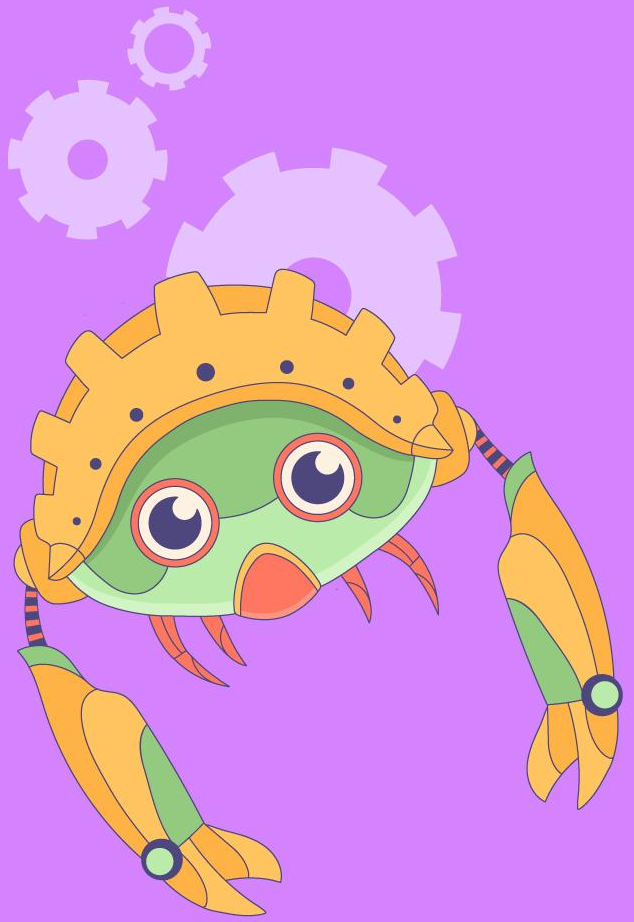
Alessandro Cerruti

Software Engineer @ Prima

Parsing integers with SIMD



What is SIMD?

- set of instructions that operates on vector registers
- 128/256/512 bit registers, interpreted as “cells”
  - e.g. 16 u8, 8 u16, 4 u32, …
- invoked via special functions (intrinsics)
  - directly map to underlying assembly
- difficult to port between architectures
  - different instructions
  - different performance profiles
  - content warning: SSE and x86 only

PADD (addition)

```
src1 = {1, 1, 1, 1, 1, 1, 1, 1}
src2 = {2, 2, 2, 2, 2, 2, 2, 2}

dst =  {3, 3, 3, 3, 3, 3, 3, 3}
```

Sums 8/16/32/64 bit “packed” integers in the two source registers.

The destination register should be interpreted as packed 8/16/32/64 bit integers each containing the sum of the source register packed integers in that position.

Intrinsic: **_mm_add_epi8**

PCMPGT (compare)

Compares the packed integers in the source registers.
The destination register will contain all 1s in the position where the first is greater than the second and all zeros otherwise.

Intrinsic: **_mm_cmpgt_epi8**

```
src1 = {1, 1, 1, 1, 2, 2, 2, 2}
src2 = {2, 2, 2, 2, 1, 1, 1, 1}

dst =  {
    0, 0, 0, 0,
    255, 255, 255, 255
}
```

MOVMSK (movemask)

```
src = {
    255, 255, 255, 255,
    0, 0, 0, 0
}

dst =  11110000
```

Returns a 64 bit integer with bits set to:
- 1 where the most significant bit of each corresponding packed 8 bit integer is set to 1
- 0 otherwise

Useful when we need to switch “back” from vector to scalar code.

Intrinsic: **_mm_movemask_epi8**

PMADD (multiply-add)

Multiplies packed integers the source registers, then sums each adjacent pair (first with second, third with fourth and so on).

The destination register should be interpreted as containing packed integers “one size larger” e.g. a PMADD of 8 16 bit integers will result in 4 32 bit integers.

