정답과 해설

esg003

2019/7/10

연습문제

응용 1. 다음과 같이 이루어진 수열이 있다. 인자로 주어지는 n번째 항까지의 합을 출력하여라.

3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2, 2, 2, 1, 84, 2, 1, 1, 15, 3, 13, 1, 4, 2,
6, 6, 99, 1, 2, 2, 6, 3, 5, 1, 1, 6, 8, 1, 7, 1, 2, 3, 7, 1, 2, 1, 1, 12, 1, 1, 1, 3, 1, 1, 8, 1, 1, 2,
1, 6, 1, 1, 5, 2, 2, 3, 1, 2, 4, 4, 16, 1, 161, 45, 1, 22, 1, 2, 2, 1, 4, 1, 2, 24, 1, 2, 1, 3, 1, 2

seq와 clist clist가 seq의 간략화한 것임을 이미 언급하였습니다. 일반적으로 seq가 속도도 (극히 조금) 빠르고 할 수 있는 일도 많습니다. 그러므로 "리스트"가 문제가 되면 대체로 seq로 문제를 해결하는 것이 좋습니다.
그러면 clist라는 것은 왜 있는가? LaTeX에 콤마로 분리된 리스트가 많이 쓰였기 때문입니다. 특히 명령이나 패키지 선언의 인자로 이런 형태가 자주 나오는데 이것을 간편하게 처리하기 위한 장치입니다.
지금까지 clist는 충분히 연습했으므로 이 문제는 seq로 해결해봅니다.

```
\ExplSyntaxOn
\seq_set_from_clist:Nn \l_tmpa_seq
{
    3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2, 2, 2, 1, 84,
    2, 1, 1, 15, 3, 13, 1, 4, 2, 6, 6, 99, 1, 2, 2, 6, 3, 5, 1, 1, 6, 8, 1,
    7, 1, 2, 3, 7, 1, 2, 1, 1, 12, 1, 1, 1, 3, 1, 1, 8, 1, 1, 2, 1, 6, 1, 1,
    5, 2, 2, 3, 1, 2, 4, 4, 16, 1, 161, 45, 1, 22, 1, 2, 2, 1, 4, 1, 2, 24,
    1, 2, 1, 3, 1, 2
}
\ExplSyntaxOff
```

clist에서는 item의 index를 처리하기 위하여 int 하나를 zero로 만들고 map function에서 이것을 incr 하면서 세었습니다. 이것도 나쁘지 않지만 seq에는 아예 인덱스를 활용할 수 있는 함수가 제공됩니다. 세 개마다 fbox하는 일을 해보자면,

```
\ExplSyntaxOn
\seq_set_from_clist:Nn \l_tmpa_seq
{
    3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2, 2, 2, 1, 84,
    2, 1, 1, 15, 3, 13, 1, 4, 2, 6, 6, 99, 1, 2, 2, 6, 3, 5, 1, 1, 6, 8, 1,
    7, 1, 2, 3, 7, 1, 2, 1, 1, 12, 1, 1, 1, 3, 1, 1, 8, 1, 1, 2, 1, 6, 1, 1,
    5, 2, 2, 3, 1, 2, 4, 4, 16, 1, 161, 45, 1, 22, 1, 2, 2, 1, 4, 1, 2, 24,
    1, 2, 1, 3, 1, 2
}
```

```
\seq_indexed_map_inline:Nn \l_tmpa_seq
{
    \int_compare:nTF { \int_mod:nn { #1 } { 3 } == 0 }
    {
        \fbox { #2 }
    }
    {
        #2
    }

    \int_compare:nF { \seq_count:N \l_tmpa_seq == #1 }
    {
        ,~
    }
}
\ExplSyntaxOff
```

3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2, 2, 2, 1, 84, 2, 1, 1, 15, 3, 13, 1, 4, 2, 6, 6, 99, 1, 2, 2, 6, 3, 5, 1, 1, 6, 8, 1, 7, 1, 2, 3, 7, 1, 2, 1, 1, 12, 1, 1, 1, 3, 1, 1, 8, 1, 1, 2, 1, 6, 1, 1, 5, 2, 2, 3, 1, 2, 4, 4, 16, 1, 161, 45, 1, 22, 1, 2, 2, 1, 4, 1, 2, 24, 1, 2, 1, 3, 1, 2

`\seq_indexed_map_function:`을 쓸 적에 주의할 점은 인덱스가 #1 (##1), 아이템이 #2 (##2)라는 것입니다. 착각하지 않도록 하세요.

