Final Language Report: Prime

Alexander Liebeskind (al3853), Project Manager
Nikhil Mehta (nm3077), Language Guru
Pedro B T Santos (pb2751), Tester
Thomas Tran (tkt2120), Systems Architect

Programming Languages and Translators, Spring 2021
1 References

We cite the following sources and thank the respective owners:

1. https://gmplib.org/manual/ The GMP library for large number functions
2. Professor Dorian Goldfeld slides for numbers and presentation snapshots

2 Introduction

2.1 Overview

PRIME is a programming language specifically created for the implementation of cryptography algorithms. The main features of PRIME aim to facilitate the implementation of encryption and decryption schemes. Since modular arithmetic, large numbers, and elliptic curve processing are common in cryptography, the main types and operators in PRIME address these topics. PRIME also includes more general basic features for ease of usage.

2.2 Background

The basis for all modern cryptography is mathematical trapdoor functions. That is, mathematical operations that are easy to do in one direction but very difficult to do in the backwards (inverse) direction. Modular arithmetic using large primes is at the heart of many trapdoor functions used in industry today so our language will focus on making this kind of arithmetic easy to implement. Additionally, the recent advent and widespread adoption of elliptic curve cryptography is changing how we encrypt data. Trusted protocols are now being re-implemented more securely with the use of modular elliptic curves. So, building off of our use of big numbers our language will make elliptic curve operations a core feature.

Modular Arithmetic

Modular arithmetic, sometimes referred to as clock arithmetic, is a mathematical system for integers that looks at the remainder of standard mathematical operations for a given modulus. For example, 13 divided by 6 is equal to 2 reminder 1. In modular arithmetic we only care about the remainder so we say 13 is congruent to 1 modulo 6 or 1 mod 6 for short. The notation for this is:

\[ 13 \equiv 1 \pmod{6} \]

We can then build up from here to denote entire mathematical expressions as being modded by a number, such as:

\[ 2^4 \equiv x \pmod{5} \]

In this case we find that \( x \) is congruent to 1. Modular arithmetic is really easy to compute in one direction. But take the expression:

\[ 3^x \equiv 4 \pmod{7} \]

It’s much harder to find \( x \) now. When we replace these single digit integers with massive numbers, say around 200 digits long, this problem becomes impossible even for a computer to solve.

Elliptic Curves

An elliptic curve is defined as the function

\[ y^2 = x^3 + ax + b \]
We then define the operation point addition. For points $p, q$, their sum $r = p + q$ is taken as the third point of intersection on the curve with the line formed by points $p$ and $q$ and then reflected about the $x$-axis. Lines that are tangent to the curve count as intersecting the line twice at the point of tangency. In elliptic curve cryptography we define a curve over a modulus.

$$y^2 = x^3 + ax + b \pmod{c}$$

The instead of working with an infinite set of points over the $\mathbb{R} \times \mathbb{R}$ we now have a finite set of integer points. When $c$ is a prime number this set forms a finite abelian group, which is therefore cyclic over the point addition with a generator point $g$. That is a complicated way is saying a few things. First, for any points $p$ and $q$, $p + q = q + p$. Second, there exists a point $g$ where the repeated addition of $g$ onto itself will yield every point on the modular elliptic curve. Third, since a modular elliptic curve forms a group, there is an identity element. This element is the point at infinity. The point at infinity added to any point $p$ on the curve will yield $p$. Lastly, for every $p$ on the curve, there exists an inverse denoted $-p$ such that $p + -p$ is equal to the point at infinity.

With the use of computers and calculators, computing the repeated sum of a point $p$ by $k$ number of times, denoted $q = k \times p$ is easy. But, if given only points $q$ and $p$, it can become impossibly difficult to find $k$. This is how trapdoor functions are built using elliptic curves and where they become useful in cryptographic contexts.

2.3 Related Work

Due to syntactical and functional similarities between PRIME and C, the C Reference Manual [https://www.bell-labs.com/usr/dmr/www/cman.pdf] is a helpful resource in understanding PRIME and its underlying mechanics.
2.4 Goals

2.4.1 Big Number and Modular Arithmetic

The foremost goal of PRIME is to implement features related to modular arithmetic of big numbers. To accomplish this we wrapped the GMP GNU Big Number Library with our language. This library is commonly used in cryptography applications today, however, the GMP GNU library is unintuitive to use and clunky to write. One of PRIME’s largest benefits is the ease in which a user can declare and then work with a large number. While libraries like GMP GNU exclusively rely on the use of verbose function calls, PRIME utilizes built in operators and a simplified syntax. Building off of big numbers, one of PRIME’s core functionalities is the support of elliptic curves and elliptic curve arithmetic. Elliptic curves are not built into standard big number libraries. So a function like adding two points on an elliptic curve which took around 200 lines to write in the GMP GNU big number library has been reduced to a single character operator in PRIME. This allows for commonly used cryptosystems and protocols to be implemented concisely in just a few lines of code.

2.4.2 Ease of Usage and Readability

As eluded to above, PRIME seeks to make arithmetic with big numbers more intuitive. As a result, we sought to make our arithmetic operators and expressions closely resemble standard mathematical notation so a user with little experience could start programming in PRIME on the fly. Below are a couple comparisons between mathematical notation and their PRIME implementations.

*Multiplicative inverse of a modulo b*

Mathematical notation: \( a' \pmod{b} \)

```
    a'b /* PRIME implementation of Multiplicative inverse of a modulo b */
```

So, in addition to \( a \) raised to \( b \) modulo \( c \)

Mathematical notation: \( a^b \pmod{c} \)

```
    a ^ b @ c /* PRIME implementation of a raised to b modulo c */
```

So, in addition to allowing for concise implementations cryptographic functions, PRIME makes code intelligible for other users. This is made possible by complex algorithms and functions that we wrote or wrapped behind the scenes. If a user chose they would have to learn many of these algorithms and implement them themselves.

3 Language Tutorial

For installation, unzip the tar.gz file, then run make to build and run the regression test suite. Run this in a docker container that corresponds to Professor Edward’s microc docker hub container. Make will install the gmp library via apt for you.