Intrinsic: **_mm_madd(ubs)_epi16**

```
src1 = {3, 3, 3, 3, 3, 3, 3, 3}
src2 = {2, 2, 2, 2, 2, 2, 2, 2}

dst (intermediate) = {
    6, 6, 6, 6, 6, 6, 6, 6
}

dst =  {12, 12, 12, 12}
```

PSHUF (shuffle)

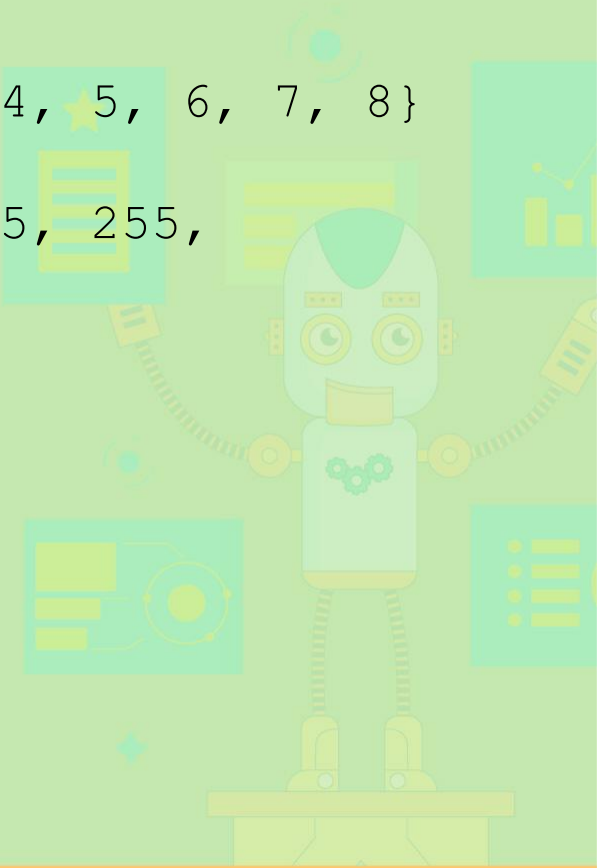
```
src = {1, 2, 3, 4, 5, 6, 7, 8}
mask = {
    255, 255, 255, 255,
    0, 1, 2, 3
}

dst =  {
    0, 0, 0, 0,
    1, 2, 3, 4
}
```

The destination register will contain:

- 0 if the MSB of the packed integer in the mask at that position is set
- the packed integer in src at the position of the packed integer in the mask otherwise

Intrinsic: **_mm_shuffle_epi8**



Shuffle tricks: bytewise shift (right)

```
fn shift_right_8x16(x: __m128i, amount: usize) -> __m128i {
    static SHUFFLE_LUT: [i8; 32] = ...;

    let shuffle = unsafe {
        _mm_lddqu_si128(SHUFFLE_LUT.get_unchecked(amount)
        as *const i8 as *const __m128i) };

    _mm_shuffle_epi8(x, shuffle)
}
```

Shuffle tricks: bytewise shift (right)

```
fn shift_right_8x16(x: __m128i, amount: usize) -> __m128i {
  static SHUFFLE_LUT: [i8; 32] = {
       0, 1, 2, 3, 4, 5, 6, 7,
       8, 9, 10, 11, 12, 13, 14, 15,
       -128, -128, -128, -128, -128, -128, -128, -128,
       -128, -128, -128, -128, -128, -128, -128, -128,
  };
  ...
}
```

Actually parsing integers!

We have a slice of variable length starting with 0 or more digit characters, possibly containing many leading zeros, possibly containing invalid data at the end.
We want to return an u64 integer corresponding to the base 10 interpretation of those digits AND the amount of bytes we consumed to do so.

Actually parsing integers!

The obvious approach

```
let mut result: usize = 0;
for (index, byte) in x.into_iter().enumerate() {
    let char = byte.wrapping_sub('0' as u8);
    if char > 9 { return result; }

    result = result * 10 + (char as usize);
}

return result;
```

Steps. 1 byte at a time

- load the byte in a register
- subtract ASCII zero
- if it’s not a digit: stop
- if it’s a digit: multiply the total by 10 and add the digit value to it

Steps. 16 bytes at a time

- load 16 bytes in a register (assume the string is 16 bytes long)
- subtract ASCII zeroes
- remove trailing data
- multiply each digit by its corresponding power of 10 (e.g. if the number is 5 digits long the first digit must be multiplied by 10^5, the second by 10^4, …) and sum the results

Steps. 16 bytes at a time

- **load 16 bytes in a register (assume the string is 16 bytes long)**
- subtract ASCII zeroes
- remove trailing data
- multiply each digit by its corresponding power of 10 (e.g. if the number is 5 digits long the first digit must be multiplied by 10^5, the second by 10^4, …)

Actually parsing integers!