문제 자체는 어렵지 않습니다. 문제에서 주어진 수열은 "고정된" 것으로서 이 수열 자체에 조작을 가할 일이 없으므로 "상수"로 선언할 수 있습니다. 그리고 합을 구하는 과정은 전역 변수 `\g_sum_int`를 선언해두고 여기에 값을 넣어 전달하는 방식(procedure)으로 처리하겠습니다.

```
\ExplSyntaxOn
\seq_const_from_clist:Nn \c_picontfrac_seq
{
    3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2, 2, 2, 1, 84,
    2, 1, 1, 15, 3, 13, 1, 4, 2, 6, 6, 99, 1, 2, 2, 6, 3, 5, 1, 1, 6, 8, 1,
    7, 1, 2, 3, 7, 1, 2, 1, 1, 12, 1, 1, 1, 3, 1, 1, 8, 1, 1, 2, 1, 6, 1, 1,
    5, 2, 2, 3, 1, 2, 4, 4, 16, 1, 161, 45, 1, 22, 1, 2, 2, 1, 4, 1, 2, 24,
    1, 2, 1, 3, 1, 2
}

\int_new:N \g_sum_int

\NewDocumentCommand \testsum { m }
{
    \int_zero:N \g_sum_int

    \int_compare:nTF { #1 > \seq_count:N \c_picontfrac_seq }
    {
        number~too~large. ($\le$ \seq_count:N \c_picontfrac_seq ) \par
    }
```

```
    {
        \test_sum:n { #1 }
        \int_use:N \g_sum_int
    }
}

\cs_new:Npn \test_sum:n #1
{
    \seq_indexed_map_inline:Nn \c_picontfrac_seq
    {
        \int_compare:nTF { ##1 <= #1 }
        {
            \int_gadd:Nn \g_sum_int { ##2 }
        }
        {
            \seq_map_break:
        }
    }
}

\testsum{5}, \quad \testsum{500}

\ExplSyntaxOff
```

318, number too large.(≤97)

intarray 한편, 정수로 이루어진 리스트 자료형으로 intarray라는 것이 있습니다. 이름 그대로 정수 배열이기는 한데 일반적인 리스트처럼 mapping하거나 조작하는 것은 할 수 없고 그 대신 각 아이템에 대한 접근 효율을 높인 것이라고 보면 됩니다. (seq의 아이템에 대한 접근 시간은 linear인 반면 intarray는 constant) 주어진 문제는 이것을 이용하는 것이 가능합니다.

```
\ExplSyntaxOn
\intarray_const_from_clist:Nn \l_tmpa_intarray
{
    3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2, 2, 2, 1, 84,
    2, 1, 1, 15, 3, 13, 1, 4, 2, 6, 6, 99, 1, 2, 2, 6, 3, 5, 1, 1, 6, 8, 1,
    7, 1, 2, 3, 7, 1, 2, 1, 1, 12, 1, 1, 1, 3, 1, 1, 8, 1, 1, 2, 1, 6, 1, 1,
    5, 2, 2, 3, 1, 2, 4, 4, 16, 1, 161, 45, 1, 22, 1, 2, 2, 1, 4, 1, 2, 24,
    1, 2, 1, 3, 1, 2
}

\NewDocumentCommand \testsuma { m }
{
    \int_zero_new:N \g_sum_int

    \int_compare:nTF { #1 > \intarray_count:N \l_tmpa_intarray }
    {
        number~too~large. ($\le$ \intarray_count:N \l_tmpa_intarray ) \par
    }
```

```
    {
        \test_suma:n { #1 }
        \int_use:N \g_sum_int
    }
}

\cs_new:Npn \test_suma:n #1
{
    \int_step_inline:nn { #1 }
    {
        \int_gadd:Nn \g_sum_int
            { \intarray_item:Nn \l_tmpa_intarray { ##1 } }
    }
}

\ExplSyntaxOff

\testsuma{5}
```

```
318
```

정수로 이루어진 리스트의 경우에 `\seq_item:Nn`보다 `\intarray_item:Nn`이 조금 효율이 높습니다.