The first thing to write is the main function as this will be the entrypoint to the program. This should return 0 if all is successful as is convention like so:

```
int main()
{
    return 0;
}
```

To then start adding some interesting functionality, one can dive in and start using large integers (lints).

```
/* Inside a function */
lint a;
/* This or larger and terminated by an l */
a = 1230981230812301231231231231231231231231231231231231231231231231231231
/* Now you can print them and their operations */
printf("a\n");
printf("a\*21\n"); /* note that these must be the same type */
```

One can then use these lints to declare curves where our points will then lie.
Now to start getting to more advanced functionality, we combine lints and curves to create points.

```plaintext
/* At very start of function */
pt p;
/* ...

....sometime later */
/* Following on from code in previous box */
p = [a, b] & c;
printpt(p);
printpt(p * p);  /* Point addition on elliptic curve */
```

Now for familiar cryptography applications you may want to have some strings there too

```plaintext
string s;
lint a;
s = "Hello World";
\* turns string into its numerical equivalent */
printl(a);
prints(\*decodes the string */
```

Then you can wrap some logical statements together into a function:

```plaintext
/* function returning a new lint */
lint newLint(string s)
{
\* return encode(s);
}
```

Several other code examples are presented throughout the course of the report.

## 4 Language Manual

### 4.1 Types

Types in PRIME are similar to that of C and C++ with a few others that are particularly useful for cryptography operations. Note that in the below CFG, the term constant is used to refer to fundamental literal types, which are then used to create all of the basic types in PRIME.

#### 4.1.1 Initialization

In the absence of variable assignment, types are handled as literals (here 'constants', in the CFG). Initializing variables in PRIME is similar across all types, requiring the type of the variable to be declared before initialization with the assignment operator. All declaration must occur on its own lines, before any variable is set to a value.

**Examples:**

```plaintext
int x;  /* Declaration x to be of type int*/
x = 4;  /* Assign 4 to x */
int y;  /* Parsing error! */
```

#### 4.1.2 int

A int can take signed 32-bit integer values ranging from −2,147,483,648 to 2,147,483,647. ints are declared without an original value. After initialization, int can then be set to a value. The value of the
int may be reassigned, set using an expression, or set to the value of another int variable. The value of an uninitialized int is undefined, and is a garbage number.

Examples:

```
int foo;
int bar;
foo = 4; /* Assign 4 to x */
foo = 6 * 5; /* Reinitialize foo */
bar = foo; /* Assign bar to foo's value */
print(bar); /* prints 30 */
```

4.1.3 string

A string is a datatype meant to hold a sequence of ASCII characters. We denote string assignment by using quotations for literal sequences of characters. strings can be declared with or without an original value. The value of an uninitialized string is undefined, and is a garbage string. After initialization, a string can be reassigned, set using a string literal, or set to the value of another string variable.

Examples:

```
int foo;
int bar;
foo = "test1";
foo = "test2";
bar = foo; /* Assign bar to foo's value */
printl(bar); /* prints test2 */
```

4.1.4 lint

lints are the primary type of the PRIME language and serve as the building blocks for all of our cryptographic types and the operands in much of our arithmetic. A lint or large integer is a positive or negative integer with minimal restraints on size. A lint can hold up to $2^{32}$ bits to conform with current state-of-the-art cryptographic security requirements. Large integer types are usually used to hold large primes for later use in computationally expensive products or exponents in practice.

Similar to other types, lint s must be declared first, and then initialized. When initializing, digits are followed by an 'l' to distinguish between integers and large integers. ints may also be cast into lint s using tolint(), but not vice versa.

Examples:

```
lint foo;
lint bar;
int x;
foo = 100271002710027100271002710027 l;
x = 100;
bar = tolint(x);
printl(bar); /* prints 100 */
```

4.1.5 curve

curve defines a third degree univariate polynomial function with respect to a positive lint modulus. Within elliptic curve cryptography, all curves are defined as

$$y^2 \equiv x^3 + c_1 x + c_2 \pmod{n}$$

Thus, curves in PRIME only need to define in c1 and c2 and a base modulus n. In PRIME curve literals are written encased in brackets. First, The two coefficients are written inside parenthesis and separated
by a comma. This is followed by a colon and then the modulus. More simply put, the curve above can be written as

\[ \[(c_1, c_2) : n]\]

in PRIME. Both the coefficients and the modulus must be of type lint. curves are immutable so once initialized, none of their values may be changed. However, curves can be reassigned. curves are used as a building block for points and thus have no arithmetic nor relational operators defined for them.

Much of modern day elliptic curve cryptography is based on elliptic curves being defined over a prime modulus. Primality testing is an entire sub-field of mathematics and for large numbers, most tests can only give high likelihoods of numbers being prime, not a guarantee. So onus is on the user to pass in a valid prime number.

**Examples:**

```
curve bar;
bar = [(5l, 12l) : 13l];
printc(bar); /* prints \[(5, 12) : 13\]; */
bar = [(3l, 7l) : 23l] /* bar reassigned */
```

4.1.6 **pt**

A pt is a datatype meant to represent a point on an elliptic curve. When points are initialized, they must be defined with regards to a curve. Additionally, they may also specify two coordinates of lint types, representing the \(x\) and \(y\) coordinates.

When a pt is initialized with a set of immutable coordinates, a value must be given for every dimension. These values must be enclosed in brackets and separated by commas. A pt declared without 2 lints is undefined. Furthermore, they must be defined with respect to a curve using &. See examples below.

The coordinates within a pt can be accessed using pt.\(x\) and pt.\(y\); however, the coordinates cannot be reinitialized. Since pts are specific to an elliptic curve, a pt’s curve may not be changed. However, a pt may be reassigned.

A given modular elliptic curve has a set of valid points associated with it. Building safe elliptic curves and generator points for those curves is a feat in it of itself so the onus is on the user to use valid points for a given elliptic curve. This includes using coordinates that are non-negative lints less than the elliptic curve modulus. The one exception to this rule is the point \((-1, -1)\) which is written as \([-1l, -1l]\) & crv for a given curve crv. This represents the point at infinity. The point at infinity exists with respect to any curve and is the identity element for point arithmetic.

**Examples:**

```
curve bar;
pt foo;
bar = [(5l, 12l) : 13l];
foo = [2l, 3l] & bar; /*x = 2, y = 2, defined on the bar curve*/
printl(foo.x); /* prints 2 */
printl(foo.y); /* prints 3 */
printpt(foo); /* prints [2, 3] & [(5, 12) : 13] */
foo = [-1l, -1l] & [(7l, 2l) : 23l];
printpt(foo) /* prints [-1, -1] & [(7, 2) : 23] */
foo.x = 4 /* Parsing error */
```

4.2 **Operators**

4.2.1 **Unary Operators**

Unary operators include ! and -. Unary operators take precedence above all other mathematical operators.
Not: 

\[ ! \text{expression} \]

The logical negation operator obtains the logical opposite of a value. It works on \texttt{ints} and \texttt{lints}. Since there are no booleans in \texttt{PRIME}, true and false may be represented by 0 and non-zero integers respectfully. If the expression evaluates to a non-zero integer, \texttt{! expression} will return a 0. If the expression evaluates to zero, \texttt{! expression} will return 1. The same applies for \texttt{lints}, returning another \texttt{lint}.