16 bytes at a time: load

```
pub fn parse_16_bytes(x: &[i8]) -> ParseResult {
    // WARNING: most significant digit == least significant byte
    let mut chunk = unsafe {
        _mm_loadu_si128(x.as_ptr() as *const __m128i)
    };

    ...
}
```

Steps. 16 bytes at a time

- load 16 bytes in a register (assume the string is 16 bytes long)
- **subtract ASCII zeroes**
- remove trailing data
- multiply each digit by its corresponding power of 10 (e.g. if the number is 5 digits long the first digit must be multiplied by 10^5, the second by 10^4, …)

Actually parsing integers!

16 bytes at a time: subtract ASCII zeroes

```
pub fn parse_16_bytes(x: &[i8]) -> ParseResult {
    ...

    let ascii_zeros = _mm_set1_epi8('0' as i8);
    chunk = _mm_sub_epi8(chunk, ascii_zeros);

    ...
}
```

Steps. 16 bytes at a time

- load 16 bytes in a register (assume the string is 16 bytes long)
- subtract ASCII zeroes
- **remove trailing data**
- multiply each digit by its corresponding power of 10 (e.g. if the number is 5 digits long the first digit must be multiplied by 10^5, the second by 10^4, …)

16 bytes at a time: remove trailing data (building the mask)

```
pub fn parse_16_bytes(x: &[i8]) -> ParseResult {
    ...

    let wrap = _mm_set1_epi8(-128);
    chunk = _mm_sub_epi8(chunk, wrap);
    let mut nine_bytes = _mm_sub_epi8(_mm_set1_epi8(9), wrap);

    let byte_mask = _mm_cmpgt_epi8(chunk, nine_bytes);

    ...
}
```

Actually parsing integers!

16 bytes at a time: remove trailing data (using the mask)

```rust
pub fn parse_16_bytes(x: &[i8]) -> ParseResult {
    ...

    let bitmask = _mm_movemask_epi8(byte_mask) | 0x10000;

    let digit_count = bitmask.trailing_zeros();
    let non_digit_count = 16 - digit_count;

    chunk = shift_left_8x16(chunk, non_digit_count);

    ...
}
```

Steps. 16 bytes at a time

- load 16 bytes in a register (assume the string is 16 bytes long)
- subtract ASCII zeroes
- remove trailing data
- **multiply each digit by its corresponding power of 10** (e.g. if the number is 5 digits long the first digit must be multiplied by 10^5, the second by 10^4, …)

Actually parsing integers!

16 bytes at a time: multiply by powers of 10

```rust
pub fn parse_16_bytes(x: &[i8]) -> ParseResult {
    ...

    // for each group of two digits we multiply by either 1 or 10
    // and sum each pair of results
    // e.g. 9 | 2 | 1 | 5 becomes 92 | 15
    let tens = _mm_set2_epi8(1, 10);
    chunk = _mm_maddubs_epi16(chunk, tens);

    ...
}
```

Actually parsing integers!

16 bytes at a time: multiply by powers of 10

```rust
pub fn parse_16_bytes(x: &[i8]) -> ParseResult {
    ...

    // again with each group of four digits
    // e.g. 92 | 15 becomes 0 | 9215 (still fits in a short)
    let hundreds = _mm_set2_epi16(100, 1);
    chunk = _mm_madd_epi16(chunk, hundreds);

    ...
}
```

Actually parsing integers!

16 bytes at a time: multiply by powers of 10

```
pub fn parse_16_bytes(x: &[i8]) -> ParseResult {
    ...

    chunk = _mm_packus_epi32(chunk, chunk);

    // again with groups of eight digits
    let ten_thousands = _mm_set_epi16(0, 0, 0, 0, 1, 10000, 1, 10000);
    chunk = _mm_madd_epi16(chunk, ten_thousands);

    ...
}
```

Actually parsing integers!