숙제에 대한 코멘트

박승원 군은 clist에 넣고 `\int_step_inline:nn` 하면서 `\clist_item:Nn`으로 취한 값을 더해가는 방법을 썼습니다. 표준적인 해결방법이고 다음 한 줄이 핵심입니다.

```
    \int_add:Nn \l_tmpa_int { \clist_item:Nn \l_my_clist { ##1 } }
```

이재호 군은 이 방법이 뭔가 (뻔해서?) 마음에 들지 않았는지 `\clist_map_inline:Nn`으로 처리하다가 `\clist_map_break:`하는 방법과 `\clist_pop:NN`을 이용하는 방법을 보여주었습니다. 다음은 `\clist_pop:NN`을 이용한 코드입니다.

```
    \int_zero:N \l_tmpa_int
    \int_step_inline:nn { #1 }
      {
        \clist_pop:NN \l_tmpa_clist \l_tmpa_tl
        \int_add:Nn \l_tmpa_int { \l_tmpa_tl }
      }
    \int_use:N \l_tmpa_int
```

`\clist_pop:NN`은 원본 clist에 변경을 가합니다. 이 점을 유념하여 clist의 재사용이 문제되지 않는 상황이라면 좋은 해결책입니다.

연습문제

기본 2. 주어진 수까지 더하는 대신 곱한다면 계승(factorial)을 구하는 trivial한 함수를 만들 수 있다. 이를 작성하여라. 단 인자는 12이하의 정수로 한다.

```
\ExplSyntaxOn
\NewDocumentCommand \simplefact { m }
{
    \int_set:Nn \l_tmpa_int { \c_one_int }

    \int_step_inline:nn { #1 }
    {
        \int_set:Nn \l_tmpa_int { \l_tmpa_int * ##1 }
    }

    \int_use:N \l_tmpa_int
}
\ExplSyntaxOff

\simplefact{10}
```

3628800

`\int_set:Nn`은 왜 `\int_set:Nx`나 `\int_set:No`가 없는가에 대해 생각해봅시다.
interface 함수 `\int_set:Nn`의 두 번째 인자 n 부분은 사실 `\int_eval:n`을 거친 결과를 받아들입니다. 이것을 "정수 표현식"이라고 하는데 `\int_set:Nn`의 두 번째 인자는 이 "정수 표현식"이 입력되어야 하는 것입니다.
`\int_eval:n`은 그 인자로 오는 정수 표현식 안에 있는 tl을 x-확장한 결과가 other 숫자이면 유효하게 처리합니다. 그리고 이미 int로 정의된 변수라면 이것을 확장하여 정수로 인식합니다. 즉, 다음과 같은 식은 효력이 있습니다.

```
\ExplSyntaxOn
\int_set:Nn \l_tmpa_int { 10 }
\tl_set:Nn \l_tmpa_tl { 100 }
\int_set:Nn \l_tmpb_int { \l_tmpa_int + \l_tmpa_tl }
\int_use:N \l_tmpb_int
\ExplSyntaxOff
```

110

세 번째 줄 `\int_set:Nn \l_tmpb_int` 부분의 인자는 사실 모두 확장되어야 할 매크로를 포함하지만 이 인자 부분이 미리 `\int_eval:n` 처리를 거치므로 별도의 확장 명령을 쓰지 않아도 모두 처리되는 것입니다. 그러므로, 이재호 군이

```
\exp_args:NNo \int_set:Nn \l_tmpa_int { \l_tmpa_tl }
```

처럼 한 것이나, 박승원 군이

```
\int_set:Nn \l_tmpa_int { \int_eval:n { \l_tmpa_tl } }
```

한 것은, (지금까지 강의를 잘 따라온 증거겠지만) 간단히

```
\int_set:Nn \l_tmpa_int { \l_tmpa_tl }
```

로 충분하다는 것을 알 수 있습니다.
`\clist_item:Nn`이나 `\seq_item:Nn`의 n인자와 같이 "반드시 정수"를 요구하는 interface 명령은 대부분 `\int_eval:n`을 거치도록 정의되어 있습니다.
그러나, 사용자 정의 명령이라면 정수 표현식의 처리는 함수 작성자에게 그 책임이 있습니다.
또한,

```
\int_set:Nn \l_tmpa_int { \l_tmpa_int * 10 }
```

이런 변수 할당 식에서 `\int_eval:n`의 효과 때문에 먼저 괄호 안이 확장되고 그 다음에 할당이 이루어집니다.