\textbf{Examples:}

```c
int foo;
lint bar;
lint baz;
foo = 1;
bar = 01;
baz = !bar;
print(!foo);  // prints 0 */
if (baz) {
  // evaluates to true */
  print("baz is true");
}
```

Negative: 

\[ -\text{expression} \]

The negative unary operator is used to obtain the mathematical opposite of a value. It works on \texttt{ints}, \texttt{lints}, and \texttt{pts}. When performed on an \texttt{int} or \texttt{lint}, the opposite value will be returned. For example, 1 becomes -1 and vice versa. When performed on an \texttt{pt}, it is the additive inverse of the \texttt{pt} with respect to its elliptic curve. The sum of a point and its additive inverse is always the point at infinity (-1, -1), which serves as the identity element in modular elliptic curves. The point at infinity can be written in \texttt{PRIME} as \([\[-1l, -1l\] & crv\] for a given curve \texttt{crv}.

\textbf{Examples:}

```c
int foo;
int bar;
pt baz;
foo = 1;
bar = -foo;
print(bar);  // prints -1 */
baz = [21, 21] & [(51, 121) : 131];
printpt(-baz);  // prints additive inverse of foo
  // which is [21, 111] & [(51, 121) : 131] */
```

4.2.2 Multiplicative Operators

Multiplicative operators include multiplication, division, modulo, power, and the multiplicative invert operator. Multiplicative operators have a ranges of precedences which can be viewed in following section.

Multiplication: 

\[ \text{expression} \ast \text{expression} \]

The multiplication operator is used to obtain the product of two numbers. The multiplication operator functions on \texttt{ints}, \texttt{lints}, and \texttt{pts}. \texttt{ints} may only be multiplied by other expressions that evaluate to \texttt{ints}. Multiplication of \texttt{lints} by \texttt{lints} and \texttt{ints} by \texttt{ints} return their same type. When multiplication is performed on a \texttt{pt}, the other expression must be a positive \texttt{lint}. Multiplication of a \texttt{pt} \texttt{p} by \texttt{lint} \texttt{l} and vice versa acts as a repeated addition of \texttt{pt} \texttt{p} onto itself \texttt{l} times. If \texttt{l} is negative the behavior is undefined. \texttt{pts} may not be multiplied by other \texttt{pts}. \texttt{pt} multiplication returns a \texttt{pt}.

\textbf{Examples:}
Division: /

```
expression / expression
```

The division operator will be used to obtain the integer quotient of two integers. It will function on ints and lints. ints may only be divided by or divide expressions of type int. Similarly, lints may only be divided by or divide expressions of type lint. If the divisor, the right expression, does not divide the left expression, the quotient will be truncated. Division returns the same type as the two operands in the expression.

**Examples:**

```
lint bar;
int three;
int four;

bar = 1837641987193287l;
21474836591l/bar;  // is a lint */

three = 3;
four = 4;
three * four;  // is an int */

baz = [21, 21] & [(51, 121) : 131];
baz * 3l;  // evaluates to: baz + baz + baz + baz */
three * baz;  // same as line above */
```

Modulo: %

```
expression % expression
```

The modulo operator will be used to obtain the positive remainder from euclidean integer division. It will function on ints and lints. ints may only be modded by other ints and lints by other lints. The return type is the same as the type of the two operand expressions.

**Examples:**

```
lint foo;
lint bar;
int three;
int four;

foo = 21474836591l;
bar = 1837641987193287l;
bar%foo;  // is a lint */

three = 3;
four = 4;
four%three;  // evaluates to 1 as an int */
```

Exponent: /\

```
expression/\expression
```

The exponent operator will be used to compute the repeated multiplication of integer values. The exponent operator is written as forward slash immediately followed by a backslash. The base expression
must be of type \texttt{lint} and the exponent expression must be of type \texttt{int}. The exponent expression may negative but the output will truncate to zero. The return type is always a \texttt{lint}.

\textbf{Examples:}

```plaintext
lint foo;
int bar;
foo = 10000;
bar = 2;
bar / foo; /* is a lint */
```

\textbf{Multiplicative Inversion: `}

\textit{expression ` expression}

The multiplicative inversion operator is used to find the multiplicative inverse of the left expression modulo the right expression. A multiplicative inverse of \texttt{a(mod b)} is the smallest positive integer \texttt{c} such that \texttt{a * c} is congruent to \texttt{1(mod b)}. The multiplicative inversion operator is denoted with the backward apostrophe typically found just left of the '1' key on American keyboards. The multiplicative inverse is found efficiently using the extended euclidean algorithm. Both operand expressions must be of type \texttt{lint} and the resultant output value will also be a \texttt{lint}. Not every integer has a multiplicative inverse modulo another integer. If that is the case the output will be \texttt{0}. \texttt{0} is never the multiplicative inverse of any integer.

\textbf{Examples:}

```plaintext
lint a;
lint b;
lint c;
lint ;
a = 3l;
b = 7l;

/* c is equal to 5
 * 5*3 = 15
 * 15 is congruent to 1 (mod 7) */
c = a ` b;

printl(c); /* prints "5" */

d = 2l ` 4l /* 2 has no multiplicative inverse mod 4 */
printl(d); /* prints "0" */
```

\textbf{4.2.3 Additive Operators}

Additive operators include addition and subtraction and have lower precedence than multiplicative operators.

\textbf{Addition: +}

\textit{expression + expression}

The addition operator will be used to obtain integer and point sums. It will function on \texttt{ints}, \texttt{lints}, and \texttt{pts}. \texttt{ints} may only be added to \texttt{ints} and \texttt{lints} to \texttt{lints}.

\texttt{pt} addition is always defined with respect to a specific elliptic curve and so \texttt{pts} may only be added to other \texttt{pts} of the same curve. If two \texttt{pts} of different curves are added to one another the behavior is undefined. The identity element in elliptic curve point addition is the point at infinity. In our language we denote this point as \texttt{(-1,-1)}. In PRIME this is written as \texttt{[-1l, -1l] & crv} for a given curve \texttt{crv}. Any \texttt{pt} on curve \texttt{crv} added to the point at infinity will return itself. A \texttt{pt} added with its additive inverse will return the point at infinity.

Addition always returns the same type as the two operands in the expression.

\textbf{Examples:}

```plaintext
lint one;
lint two;
int three;
```
int four;
pt baz;
pt foo;

one = 2147483659l;
two = 1837641987193287l;
one + two; /* is a lint */
three = 3;
four = 4;
three + four; /* is an int */
baz = [2l, 2l] & [(5l, 12l) : 13l];
foo = -baz; /* set foo to the additive inverse of baz */
baz + baz; /* is a pt */
baz + [-1l, -1l] & [(5l, 12l) : 13l]; /* evaluates to baz */
baz + foo; /* evaluates to [-1l, -1l] & [(5l, 12l) : 13l] */

Subtraction: -

expression - expression

The subtraction operator is used to obtain integer differences. It will function on ints and lints. ints may only be subtracted with other ints and lints may only be subtracted from other lints. Subtraction returns the same type as the two operands in the expression.

