Compliler Construction: Infinite precision signed integers

Klaas Kliffen, Rik Schaaf

1 Problem description

The aim of this lab is to give a library to use with a compiler for a language with infinite precision integers. Furthermore the following operations on these integers must be implemented:

- Creating an integer from string
- Printing an integer to stdout
- Adding two integers
- Substracting two integers
- Multiplying two integers
- Dividing an integer by another integer
- Apply a modulo on an integer
- Apply a power to an integer

2 Problem analysis

First there must be a datastructure to define the Big integers. A logical solution is to store each digit of the number in an array as integers in C are not infinite. To keep track of the sign, another numeric variable can be used. For exanple a -1 for negative integers and 1 for positive. When using an array representation, a length too can be usefull, as C does provide a length function for integers.

2.1 Creating from string

2.2 Printing an integer

2.3 Adding

2.4 Subtracting

2.5 Multiplying

There are multiple algorithms to choose from when multiplying. The standard long multiplication algorithm which runs in $O(n^2)$. As there can be a lot of mulitplications in a program, a lower order function is prefered. An example discussed here is the karatsuba algorithm which uses a divide and conquer method. In pseudo code it looks like this:

```
procedure karatsuba(num1, num2)
  if (num1 < 10) or (num2 < 10)
    return num1*num2
  /* calculates the size of the numbers */
  m = max(size_base10(num1), size_base10(num2))
  m2 = m/2
  /* split the digit sequences about the middle */
  high1, low1 = split_at(num1, m2)
  high2, low2 = split_at(num2, m2)
  /* 3 calls made to numbers approximately half the size */
  z0 = karatsuba(low1,low2)
  z1 = karatsuba((low1+high1),(low2+high2))
  z2 = karatsuba(high1,high2)
  return (z2*10^(2*m2))+((z1-z2-z0)*10^(m2))+(z0)
```

Using this algorithm you perform three multiplications of integers with approximately half of the number of digits of large integers. This should yield a complexity (for sufficiently high n) of $\Theta(n^{\log_2(3)})$.

2.6 Division

There are not many efficient division algorithms so the algorithm chosen here is the long division method.

2.7 Modulo

The module operation is the counterpart of the devision method, so the same method for the long division is used here.

2.8 Power function

The straigforward method of exponeniation of multiplying k times for n^k can be easily optimised. For the power operation the exponanitiation by squaring is used. This vastly reduces the number of multiplying operations and runs in $O((n\log(x))^{\log_2(3)}$ when using the karatsuba algorithm for mulitplication.

3 Design of solution

The chosen datastructure has been implemented as follows:

```
struct EGCLint {
  uint32_t *digits; /* the value for each of the digits, big
     endian */
  unsigned long length; /* maximum value is approx.
     10^4294967295, not infinite, but close enough. */
  int sign; /* wheter it is positive, of negative */
};
typedef struct EGCLint Integer;
```

3.1 Creating from string

3.2 Printing an integer

3.3 Adding

3.4 Subtracting

3.5 Multiplying

3.6 Division

3.7 Modulo

3.8 Power function

4 Evaluation

The performance of the algorithms used is adequate. One thing that could be improved upon is the allocation of memory. At runtime a lot of memory is allocated, reallocated and copied over. Instead of keeping the digit arrays at the exact needed length, a function can be used to double the amount digits when needed. Doubling the length of the array makes sure there are less memory allocations and the digit array is always large enough to contain all digits.

5 Program code