예시 답안에 `\c_one_int`라는 상수 하나를 썼습니다. 숫자 상수는 `\c_zero_int`와 `\c_one_int`, 그리고 `\c_max_int`가 있는데 각각 다음 수를 의미합니다.

```
\ExplSyntaxOn
\int_use:N \c_zero_int \quad
\int_use:N \c_one_int \quad
\int_use:N \c_max_int
\ExplSyntaxOff
```

```
0  1  2147483647
```

상수를 쓰면 0이나 1의 경우에 parsing하는 데 걸리는 시간이 절약된다고 합니다만 큰 차이 있겠어요?

확장 문제의 보충 다음 예를 보세요.

```
\ExplSyntaxOn
\clist_set:Nn \l_tmpa_clist { 3, 1, 4, 1, 5, 9, 2 }

\int_zero:N \l_tmpa_int
\tl_clear:N \l_tmpa_tl

\cs_new:Npn \test_cs:n #1
{
    \int_step_inline:nn { #1 }
    {
        \int_set:Nn \l_tmpb_int { ##1 }
        \int_add:Nn \l_tmpa_int { \clist_item:Nn \l_tmpa_clist { \l_tmpb_int
    + 1 } }
        \tl_put_right:Nn \l_tmpa_tl { \clist_item:Nn \l_tmpa_clist {
    \l_tmpb_int + 1 } }
    }
}

\test_cs:n { 4 }
\int_use:N \l_tmpa_int \\
\tl_use:N \l_tmpa_tl
\ExplSyntaxOff
```

```
11
5555
```

둘 다 `:Nn`을 썼는데, 위의 문장은 성공적으로 아이템을 가져오지만 뒤의 것은 "맨 마지막에 호출된 아이템"만이 성공합니다. 그 이유를 설명할 수 있겠나요? 이것을 의도대로 매번 달라지는 값이 들어가게 하려면 어떻게 해야 할까요?

팩토리얼 팩토리얼을 구하는 것은 프로그래밍 연습에서 많이들 해보는 예제입니다. 다음과 같이 재귀적으로 정의되는 함수

```
def fact(n):
    if n<=0:
        return 1
    return n * fact(n-1)
```

를 expl3로도 물론 정의는 할 수 있는데요,

```
\ExplSyntaxOn
\NewDocumentCommand \rfacto { m }
{
    \factorial_rec_fn:n { #1 }
}

\cs_new:Npn \factorial_rec_fn:n #1
{
    \int_compare:nTF { #1 <= 1 }
    {
        1
    }
    {
        \int_eval:n { #1 * \factorial_rec_fn:n { \int_eval:n { #1 - 1 } } }
    }
}

\ExplSyntaxOff

\rfacto{10}
```

3628800

이미 설명한 대로, 직접 값을 돌려주는 위의 정의보다는 다음처럼 하는 것이 대체로 더 안전합니다.

```
\ExplSyntaxOn
\NewDocumentCommand \factorialp { m }
{
    \int_set:Nn \g_tmpa_int { \c_one_int }
    \facto_func:n { #1 }
    \int_use:N \g_tmpa_int
}

\cs_new:Npn \facto_func:n #1
{
    \int_compare:nT { #1 > 1 }
    {
        \int_gset:Nn \g_tmpa_int { \g_tmpa_int * #1 }
        \facto_func:n { \int_eval:n { #1 - 1 } }
    }
```

```
}

\ExplSyntaxOff
\factorialp{10}
```

```
3628800
```

expl3의 interface function들인 `\clist_item:Nn`이나 `\int_set:Nn`, `\int_add:Nn` 같은 데서 n 부분에 굳이 `\int_eval:n`을 쓰지 않아도 됐는데, 위의 사용자 정의 함수 `\facto_func:n`은 왜 이걸 인자에다가 써주어야 하는지에 대해 앞서 설명하였습니다. **사용자 정의 함수에서 인자의 확장에 대한 책임은 작성자에게 달려 있다**는 것입니다.