Examples:

lint foo;
int three;

foo = 2147483659l;
foo - 1837641987193287l; /* is a lint */
three - 4; /* is an int */

4.2.4 Relational Operators

expression < expression
expression > expression
expression <= expression
expression >= expression

The relational operators (\(<\), \(\geq\), \(\leq\), \(\leq\)) denote less than, greater than, less than or equal to, and greater than or equal to, respectively. These operators compare two expressions of type int or lint and will return an integer value 1 (true) or 0 (false) based on a relational comparison in \(\mathbb{R}\). These relational operators are equivalent in precedence and have a lower precedence than multiplicative and additive operators.

Examples:

lint foo;
lint bar;
int three;

foo = 2147483659l;
bar = 1837641987193287l;
bar > foo; /* returns 1 as an int */
three = 3;
three <= -15; /* returns 0 as an int */
4.2.5 Equality Operators

\[
expression == expression \\
expression != expression
\]

The equality operators (==, !=) denote equals and not equals, respectively. These operators compare two expressions of type int, lint, or pt. The == operator returns an integer value 1 (true) if the two expressions are equivalent by value, and 0 (false) otherwise. != operator returns an integer value 1 (true) if the two expressions are not equivalent by value, and 0 (false) otherwise. Two pts on the different elliptic curves will never be equal to one another regardless of their coordinates. Two pts on the same elliptic curve may be equal if their respective coordinates are equal. These equality operators are equivalent in precedence to one another and have a lower precedence than relational operators. A lint may only be compared to another lint, ints only with other ints, and pts only with other pts. Examples:

```c
lint foo;
lint bar;
int three;
pt baz;

foo = 2147483659l;
bar = 1837641987193287l;
bar != foo; /* returns 1 as an int */
three = 3;
three == 3; /* returns 0 as an int */
baz = [2l, 2ll] & [(5l, 121l) : 13l];
baz == [2l, 2ll] & [(5l, 121l) : 13l]; /* returns 1 as an int */
baz != [2l, 2ll] & [(5l, 7l) : 13l]; /* returns 1 as an int */
```

4.2.6 Logical Operators

\[
expression && expression \\
expression || expression
\]

The logical operators (&&, ||) denote logical and and logical or, respectively. These operators compare two expressions, both of type int or lint. The && operator returns an integer value 1 (true) if the two expressions are nonzero, and 0 (false) otherwise. The || operator returns an integer value 0 (false) if the two expressions are zero, and 1 (true) otherwise. These logical operators are evaluated from left to right, equivalent in precedence and have a lower precedence than equality operators. A logical expression must only have types that are all ints or all lints. Examples:

```c
lint foo;
lint bar;
int three;

foo = 2147483659l;
bar = 0l;
bar && foo; /* returns 0 as an int */
three = 3;
three || 0 || 1; /* returns 1 as an int */
1 || (bar || foo) /* returns 1 as an int */
```

The equals sign is the assignment operator in our language. Once the right hand expression has been evaluated, it is stored in the variable. The expression to the left of the equals sign must be a variable of the same type as the expression. Assignment must happen in a separate statement to that of variable declaration. An assignment expression returns void. Examples:
4.2.7 Ternary Operators

In PRIME there are two ternary operators that can be used to form a single ternary operation. The ternary operators take precedence above all other arithmetic, relational, equality, and logical operators.

\[ \text{expression} \, ^{\wedge} \, \text{expression} \, @ \, \text{expression} \]

The ternary operation using the carrot and then the at symbol functions only on \textit{lint}s and is taken to mean value of the first expression raised to the power of the middle expression modulo the right expression. Or, more simply put, \(a^b \% c\) where \(a\) and \(b\) are the first two expressions respectively and \(c\) is the last expression. The return type of ternary operation is a \textit{lint}. The tenary operator is the only explicit way to raise a \textit{lint} to the power of another \textit{lint}.

\textit{Examples}:

\begin{verbatim}
lint a;
lint b;
lint c;
a = 123456789 l;
b = 987654321 l;
c = 231
a ^ b @ c /* mathematically the same as (a^b)%c */
\end{verbatim}

4.3 Precedence

Precedence in PRIME is defined as follows. If two operators of equal precedence are present, they will be parsed from left to right within a line and top to bottom if they include operators from different precedence levels as according to this table.

<table>
<thead>
<tr>
<th>1</th>
<th>()</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>. (Access)</td>
</tr>
<tr>
<td>3</td>
<td>^</td>
</tr>
<tr>
<td>4</td>
<td>@</td>
</tr>
<tr>
<td>5</td>
<td>&amp;</td>
</tr>
<tr>
<td>6</td>
<td>\</td>
</tr>
<tr>
<td>7</td>
<td>!, Unary Minus</td>
</tr>
<tr>
<td>8</td>
<td>*</td>
</tr>
<tr>
<td>9</td>
<td>+, - (Subtraction)</td>
</tr>
<tr>
<td>10</td>
<td>%</td>
</tr>
<tr>
<td>11</td>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
</tr>
<tr>
<td>12</td>
<td>==, !=</td>
</tr>
<tr>
<td>13</td>
<td>&amp;&amp;,</td>
</tr>
<tr>
<td>14</td>
<td>=</td>
</tr>
</tbody>
</table>

4.4 Lexical Conventions

4.4.1 Key Words

PRIME has several keywords that are reserved in the language to prevent ambiguity.
Type Related Keywords

The following are the types that exist in our language. We do not allow for users to define their own types. For cryptography uses these should be sufficient.

- int
- lint
- pt
- curve
- string

Statement Related Keywords

The following are the only keywords we have for specific statements.

- if
- else
- for
- while
- return

4.4.2 Variable Names

Variables must be named starting with a letter a through z lower or uppercase followed by any number of alphanumeric characters or underscores.

4.4.3 Comments

Comments in Prime are initiated using the character sequence /* and concluded using */. All text between the /* and the */ will be ignored by the compiler, including across lines. This commenting format applies to both single line and multiline comments.

4.5 Program Structure

The following structural elements dictate the control flow of a program in PRIME.

4.5.1 Conditionals

Conditionals in PRIME come in two forms, both of which begin with an if statement:

```plaintext
if (expression) statement
```

```plaintext
if (expression) statement else statement
```

In each of the above cases, the expression following the initial if statement is evaluated and if the value returned is nonzero, the following statement is executed. The else keyword allows for an alternative statement to be executed if the expression evaluates to zero. Each else will be paired to the last encountered if that is not already paired with an else.
4.5.2 Loops

Loops are used for iteration in PRIME. They may be implemented as while or for loops, described in detail here.

