typdesc* 는 7.2절에서 다룬 것과 같다. 모듈 타입 내의 각 타입 정의에 해당되는 타입식은 관련된 모듈의 구현과 모순이 없어야 한다.

예외상황 접속 방안(**Exception specifications**)

예외상황 접속 방안은 `exception` *conid* ⟨`of` *ty*⟩ 의 형태로 정의된다. *conid* 는 패턴 맞추기에 사용되는 데이타 구성자의 이름이다. 선택적으로 붙을 수 있는 ⟨`of` *ty*⟩ 는 데이타 구성자 *conid* 가 받아들이는 인자의 타입을 의미한다.

모듈 타입 포함하기(**Including a signature**)

`include` *sig* 를 이용해서 모듈 타입을 포함하는 것은 *sig* 에 있는 모듈 타입을 단지 텍스트상으로 포함하는 것을 의미한다. 즉, 포함되는 모듈 타입이 `include` 위치에 복사되는 효과를 갖는다. 인자인 *sig*는 반드시 모듈 타입을 가리키는 이름이어야 한다.

모듈 접속 방안(**Structure specifications**)

`structure` *strid* `:` *sig* 은 모듈 타입 내에서 나타나는 모듈 정의이다. 임의의 모듈은 모듈 내부에 선언되어 사용될 수 있다. *strid* 는 현재의 모듈 타입 내부에서 선언되는 모듈의 이름이고 *sig* 는 그 모듈의 모듈 타입이다.

`where` 연산자

어떤 모듈 타입 *modtype* 이 *modtype* `where type` *tyid* `=` *ty* 새로운 연산자인 `where` 과 함께 사용이 된다면, 그 의미는 모듈 타입 내의 타입인 *tyid* 가 *ty* 의 타입으로 제약된다는 것이다.

예를 들어, 이름이 S 인 어떤 모듈 타입이 다음과 같은 모듈 타입으로 정의되었다고 하자.

`sig type` t `structure` M `: (sig type` u `end) end`

이때 S `where type` t `= int` 는 다음과 같은 모듈 타입을 의미한다.

`sig type` t `= int structure` M `: (sig type` u `end) end`

8.2 모듈(Structures)

`structure` 모듈식은 모듈 타입과 어울리는 모듈의 실제 값을 정의한다.

$$
\begin{array}{rcl}
\textit{strdec} & ::= & \texttt{structure}\ \textit{strid}\ \texttt{=}\ \textit{strexp} \\
& | & \texttt{structure}\ \textit{strid}\ \texttt{:}\ \textit{sig}\ \texttt{=}\ \textit{strexp} \\
\\
\textit{strexp} & ::= & \textit{strlongid} \\
& | & \texttt{struct}\ \langle \textit{strbody} \rangle\ \texttt{end} \\
& | & \textit{strexp}\ \texttt{:}\ \textit{sig} \\
& | & \textit{fctid}\ \texttt{(}\textit{strexp}\texttt{)} \\
& \dagger & \textit{fctid}\ \texttt{(}\textit{strexp}\ (\texttt{,}\ \textit{strexp})^{+}\texttt{)} \\
\\
\textit{strbody} & ::= & \textit{dec} \\
& | & \textit{strdec} \\
& | & \textit{strbody}\ \langle \texttt{;} \rangle\ \textit{strbody}
\end{array}
$$

`structure` *strid* `=` *strexp* 는 이름이 *strid* 인 어떤 모듈을 정의한 것으로, *strexp* 는 실제 값들의 모임을 정의하는 프로그램식이다. `structure` *strid* `:` *sig* `=` *strexp* 도 똑같은 모듈 정의이나, 이 경우에는 특히 모듈 타입 *sig* 으로써 모듈 외부에 보여지는 부분을 제약한다. 이때 모듈 타입 *sig* 의 모든 정의는 그것의 구현인 모듈 *strid* 와 모순없이 잘 맞아야 한다.

모듈의 값정의(*strexp*)

모듈의 실제 값으로는 모듈 이름이나, `struct` ... `end` 를 이용한 값들의 모임 혹은 모듈 함수 호출식이 올 수 있다. 모듈의 이름을 나타내는 *strlongid* 는 '`.`'을 포함하는 긴 이름을 말하는데, 이 이름이 가리키는 것은 모듈 값이어야 한다.