아무튼 팩토리얼에 대하여 재귀 함수 연습을 위한 목적으로는 이런 걸 해볼 수 있지만 실용적으로 팩토리얼을 써야 할 때는 다음처럼 합니다.
fp 자료형의 함수 가운데 `fact`가 있습니다.

```
\ExplSyntaxOn
\fp_eval:n { fact ( 13 ) } \\
\fp_eval:n { fact ( 25 ) }
\ExplSyntaxOff
```

```
6227020800
15511210043330990000000000
```

큰 수 다루기는 다음 기회에 따로 공부할 기회가 있겠지만 미리 bnumexpr라는 xint 엔진을 사용하는 패키지 하나를 사용해보면

```
\thebnumexpr 13!\relax \\
\thebnumexpr 25!\relax
```

```
6227020800
15511210043330985984000000
```

이 두 결과에 차이가 나는 이유는 fp 자료형이 유효숫자 16자리의 근삿값 계산을 하기 때문입니다.

연습문제

기본 3. 30이하의 정수 인자를 받아서 2^n 연산의 결과를 출력하는 함수 `\bin_power:n`을 작성하여라. 단 fp 자료형의 power 연산자 `**`를 쓰지 말고 int로 계산하여야 하며, 그 결과는 `\fp_eval:n { 2**#1 }`과 같아야 한다.

다음 관계식을 이용하겠습니다.

$$2^n = 2 \times 2^{n-1} \quad (2^0 = 1,\ 1 \le n \le 30)$$

```
\ExplSyntaxOn
\cs_new:Npn \bin_power:n #1
{
    \int_compare:nTF { #1 > 30 }
    {
        number~too~large \par
    }
    {
        \int_case:nnF { #1 }
        {
            { 0 } { 1 }
            { 1 } { 2 }
        }
        {
            \int_eval:n { 2 * \bin_power:n { #1 - 1 } }
        }
    }
}

\bin_power:n { 10 }

\ExplSyntaxOff
```

```
1024
```

숙제에 대한 코멘트

이재호 군이 간명한 해결책을 보여줬습니다.

```
    \int_step_inline:nn { #1 }
      {
        \exp_args:NNo \int_set:Nn \l_tmpa_int { \l_tmpa_int * 2 }
      }
```

누누이 설명한 바 `\int_set:Nn`의 두 번째 인자는 `\int_eval:n` 확장을 거치므로 `\exp_args:NNo`는 이 경우에 필요없겠죠.
다음은 박승원 군의 코드인데요,

```
    \int_step_inline:nn { #1 }
    {
        \int_set:Nn \l_tmpa_int { \int_eval:n { \l_tmpa_int * 2 } }
    }
```

역시 `\int_set:Nn`의 정의상 `n`인자의 `\int_eval:n`은 필요없겠죠.

연습문제

실력 4. 인자로 2진수가 주어진다. 이를 10진수로 바꾸어서 출력하는 함수를, expl3가 제공하는 진법 변환 함수를 차용하지 않고 작성하여라. 필요하다면 연습문제 3에서 작성한 `\bin_power:n`을 활용하여라.

이재호 군의 코드는 다음과 같습니다.

```
\tl_set:Nn \l_tmpa_tl { #1 }
\tl_reverse:N \l_tmpa_tl

\int_zero:N \l_tmpa_int
\int_set:Nn \l_tmpb_int { 1 }
\tl_map_inline:Nn \l_tmpa_tl
  {
    \int_add:Nn \l_tmpa_int { \l_tmpb_int * ##1 }
    \exp_args:NNo \int_set:Nn \l_tmpb_int { \l_tmpb_int * 2 }
  }
\int_use:N \l_tmpa_int
```

문자열을 뒤집어서 처리했네요. 명시적으로 `\bin_power:n`을 사용하지는 않았지만 사실상 같은 방법을 사용하였습니다.
박승원 군의 코드는 다음과 같습니다.

```
\int_zero:N \l_tmpa_int
\tl_map_inline:nn { #1 }
{
    \int_set:Nn \l_tmpa_int { \int_eval:n { \l_tmpa_int * 2 + ##1 } }
}
\int_use:N \l_tmpa_int
```

문자열을 뒤집지 않고도 2를 곱하여 한 자리씩 왼쪽으로 보낼 수 있음을 보여주었습니다.