While

While loops in PRIME are of the form:

```plaintext
while (expression) statements
```

This functions similarly to an if statement, except for the fact that the statements following a while statement will be repeatedly executed for as long as the expression following the while evaluates to a nonzero value. For this reason, it is essential that the expression be updated at some point over the course of the while loop, to prevent infinite iteration. The evaluation and check of the expression occurs at the beginning of each iteration.

Examples:

```plaintext
int foo;
int bar;
bar = 0;
foo = 0;
while (bar < 5) {
    foo = foo + 1;
    bar = bar + 1;
}
print(foo); /* prints 5 */
print(bar); /* prints 5 */
```

For

For loops in PRIME are of the form:

```plaintext
for ( expression1 ; expression2 ; expression3 ) statement
```

expression1 is executed prior to the first iteration of the loop. expression2 is evaluated prior to every iteration of the loop, and the loop is entered if expression2 evaluates to true. expression1 is executed prior to every iteration of the loop. statements are executed during every iteration. The last expression in the for loop statement is optional (in which case this is like a while loop with the initialization done by expression1.

Examples:

```plaintext
int foo;
int bar;
for(foo = 0; foo < 5; foo = foo+1){
    print(a); /* prints 0 1 2 3 4 */
}
for(bar = 5; bar < 5; bar = bar+1){
    print(bar); /* no printing, does not enter for loop */
}
```

Loop Equivalence

As an example of loop equivalence between while and for loops in PRIME, the following implementations are identical in functionality:

```plaintext
expression1 ;
while (expression2) {
    statement ;
    expression3 ;
}
```
for ( expression1 ; expression2 ; expression3 ) statement

### 4.5.3 Functions

Functions in PRIME are of the form:

```
return-type function-name (parameters) statement
```

Note that all functions in PRIME must return some value (as there is no void type), and the `statement` (i.e. the function body) must therefore contain some return line. This is so that the aggregate types point and lint can be passed in under the hood as a pointer implicitly into the function (allowing for less wasteful memory use), similar to what clang does for some C functions. The return statement is the keyword return followed my a space and the return expression followed by a semicolon. The parameters list can be empty.

```
return expression;
```

### 4.5.4 Return

The return statement is used to return some value from a function to its caller. Each return statement must have an expression of the same type of the function that it is defined for. The user cannot declare a void return type as that is not a keyword in our language. This is so that implicit pointer passing can be done when calling a function that has a point or lint return type. Return types include `ints`, `lints`, `pts`, and `strings` but not curves. Note that behaviour when attempting to return a curve is left undefined and is not advisable. This is because points are usually to be defined on curves that should be known and set explicitly.

### 4.6 Scope

#### 4.6.1 Functions

Each function has its own scope. In our language, formal symbols overwrite globals, and locals overwrite formals in the function’s symbol table. Variables defined in one function are not accessible to another function.

**Examples**:

```plaintext
int add(int a, int b)
{
    return a + b;
}

int main()
{
    int a;
    a = add(39, 3);
    print(a); /* prints 42 from add(39, 3) */
    return 0;
}
```

#### 4.6.2 Curly Brackets

Curly brackets are used to define the scope within which variable identifiers are recognized. Invoking variables and functions must occur within the same scope where the variables/function was defined. Functions, loops, and conditionals all have their own scope, and attempting to access variables defined within this scope externally will lead to an error.

**Examples**:
4.6.3 Semicolons

A semicolon character denotes the end of an expression and the start of the next (since empty space will be ignored). This applies to loops, declarations, conditionals, and all other statement types.

Examples:

```c
int i;
int a;
while (i < 5) {
    i = i + 1;
    a = 0;
}
return a; /* works since 'a' defined prior to while loop */
```

4.7 Built-Ins

4.7.1 Printing

Though we would like to have a Python-style print function that infers our types at compile time, we settled for various print functions for the different types. Note that these print statements add a newline. These are the following, with their corresponding types:

- `print` - prints out integers.
- `printl` - prints out lints.
- `printpt` - prints points.
- `prints` - prints out strings.
- `printc` - prints out curves.

Errors with these types are caught by our semantic checker by comparing to our built-in declared types.

4.7.2 Type Conversions

Since our central types are lints, as curves and points are both built on these, we have ways to convert to lints from ints and strings.

For integers, since the size of these is a strict subset to the possible size of a lint, this is a straightforward conversion given by the function `tolint()`. Please see the following example:

```c
int a;
lint b;
a = 2387468;
printl(tolint(a));
b = tolint(a);
```

For strings, this is a little harder since we need to get the integer representation of the particular sequence of characters. Since this is a cryptography language, we do this with `encode()` and `decode()` built in functions. These methods, done with a C interface, take the string and convert to a large integer (so that they can take somewhat longer length strings) with padding. The decoding step then takes the encoded number, turns it back into a string, adds whatever padding is required, and turns it character.
by character into the input word.
Example usage is as follows:

```ocaml
string s;
lint a;
s = "Hello World";
a = encode(s);
printl(a);
prints(decode(a));
```

### 4.7.3 Random

Because this is a cryptographic language, it would be nice to have a way of obtaining (pseudo)random numbers. For this purpose, we provide a random function that allows provision of a seed and a max number and returns the random number. This is done through C interfacing with GMP external library. Example usage is as follows:

```ocaml
lint max; lint seed;
lint rand;
max = 123451;
seed = 101;
rand = random(seed, max);
printl(rand);
```

## 5 Project Plan

### 5.1 Process

#### 5.1.1 Planning

We used the milestones set out by the course and started working towards them at least two weeks before each deadline. For the final deliverable, we worked every week from the midterm until the final deadline.

Each week when we met we would update the team on what areas of individual progress each had made, what roadblocks had been encountered and what was on the list to do later that week. Main roadblocks would then be discussed as a team, even debugged and then delegated to someone to complete. At the end of each meeting we would set out goals for the week and what times were best to meet the next time. Any major roadblocks led us to OH. We met with Professor Edwards each week after the middle of the term for updates and ideas on how to solve them.

#### 5.1.2 Testing

For testing, we would write a test for successful use of a feature (developed on separate feature branches) and run the entire regression test suite to ensure that the new addition worked AND did not break anything. Fail test cases were then added and semantic checking and scanning changes were done to ensure that these were caught.

#### 5.1.3 Style Guide

We followed two general style guides: one for writing our source code in OCaml/C, and then another for writing our PRIME language test files.