16 bytes at a time: multiply by powers of 10

```rust
pub fn parse_16_bytes(x: &[i8]) -> ParseResult {
    ...

    let high_32: i64 = _mm_cvtsi128_si64(chunk) >> 32;
    let low_32: i64 = _mm_cvtsi128_si64(chunk) & 0xffffffff;
    let value = high_32 + (low_32 * 100000000)

    ParseResult {value: value, len: digit_count}
}
```

Making it work for length != 16

Problems:

- load past the end of the slice: segfault
- consume valid digits past the end of the slice
  - e.g. slice containing {1, 2, 3, 4} sharing memory with a slice containing {1, 2, 3, 4, 5}

Making it work for length != 16

Problems:

- **load past the end of the slice: segfault**
- consume valid digits past the end of the slice
  - e.g. slice containing {1, 2, 3, 4} sharing memory with a slice containing {1, 2, 3, 4, 5}

- Problem: load past the end of the slice

Solution: always load 16 bytes, but start at the previous address multiple of 16, then shift away the garbage!

The address is guaranteed to be in the same memory page, thus not triggering segfaults.

Actually parsing integers!

Problem: load past the end of the slice

```rust
pub fn parse(x: &[i8]) -> ParseResult {
    ...

    let mut str = x.as_ptr();
    let offset = str.addr() & (0b1111 as usize);

    let mut chunk = _mm_load_si128(str.sub(offset) as *const __m128i);
    chunk = shift_right_8x16(chunk, offset);

    ...
}
```

Making it work for length != 16

Problems:

- load past the end of the slice: segfault
- **consume valid digits past the end of the slice**
  - e.g. slice containing {1, 2, 3, 4} sharing memory with a slice containing {1, 2, 3, 4, 5}

Actually parsing integers!

- Problem: consume valid digits past the end of the slice

Solution: mask them away before parsing!

Problem: consume valid digits past the end of the slice

```
fn parse_last_chars(input: __m128i, len: i8) -> ParseResult {
    let indices = _mm_set_epi8(
         15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0);
    let len_mask = _mm_set1_epi8(len);
    let is_before_end_mask = _mm_cmpgt_epi8(len_mask, indices);

    let chunk = _mm_and_si128(input, is_before_end_mask);

    return parse_16_chars(chunk);
}
```

Actually parsing integers!

High level view

```
pub fn parse(x: &[i8]) -> ParseResult {
    if x.len() < 7 { /* for loop: parse 1 byte at a time */}

    let (mut chunk, offset) = align_pointer_to_16_and_load(x);
    chunk = mask_leading_garbage(loaded_chunk, offset);

    let result = parse_last_chars(chunk, 16.min(x.len()));
    if (result.len() != 16 - offset) { return result; }

    ...
}
```

Actually parsing integers!

High level view

```
pub fn parse(x: &[i8]) -> ParseResult {
    ...

    while x.len() - result.index > 16 {
        chunk = load_vector(x, result.index);
        result.check_overflow_and_update(parse_16_chars(chunk));
        if (result.overflow) { /* return error */ }
    }

    ...
}
```

Actually parsing integers!

High level view

```
pub fn parse(x: &[i8]) -> ParseResult {
    ...

    chunk = load_vector(x, result.index);
    result.check_overflow_and_update(parse_16_chars(chunk));
    if (result.overflow) { /* return error */ }

    return result;
}
```

Credits

- Alexander Kondratskiy AKA Kholdstare
  - https://kholdstare.github.io/technical/2020/05/26/faster-integer-parsing.html
- Wojciech Muła
  - http://0x80.pl/notesen/2018-04-19-simd-parsing-int-sequences.html
- Steven Hoving (shuffle shift trick)
  - https://github.com/KholdStare/qnd-integer-parsing-experiments/issues/3