만약 `\bin_power:n`을 반드시 사용하라는 조건이 붙었다면 대략 다음과 같이 하는 것도 가능합니다만 2진법의 10진법 전환이라는 목적에 비추어보면 계산이 효율적이지는 못합니다. 다만 외부 함수를 부르는 것, case를 쓰는 것 등을 보이는 의미에서 다음 코드도 붙여두기로 합니다.

```
\ExplSyntaxOn
\int_new:N \l_sum_int

\NewDocumentCommand \mytodec { m }
{
    \tl_set:Nn \l_tmpa_tl { #1 }
    \tl_reverse:N \l_tmpa_tl

    \int_zero:N \l_tmpa_int
    \int_zero:N \l_sum_int

    \tl_map_inline:Nn \l_tmpa_tl
    {
        \str_case:nn { ##1 }
```

```
        {
            { 0 } { }
            { 1 } {
                 \int_add:Nn \l_sum_int { \bin_power:n { \l_tmpa_int } }
            }
        }

        \int_incr:N \l_tmpa_int
    }

    \int_use:N \l_sum_int
}

\mytodec{101111}

\ExplSyntaxOff
```

```
47
```

입력을 문자열로 보고 이를 뒤집어서 처리하고, 게다가 0과 1을 비교하는 데서 `\int_case:nn`가 아니라 `\str_case:nn`을 쓰고 있습니다. `\int_...` 함수가 항상 `\int_eval:n`하는 단계를 거쳐야 하기 때문에 살짝 부하가 걸린다는 것을 의식한 트릭이지만 사실은 큰 차이 있을까 싶습니다.
주의할 것은 첫 번째 자리의 지수가 0이라는 것입니다. 그래서 `\int_incr:N`을 inline 함수 맨 마지막에 두었습니다.

진법변환 expl3의 interface 함수로 진법변환에 쓰는 것은 다음과 같습니다.

- `\int_to_bin:n`, `\int_from_bin:n`
- `\int_to_hex:n`, `\int_from_hex:n`
- `\int_to_oct:n`, `\int_from_oct:n`
- `\int_to_base:nn`, `\int_from_base:nn`

이름이 모든 것을 말해주므로 설명이 필요없을 듯하고요. 임의의 진법으로의 변환에 쓰이는 `\int_to_base:nn` 정도를 주의해서 보면 될 듯합니다. `\int_to_hex`와 `\int_to_base`는 뒤의 첫 글자를 대문자로 쓴 `\int_to_Hex:n` 또는 `\int_to_Base:nn`라는 변형이 있는데 수 대신 사용하는 알파벳 문자를 대문자로 쓰게 해줍니다.

```
\ExplSyntaxOn
\int_to_base:nn { 10 } { 7 } \\
\int_to_Base:nn { 300 } { 32 }
\ExplSyntaxOff
```

```
13
9C
```

원래 문제에 대하여 이 함수를 쓰면

```
\ExplSyntaxOn
\int_from_bin:n { 101110 }
\ExplSyntaxOff
```

```
46
```

연습문제

기본 1. 주어지는 자연수를 우리말로 읽어서 그 결과를 한글로 출력하여라. 단 입력되는 숫자는 5자리 이하로 한다.