For C, we tried to follow the Linux Kernel C style as shown in [Linux Style](link). In general, this means that:

- functions have braces starting on newlines
- if, while, and for don’t have braces if they just have a one-line statement
- types for a sequence of logical statements are defined above them
- struct definitions at the top of the file or in a header
For OCaml we tried to follow that given by [OCaml Style Tutorial](#). In general, since this was a large code base, the main idea was to make things as readable as possible so that different members of the team could help out, spending as little time as possible to parse what someone else had written. To this end, when writing OCaml:

- Statements were separated or started on different lines if they would become too long to read on one screen.
- Match cases would be aligned with their corresponding match statement unless that contradicted the whitespace issue.
- Different large logical segments such as function definitions for the C libraries would be separated by at least one blank line and commented descriptively to denote their purpose.
- Complex LLVM statements should have a name before passing the builder argument so that they are readable both in OCaml and then with the generated code.
- When similar logic applied over different match cases, keep the bind namings the same.

For our PRIME language, we recommend following a style similar to that proposed by the C style guide. However, since much of cryptography can involve complex mathematical operations, we recommend commenting more around different parts of algorithms.

For C files, we separated core lint functionality, our aggregate types (point and curve) and input parsing into different files as they had different roles and could be tested iteratively at different points.

### 5.1.4 Timeline

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>February 3</td>
<td>Project Proposal</td>
</tr>
<tr>
<td>February 14</td>
<td>First Commit</td>
</tr>
<tr>
<td>March 24</td>
<td>Hello World</td>
</tr>
<tr>
<td>March 30</td>
<td>Lint and Point Types</td>
</tr>
<tr>
<td>April 19</td>
<td>Curve and Modified Point Types</td>
</tr>
<tr>
<td>April 22</td>
<td>Point Operators</td>
</tr>
<tr>
<td>April 25</td>
<td>Demonstrations</td>
</tr>
<tr>
<td>April 26</td>
<td>Final Presentation</td>
</tr>
</tbody>
</table>

### 5.1.5 Software Development Environment

The languages used were: OCaml (including OCamllex for scanner and Ocamlyacc for parser), C for interfacing and Bash scripts for testing. A Makefile was used for compilation, set-up and general orchestration. The interfacing with C was done to take advantage of the [GNU Multiple Precision Library](#) which allowed for our large number implementation. All credits for that library go to their creators seen through the link and Contributors section.

For writing code, we used a combination of Vim, VSCode and PyCharm depending on each individual’s comfort with the system in question. To test the code, we used an altered MicroC bash test script and ran it on a docker environment.

We used Git for version control. We used a GitHub private repository to store our combined code as we built it and integrated it with CircleCI to allow for quicker understanding of build success particularly after merges (see testing section for more information on this).

### 5.1.6 Roles and Responsibilities

Our roles and responsibilities were functionality driven, so all team members contributed to all portions of the code (including testing) while implementing a given function. For example, to implement a new operator, a team member would add the operator to `parser.mly`, `ast.ml`, `sast.ml`, `scanner.ml`, `semant.ml`, `codegen.ml`, and C libraries where applicable. We also collaborated on more complex features (i.e. multiple team members worked on lints, points, curves, and associated operators).
5.2 Project Log

This project log shows a history of commits from February 14th to April 26th. As you can see, each member was heavily involved in the development of the project.

```
// git log --pretty="%C(Yellow)%h %C(reset)%ad (%C(Green)%cr%C(reset))%x09 %C(Cyan)%an: %C(reset)%s" > logs.tex
```

1. c437432 Mon Apr 26 13:35:13 2021 -0400 (84 minutes ago ) Thomas Tran: Merge pull request #49 from thomasundo2/demos2
2. c32306b Mon Apr 26 13:33:11 2021 -0400 (86 minutes ago ) alex-liebeskind: Merge branch 'main' into demos2
3. 7a3846f Mon Apr 26 13:29:54 2021 -0400 (89 minutes ago ) alex-liebeskind: added recursion testing
4. a5616f1 Mon Apr 26 12:24:42 2021 -0400 (109 minutes ago ) nmofficial: small changes and tests to demos
5. cfe9d7d Mon Apr 26 04:44:34 2021 -0400 (10 hours ago ) nmofficial: updated rsa demos
6. a4735f0 Mon Apr 26 04:22:47 2021 -0400 (11 hours ago ) nmofficial: Merge pull request #46 from thomasundo2/demos2
7. fda8caa Mon Apr 26 04:21:46 2021 -0400 (11 hours ago ) nmofficial: added rsa cleaner code version
8. 79ae3f Mon Apr 26 04:19:10 2021 -0400 (11 hours ago ) nmofficial: Merge pull request #47 from thomasundo2/demos2
9. 4a76364 Mon Apr 26 04:16:58 2021 -0400 (11 hours ago ) nmofficial: merging new demos Merge branch 'demos2' of https://github.com/thomasundo2/Prime into demos2
10. 657e8c6 Mon Apr 26 04:14:13 2021 -0400 (11 hours ago ) nmofficial: added string test case
11. 0ebf524 Mon Apr 26 00:46:24 2021 -0400 (14 hours ago ) alex-liebeskind: Diffie-Hellman demo working
12. 457dc69 Sun Apr 25 23:41:21 2021 -0400 (15 hours ago ) pbt-santos: Merge pull request #46 from thomasundo2/string_parsing
13. 93ea4d2 Sun Apr 25 23:35:20 2021 -0400 (15 hours ago ) nmofficial: finished RSA Demo syntax
14. 24ff3b0 Sun Apr 25 23:14:19 2021 -0400 (16 hours ago ) pbt-santos: change encode return and ocaml warnings
15. 5b9b7e0 Sun Apr 25 22:48:26 2021 -0400 (16 hours ago ) nmofficial: added ecc demo, fixed string parsing
16. 32b646c Sun Apr 25 21:35:54 2021 -0400 (17 hours ago ) nmofficial: Merge pull request #45 from thomasundo2/access
17. 89265a Sun Apr 25 21:34:59 2021 -0400 (17 hours ago ) nmofficial: removed test file
18. 7d2a146 Sun Apr 25 21:30:17 2021 -0400 (17 hours ago ) pbt-santos: Fix point ret return and ocaml warnings
19. 8e3d92e Sun Apr 25 20:26:25 2021 -0400 (19 hours ago ) pbt-santos: Add point returns
20. 621bad4 Sun Apr 25 20:00:05 2021 -0400 (19 hours ago ) nmofficial: changing to printc from printpoly Merge branch 'access' of https://github.com/thomasundo2/Prime into access
21. 914fae2 Sun Apr 25 19:59:34 2021 -0400 (19 hours ago ) nmofficial: updated printc
22. a63dfbe Sun Apr 25 19:57:42 2021 -0400 (19 hours ago ) nmofficial: semant for crv access
23. 4bde4fa Sun Apr 25 19:32:19 2021 -0400 (19 hours ago ) alex-liebeskind: update curve printing
24. a1467e8 Sun Apr 25 19:22:02 2021 -0400 (20 hours ago ) nmofficial: small change
25. 46593d9 Sun Apr 25 19:11:29 2021 -0400 (20 hours ago ) pbt-santos: Merge branch 'main' of https://github.com/thomasundo2/Prime into main
26. b1fb591 Sun Apr 25 19:11:21 2021 -0400 (20 hours ago ) pbt-santos: Remove int exp
27. 97c9ff6 Sun Apr 25 18:30:39 2021 -0400 (20 hours ago ) nmofficial: fixed test file
28. 75e6ed3 Sun Apr 25 18:14:46 2021 -0400 (21 hours ago ) alex-liebeskind: Merge pull request #44 from thomasundo2/test
29. f568818 Sun Apr 25 18:09:36 2021 -0400 (21 hours ago ) alex-liebeskind: merge for lint returns
30. 3cb5224 Sun Apr 25 18:07:53 2021 -0400 (21 hours ago ) nmofficial: added real life test cases for big numbers
31. f4felca Sun Apr 25 16:45:43 2021 -0400 (22 hours ago ) pbt-santos: Merge branch 'main' of https://github.com/thomasundo2/Prime into main
9ce06f3 Sat Apr 24 22:31:04 2021 -0400 (2 days ago) pbt-santos: add encoding of string to 3 digit numbers