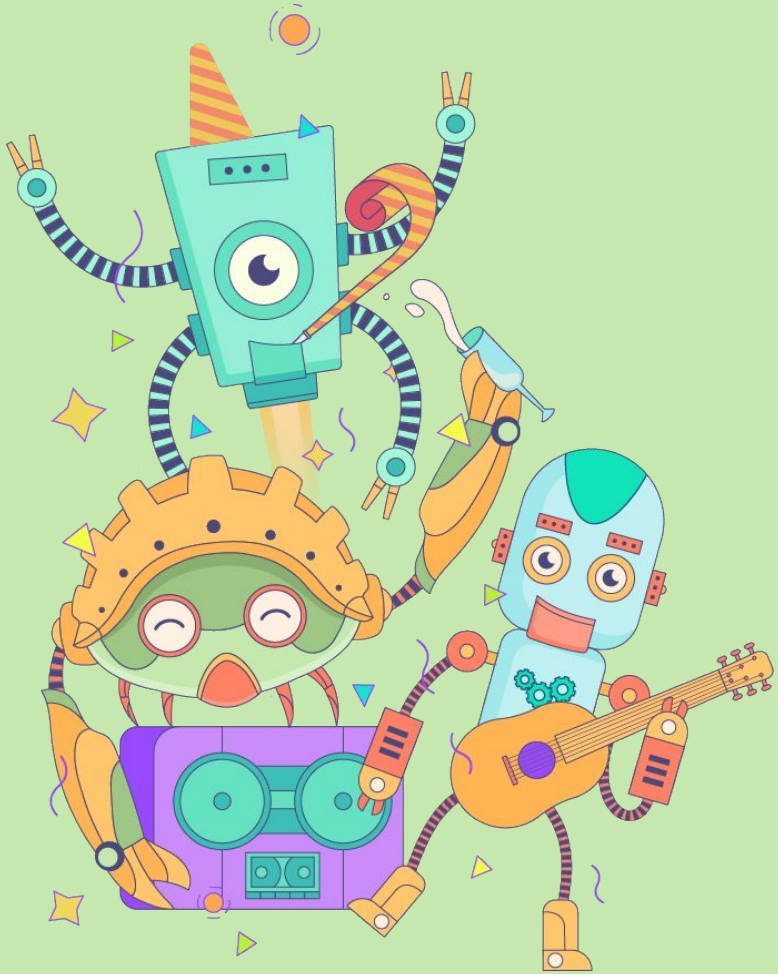
Benchmarks

- SIMD is faster when input.len() > 6, around 2.5x faster at length 21 while parsing 2^64
- Far away from the expected speedup (16x), parse_16_integers is fast, the mask tricks slow it down (comparisons and movemasks are not free!)
- Speedup also comes from avoiding branches:
  - obvious approach: ~2 branches every byte (i < len, checked add+mul)
  - SIMD approach: ~3 branches every 16 bytes (i < len, checked add, checked mul)