각 자리에 0이 오는 경우에는 단위까지 포함하여 읽지 않습니다. 1인 경우, 이것이 1의 자리일 때는 "일"을 새기고, 그 윗자리(만까지)에서는 단위만 읽습니다.

```
\ExplSyntaxOn
\clist_const:Nn \c_d_clist { {}, 이, 삼, 사, 오, 육, 칠, 팔, 구 }
\clist_const:Nn \c_u_clist { {}, 십, 백, 천, 만 }

\tl_new:N \l_res_tl

\NewDocumentCommand \numtohangul { m }
{
  \tl_clear:N \l_res_tl
  \tl_set:Nn \l_tmpa_tl { #1 }
  \tl_reverse:N \l_tmpa_tl

  \int_zero:N \l_tmpa_int

  \tl_map_inline:Nn \l_tmpa_tl
  {
    \int_incr:N \l_tmpa_int
    \int_compare:nF { ##1 == 0 }
    {
      \bool_if:nT
      { \int_compare_p:n { ##1 == 1 } && \int_compare_p:n { \l_tmpa_int == 1 } }
      {
        \tl_put_left:Nn \l_res_tl { 일 }
      }

      \tl_put_left:Nx \l_res_tl { \clist_item:Nn \c_u_clist { \l_tmpa_int } }
      \tl_put_left:Nx \l_res_tl { \clist_item:Nn \c_d_clist { ##1 } }
    }
  }

  \l_res_tl
}

\ExplSyntaxOff

\numtohangul{10211}\quad \numtohangul{1002}
```

```
만이백십일  천이
```

이번에도 문자열을 뒤집어서 처리했는데요, 그 대신 `\tl_put_left:Nn`를 이용하여 역순으로 읽은 결과를 쌓아간 것입니다.
이재호 군이 만 단위 띄어쓰기를 구현했는데요, 여기에 적용하려면 `\c_u_clist`의 마지막 아이템을

```
\clist_const:Nn \c_u_clist { {}, 십, 백, 천, 만~ }
```

로 하면 될 듯하네요. 그런데 이렇게 하면 `\numtohangul{10000}`의 경우에 마지막에 스페이스 하나가 붙는 위험이 있습니다. 안전하게 하기 위해서

```
\regex_replace_once:nnN { \s $ } { } \l_res_tl
```

을 마지막에 실행해주면 끝의 스페이스를 제거할 수 있습니다.
이 아이디어는 곧바로 6자리 이상의 수에 대해서도 확장가능합니다. 조금 복잡해지기는 하겠지만. 예를 들면 억 이후로는 100,000,000을 "일억"이라고 읽어서 "일"을 새겨주는 것을 주의한다든가…….

숙제에 대한 코멘트

"일, 십, 백, 천, 만"의 단위와 "일, 이, 삼, …, 구"의 자릿수를 "리스트"에 넣고 이를 `item` 함수로 참조하여 문자열을 구성한다는 아이디어를 모두 구현하였습니다.
실제 처리에서는 조금씩 차이를 보이는데, 이재호 군은 tl 리스트를, 박승원 군은 clist를 사용하였습니다.
입력되는 숫자를 처리하는 순서는 inline mapping 시의 카운터를 역순으로 세는 방법을 적용하였네요.
`\int_decr:N`이나 -1 하는 방식으로 세었어요.
0과 1을 처리하는 것은 구현은 조금 다르지만 원하는 목적을 잘 달성한 것으로 보입니다.
이재호 군의 코드는 다음과 같습니다.

```
\ExplSyntaxOn
\tl_gset:Nn \g_prefix_tl { 일이삼사오육칠팔구 }
\tl_gset:Nn \g_suffix_tl { 십백천만 }

\NewDocumentCommand \numtohangul { m }
 {
  \int_set:Nn \l_tmpa_int { \tl_count:n { #1 } - 1 }
  \tl_map_inline:nn { #1 }
   {
    \bool_if:nT
     {
      \int_compare_p:n { \l_tmpa_int > 3 } ||
      \int_compare_p:n { \l_tmpa_int == 0 } ||
      \int_compare_p:n { ##1 != 1 }
     }
     {
      \tl_item:Nn \g_prefix_tl { ##1 }
     }
    \int_compare:nT { ##1 != 0 }
     {
      \tl_item:Nn \g_suffix_tl { \l_tmpa_int }
     }
    \int_compare:nT { \l_tmpa_int == 4 } { ~ }
    \int_decr:N \l_tmpa_int
   }
 }
\ExplSyntaxOff

\numtohangul{2019}\\
\numtohangul{11019}\\
\numtohangul{909}
```