917aab3 Sat Apr 24 22:22:00 2021 -0400 (2 days ago) nmoofficial: fixed mult semantics issue

ac7a34 Sat Apr 24 19:59:08 2021 -0400 (2 days ago) pbt-santos: Change testfile to report all output

e5cb3bf Sat Apr 24 19:28:44 2021 -0400 (2 days ago) pbt-santos: Fix merge issues

8b9e49f Sat Apr 24 19:12:54 2021 -0400 (2 days ago) nmoofficial: Merge pull request #38 from thomasundo2/poly

6cc8bec Sat Apr 24 19:11:50 2021 -0400 (2 days ago) nmoofficial: Update parser.mly
cda2dd4 Sat Apr 24 19:08:48 2021 -0400 (2 days ago) nmoofficial: Merge branch 'main' into poly

06ad7d1 Sat Apr 24 16:32:33 2021 -0400 (2 days ago) nmoofficial: starting multiplication for points

2315e59 Sat Apr 24 15:55:44 2021 -0400 (2 days ago) alex-liebeskind: hash table for conversion
d4d66a2 Sat Apr 24 03:45:27 2021 -0400 (2 days ago) nmoofficial: fixed .out file for test_lint2

6460397 Sat Apr 24 03:44:04 2021 -0400 (2 days ago) nmoofficial: lint_2 not a fail case

e20c7b1 Sat Apr 24 03:39:25 2021 -0400 (2 days ago) nmoofficial: pulling Merge branch 'main' of https://github.com/thomasundo2/Prime into main
g878c50 Sat Apr 24 03:39:07 2021 -0400 (2 days ago) nmoofficial: deleted print statements in add function, added good test cases, everything passes

3cf1d9bf Sat Apr 24 03:32:43 2021 -0400 (2 days ago) nmoofficial: Merge pull request #37 from thomasundo2/lint_unops

993990a Sat Apr 24 02:26:53 2021 -0400 (3 days ago) nmoofficial: pt add works

255d86b Fri Apr 23 20:51:47 2021 -0400 (3 days ago) nmoofficial: Merge branch 'tests' into main

ab02611 Fri Apr 23 20:51:17 2021 -0400 (3 days ago) alex-liebeskind: added test case / update naming convention

04bdabc Fri Apr 23 18:51:48 2021 -0400 (3 days ago) alex-liebeskind: Merge branch 'main' of https://github.com/thomasundo2/Prime into main

4187535 Fri Apr 23 18:51:35 2021 -0400 (3 days ago) alex-liebeskind: added test_func.
d89c7a5 Fri Apr 23 18:41:14 2021 -0400 (3 days ago) nmoofficial: Merge pull request #37 from thomasundo2/lint_unops

31e5f13 Fri Apr 23 18:39:59 2021 -0400 (3 days ago) nmoofficial: fixed test cases, made rand same seed so test works

057dafe Fri Apr 23 18:23:07 2021 -0400 (3 days ago) nmoofficial: not working for lints
ad101ca Fri Apr 23 17:51:06 2021 -0400 (3 days ago) nmoofficial: addition throws seg fault

4b8a2da Fri Apr 23 12:30:05 2021 -0400 (3 days ago) pbt-santos: fix merging changes
b0dcc0f Fri Apr 23 12:16:59 2021 -0400 (3 days ago) pbt-santos: Merge pull request #36 from thomasundo2/casting

b710371 Fri Apr 23 12:16:47 2021 -0400 (3 days ago) pbt-santos: Merge branch 'main' into casting
c974bb1 Fri Apr 23 12:12:30 2021 -0400 (3 days ago) pbt-santos: Merge pull request #35 from thomasundo2/point

9b9cc72 Fri Apr 23 12:11:53 2021 -0400 (3 days ago) pbt-santos: Merge branch 'main' into point

a0e612 Thu Apr 22 15:19:08 2021 -0400 (4 days ago) nmoofficial: Merge pull request #33 from thomasundo2/lint_sops

335ea5b Thu Apr 22 15:16:41 2021 -0400 (4 days ago) nmoofficial: fixed test cases, made rand same seed so test works

5ec95ac Thu Apr 22 04:18:47 2021 -0400 (4 days ago) nmoofficial: ternary ops implemented successfully, power-mod works
fa36922 Thu Apr 22 03:33:47 2021 -0400 (4 days ago) nmoofficial: debugged parser, semant

5826624 Thu Apr 22 03:28:18 2021 -0400 (5 days ago) nmoofficial: ternops done except codegen
d64e8c5 Thu Apr 22 02:39:47 2021 -0400 (5 days ago) nmoofficial: changed ^ to \ and started ternops

83903b1 Thu Apr 22 02:10:00 2021 -0400 (5 days ago) nmoofficial: add option to seed with time to random
e24b4cd Thu Apr 22 00:48:09 2021 -0400 (5 days ago) nmoofficial: random works

4a9a333 Wed Apr 21 22:12:13 2021 -0400 (5 days ago) nmoofficial: neatened up code for random till not fully functional