모듈 `struct` ... `end` 는 값의 이름, 타입, 예외상황, 모듈 이름 및 타입 이름 등의 정의를 가지고 있는 묶음이다. 각 정의들은 모듈 내에 정의된 순서대로 계산된다. 각각의 값 정의들의 유효범위는 모듈의 마지막까지이며, 어떤 위치에서 이름에 대한 값정의는 모듈 내부에서 정의된 그 이전의 값정의에 따르게 된다. 프로그램식 *strexp* `:` *sig* 는 모듈의 실제 값 *strexp* 이 특정 모듈 타입 *sig* 의 제약을 받는 것을 의미한다.

모듈 함수를 이용하는 모듈의 값정의는 모듈 함수 호출식을 통해서 이루어진다. 모듈 함수 호출식 *fctid* `(`*strexp*`)` 는 모듈 함수 *fctid* 에 인자 모듈인 *strexp* 을 적용시켜 나온 모듈 값을 갖는다. 모듈 함수는 한 개 이상의 인자를 받아들일 수 있다.

모듈의 내부 프로그램식(*strbody*)

모듈의 내부에는 앞의 7에서 정의한 선언문(*dec*)들이 올 수 있고, 다른 새로운 모듈(*strdec*)을 모듈 속에서 정의하고 사용할 수 있다. 선언이나 모듈을 이용한 프로그램식의 나열도 모듈 내부에 나타날 수 있다.

8.3 모듈 함수(Functors)

$$
\begin{array}{rcl}
\mathit{fntordec} & ::= & \texttt{functor}\ \mathit{fctid}\ (\ \mathit{strid} : \mathit{sig}_1\ )\ \texttt{=}\ \mathit{strexp} \\
& \dagger & \texttt{functor}\ \mathit{fctid}\ (\ \mathit{strid} : \mathit{sig}_1\ )\ \texttt{:}\ \mathit{sig}_2\ \texttt{=}\ \mathit{strexp} \\
& \dagger & \texttt{functor}\ \mathit{fctid}\ (\ \mathit{strid}_1 : \mathit{sig}_1\ (\texttt{,}\ \mathit{strid}_2 : \mathit{sig}_2)^{+}\ )\ \langle \texttt{:}\ \mathit{sig}_3 \rangle\ \texttt{=}\ \mathit{strexp}
\end{array}
$$

모듈 함수 정의(**Functor definition**)

프로그램식 `functor` *fctid* `(` *strid* `:` sig_1 `)` `=` *strexp* 은 *fctid* 의 이름을 가진 모듈 함수 정의로써 인자로 *sig* 의 타입을 갖는 *strid* 모듈을 받아들인다. *strid* 로 이름이 붙여진 인수 모듈을 가지고 *strexp* 의 계산을 수행한다. 이 계산 수행이 끝나고 생기는 모듈이 모듈 함수의 결과 모듈이 된다. 프로그램식 `functor` *fctid* `(` *strid* `:` sig_1 `)` `:` sig_2 `=` *strexp* 는 모듈 함수의 결과 모듈의 타입을 모듈 타입 sig_2 로 한정하고 있다. 모듈 함수는 하나 이상의 인수 모듈을 받을 수 있다.

모듈 함수 호출식(**Functor application**)

프로그램식 *fctid* (*strexp*) 는 모듈 함수 타입을 갖는 *fctid* 에 모듈 *strexp* 을 인자로 적용하여 계산한다. 인자인 *strexp* 의 타입은 모듈 함수 *fctid* 의 정의에 있는 인자 타입과 일치하여야 한다.

9 최상위 단계의 선언들(Top level declarations)

$$
\begin{array}{rcl}
\mathit{topdec} & ::= & \mathit{dec} \\
& | & \mathit{sigdec} \\
& | & \mathit{fntordec} \\
& | & \mathit{strdec} \\
& | & \mathit{topdec}_1\ \langle \texttt{;} \rangle\ \mathit{topdec}_2
\end{array}
$$

프로그램의 가장 높은 단계에 올 수 있는 프로그램식들은 위의 것들로 한정된다. 값 정의 및 타입 정의, 예외상황 정의 등을 포함하는 선언문들로 구성된 *dec* 와 모듈 타입 정의인 *sigdec*, 모듈 함수 정의인 *fntordec*, 모듈 정의 *strdec* 그리고 위의 것들이 여러 개 나열된 형태인 $topdec_1\ \langle \texttt{;} \rangle\ topdec_2$ 가 최상위 단계에 올 수 있는 프로그램식이다.