이천십구
일만 천십구
구백구

박승원 군의 코드는 다음과 같습니다.

```
\ExplSyntaxOn
\clist_new:N \l_digit_clist
\clist_set:Nn \l_digit_clist { 일, 십, 백, 천, 만 }
\clist_new:N \l_num_clist
\clist_set:Nn \l_num_clist { 일, 이, 삼, 사, 오, 육, 칠, 팔, 구 }
\NewDocumentCommand \numtohangul { m }
{
  \int_set:Nn \l_tmpa_int { \tl_count:n { #1 } }
  \tl_map_inline:nn { #1 }
  {
    \int_case:nnTF { ##1 }
    {
      { 0 } { }
      { 1 } { \clist_item:Nn \l_digit_clist { \l_tmpa_int } }
    }
    { }
    {
      \clist_item:Nn \l_num_clist { ##1 }
      \int_compare:nTF { \l_tmpa_int == 1 }
      { }
      {
        \clist_item:Nn \l_digit_clist { \l_tmpa_int }
      }
    }
    \int_add:Nn \l_tmpa_int { -1 }
  }
}
\ExplSyntaxOff

\numtohangul{102}
\numtohangul{120}
\numtohangul{123}
\numtohangul{2019}
\numtohangul{98765}
\numtohangul{101}
```

백이 백이십 백이십삼 이천십구 구만팔천칠백육십오 백일

연습문제

실력 2. 두 수를 인자로 받아 그 최대공약수를 출력하는 명령 `\mygcd`를 작성하여라.

```
def mygcd(a,b):
    if a<b:
        a,b=b,a
    while b != 0:
        t = a%b
        a,b=b,t
    return a
```

위의 알고리즘을 그대로 expl3로 옮겨쓰면 다음과 같이 됩니다.

```
\ExplSyntaxOn

\int_new:N \l_tmpc_int

\NewDocumentCommand \mygcd { m m }
{
    \int_set:Nn \l_tmpa_int { \int_max:nn { #1 } { #2 } }
    \int_set:Nn \l_tmpb_int { \int_min:nn { #1 } { #2 } }

    \int_while_do:nn { \l_tmpb_int != 0 }
    {
        \int_set:Nn \l_tmpc_int
            { \int_mod:nn { \l_tmpa_int } { \l_tmpb_int } }
        \int_set_eq:NN \l_tmpa_int \l_tmpb_int
        \int_set_eq:NN \l_tmpb_int \l_tmpc_int
    }

    \int_use:N \l_tmpa_int
}

\ExplSyntaxOff
\mygcd{16}{24}
```

```
8
```

예를 들어,

```
a,b=b,a-b
```

이런 python 코드를 expl3로 쓴다고 할 때,

```
\ExplSyntaxOn
\int_set:Nn \l_tmpa_int { 10 }
\int_set:Nn \l_tmpb_int { 4 }

\int_set_eq:NN \l_tmpa_int
    \l_tmpb_int
\int_set:Nn \l_tmpb_int {
    \l_tmpa_int - \l_tmpb_int }

\int_use:N \l_tmpa_int; ~
    \int_use:N \l_tmpb_int
\ExplSyntaxOff
```

4; 0

보다시피 결과가 잘못되어 있습니다. 그 이유는 `\l_tmpa_int` 값이 변화한 이후에 이것을 계산에 사용했기 때문이므로, 반드시 temp를 하나 써서

```
\ExplSyntaxOn
\int_set:Nn \l_tmpa_int { 10 }
\int_set:Nn \l_tmpb_int { 4 }

\int_set:Nn \l_tmpc_int {
    \l_tmpa_int - \l_tmpb_int }

\int_set_eq:NN \l_tmpa_int
    \l_tmpb_int
\int_set_eq:NN \l_tmpb_int
    \l_tmpc_int

\int_use:N \l_tmpa_int; ~
    \int_use:N \l_tmpb_int
\ExplSyntaxOff
```

4; 6

이와 같이 하여야 합니다. python의 문법이 너무 간단해서 생기는 문제이고 옛날 언어들에서는 다 이렇게 했지요.

숙제에 대해서는 따로 할 말이 없습니다. python 코드를 충실히 잘 번역할 수 있었던 것으로 봅니다.

Nova de Hi.