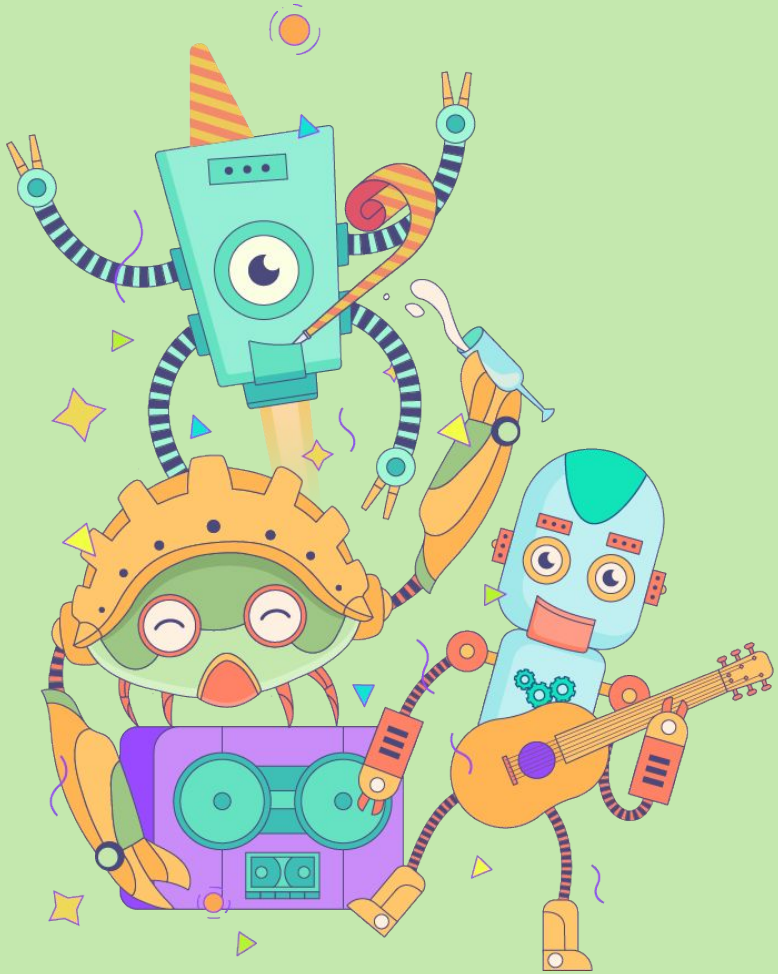
Maintainability

- Not good.
- Also, non-portable.

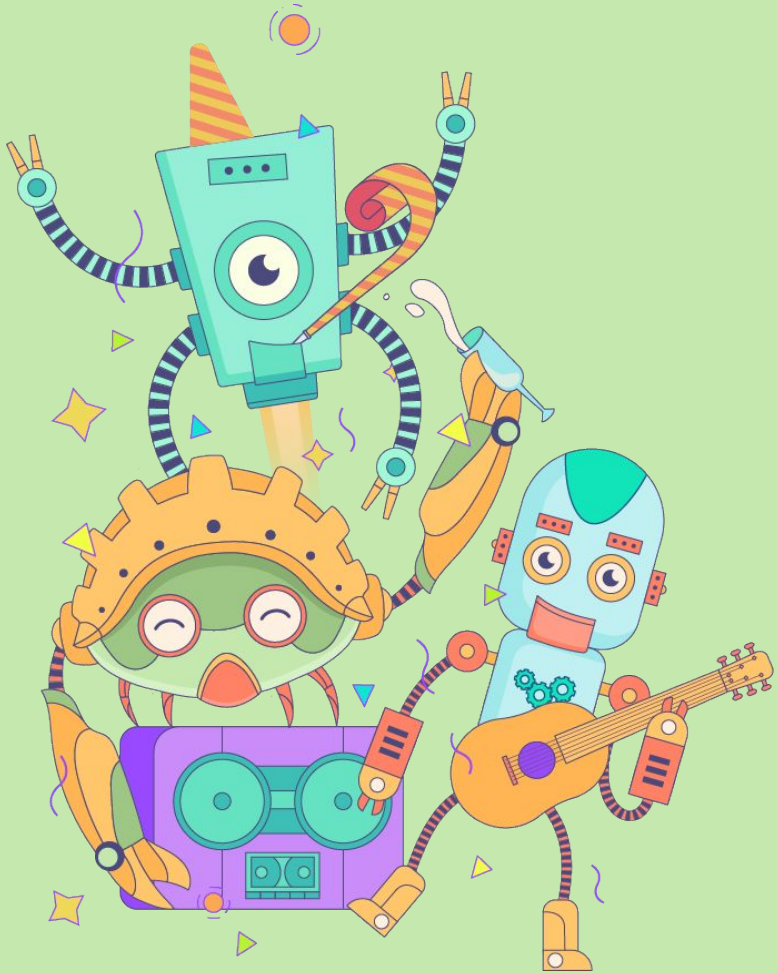
Was it worth it?



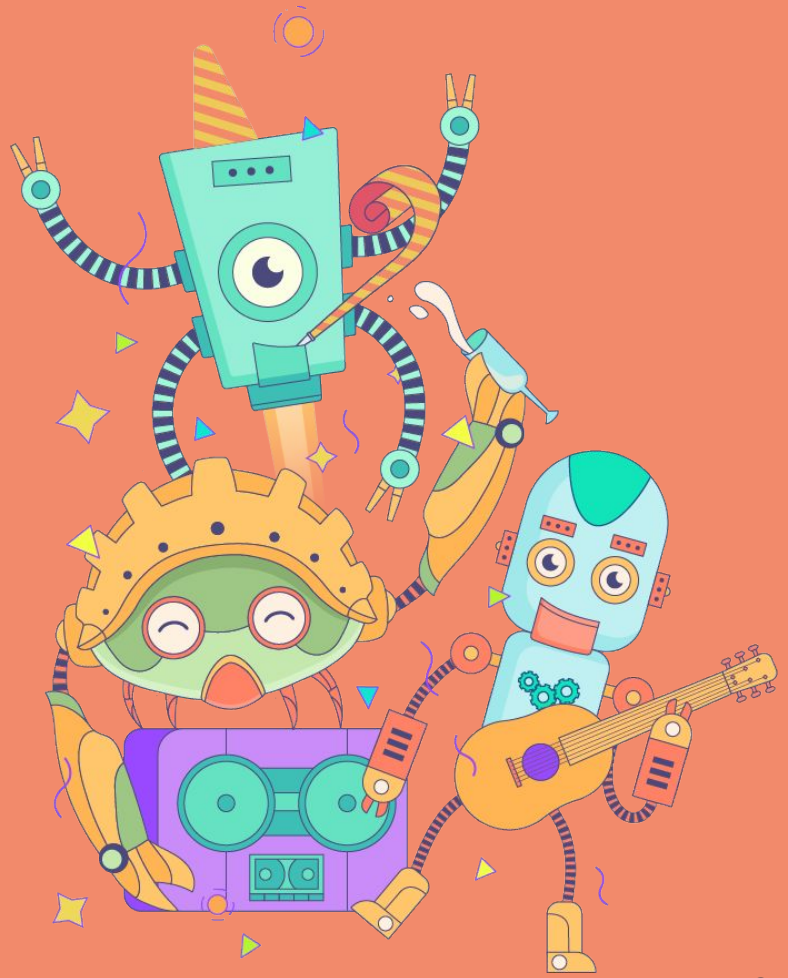
Maybe…



… at least I had fun!

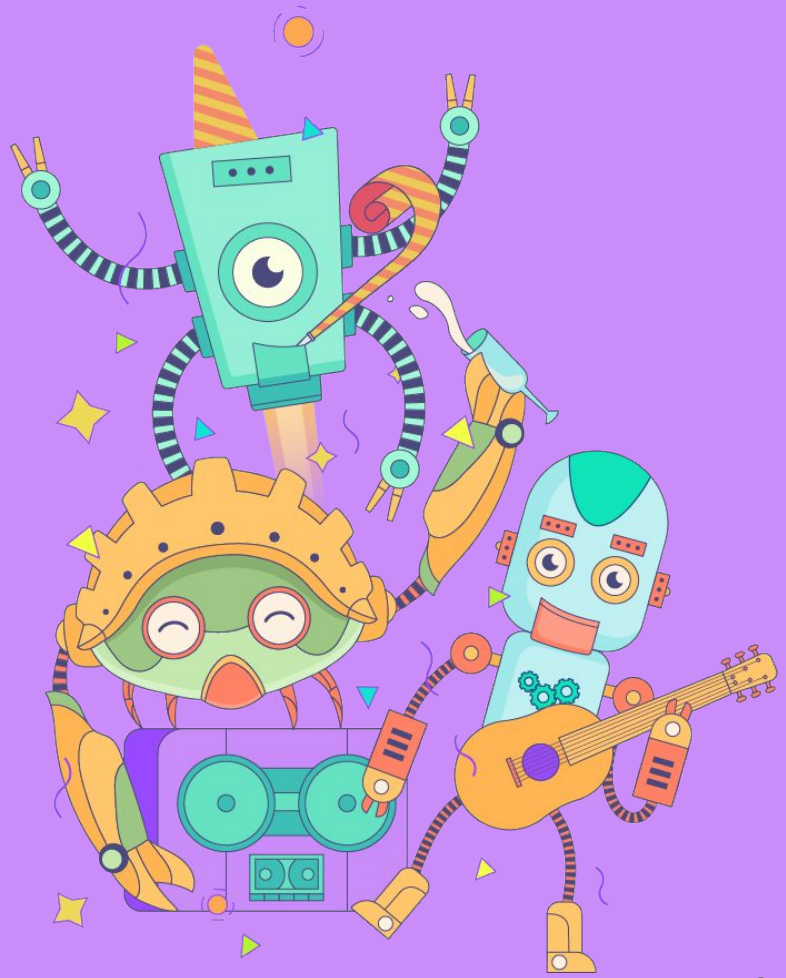


Question time!



Thank you!

code:
https://github.com/chrooti/parsing_integers_rs



Alessandro Cerruti

whisper@chrooti.org