131eebb Wed Apr 21 22:06:12 2021 -0400 (5 days ago) nmoofficial: basics for random, not fully implemented

fbb8c43 Wed Apr 21 18:52:23 2021 -0400 (5 days ago) nmoofficial: fixed small bugs with inverse. all cases work although test_all shows intened issue with test_inv_fail
working on points

Thomas Tran: start of points with lint

alex-liebeskind: testing ifelse and loops

Thomas Tran: fix syntax error forwhile to codegen

alex-liebeskind: Merge pull request #29 from thomasundo2/forwhile

alex-liebeskind: comparative operators tested

alex-liebeskind: Merge pull request #28 from thomasundo2/forwhile

alex-liebeskind: comparative operators implemented

alex-liebeskind: Added boolean operators to semant.ml, ast.ml, sast.ml

Thomas Tran: Merge pull request #27 from thomasundo2/point

Thomas Tran: fixed point testing

Thomas Tran: Merge pull request #26 from thomasundo2/maintopoint

Thomas Tran: Merge branch 'point' into maintopoint

pb-t-santos: Merge pull request #25 from thomasundo2/lints

pb-t-santos: Merge branch 'main' into lints

pb-t-santos: Finish lints and improve test_file

pb-t-santos: Add subtract and pow for lints

pb-t-santos: clean up lints code

pb-t-santos: resolve merge conflicts

pb-t-santos: start lint improvement

pb-t-santos: fix: rename testing file in yaml

mnofficial: add works barely from semant.ml

Thomas Tran: add back lint scanner

Thomas Tran: Merge branch 'lints' of https://github.com/thomasundo2/Prime into point

Thomas Tran: Merge branch 'point' into point

alex-liebeskind: removed access from semant.ml

ptb-santos: Add functionality for lints in function calls

ptb-santos: Fix code alignment and new lint print

alex-liebeskind: added for/while tests

alex-liebeskind: added ifelse test cases

ptb-santos: Add parsing for new lints and adding external funcs

alex-liebeskind: Merge pull request #23 from thomasundo2/point

alex-liebeskind: Merge pull request #22 from thomasundo2/point

alex-liebeskind: ifelse forwhile code added - move to respective branches

alex-liebeskind: Merge pull request #21 from thomasundo2/point
curve addition for $b = 1$

Thomas Tran: Merge pull request #19 from thomasundo2/point

Thomas Tran: Merge pull request #20 from thomasundo2/point

point addition and test cases added

thomasundo2: arithmetic-wise

point functional for points

thomasundo2: access fully

points partially works

thomasundo2: accessing strings

add point structs and remove point access

ptb-santos: Merge branch 'lints'
of https://github.com/thomasundo2/Prime into lints

ptb-santos: rename test script

ptb-santos: sub and exp functions

ptb-santos: Merge branch 'lints'
of https://github.com/thomasundo2/Prime into lints

ptb-santos: Add lint exponentiation

ptb-santos: Delete gmpfunc.c

ptb-santos: Update .gitignore

Thomas Tran: Merge pull request #17 from thomasundo2/point

ptb-santos: Add lint addition (memory leaks)

ptb-santos: Add lint initialization

alex-liebeskind: Point (#16)

ptb-santos: Add lint addition (memory leaks)

ptb-santos: Add lint initialization

Professor Edward's advice

ptb-santos: Fix merge and add test output

ptb-santos: Add gmp.c file for lints

ptb-santos: Additional tests

alex-liebeskind: Point type added to codegen.ml

alex-liebeskind: 2D points

ptb-santos: Change Ptlit back to one line

ptb-santos: changes to semant.ml

alex-liebeskind: added elliptic

alex-liebeskind: create and link structs.c
### 6 Architectural Design

#### 6.1 Scanner

Relevant Files: scanner.mll

Implemented by: Alexander Liebeskind, Nikhil Mehta, Pedro B T Santos, Thomas Tran
The scanner takes a program as input and converts into a stream of tokens; these tokens are determined by a set of parsing rules. The tokens include variable names, keywords, types, operators, and literals. Comments and whitespace are ignored by the scanner and not converted to tokens. Relevant files are written in Ocamllex.

6.2 Parser

Relevant Files: parser.mly, ast.ml
Implemented by: Alexander Liebeskind, Nikhil Mehta, Pedro B T Santos, Thomas Tran

The parser generates an abstract syntax tree (AST) from a syntactically valid stream of tokens – the AST is explicitly defined in ast.ml. The hierarchy of parsing from top to bottom is as follows: program, function declarations, statements, and expressions.

Functions to print the AST are also defined in ast.ml; the parsed AST of a program can be printed using Ast.string_of_program.

6.3 Semantic Checking

Relevant Files: semant.ml, sast.ml
Implemented by: Alexander Liebeskind, Nikhil Mehta, Pedro B T Santos, Thomas Tran

Generates a type safe, semantically correct AST, otherwise known as an SAST, by detecting type mismatches, invalid assignments, and incorrect parameters prior to runtime. On success, it maps every element in the AST to its equivalent in the SAST, formally defined in ast.ml.

The SAST also contains functions for printing the syntax tree along with the semantically checked types.

6.4 Code Generation

Relevant Files: codegen.ml
Implemented by: Alexander Liebeskind, Nikhil Mehta, Pedro B T Santos, Thomas Tran

Using the semantically checked SAST, codegen traverses the syntax tree and generates LLVM byte code instructions. To monitor scope, codegen uses StringMap to keep track of local and global variables.

6.5 C Libraries

Relevant Files: gmpfunc.c, structs.c, input.c
Implemented by: Alexander Liebeskind, Nikhil Mehta, Pedro B T Santos, Thomas Tran

C libraries includes external libraries (e.g. GNU Multiple Precision Arithmetic Library), structs, struct declaration, and advanced functions such as point elliptic curve operations linked to the LLVM instructions.

7 Test Plan

7.1 Description

Test case selection:
For all additions, 1-3 test cases were added at a minimum. There was some unit testing of scanner and parser though most tests were done at our Prime language level (an integration test). This testing was intended to make sure all symbols were properly tokenized and to ensure the grammar was unambiguous (i.e. no shift/reduce or reduce/reduce errors). Test cases were also designed to ensure that our semantic checker would properly check types, and argument numbers for functions.

Positive test cases would be to check that functionality we desired was implemented. We used test-driven development, therefore we wrote our test cases and ensured that our compiler passed those cases. If they did not, then we worked on our code until they did. In certain instance, several examples of functioning PRIME code were included in a single test case. This primarily occurred when the feature added was relatively simple or if behavior was similar for several examples. Failing test cases were added to ensure
that undefined behavior was not tolerated. We tailored these test cases to make sure that our compiler had as few bugs as possible.

**Automation:**
The main parts of automation were local and remote.

For local testing, we had the MicroC test_all script slightly modified/simplified. This would run the full regression and integration test suite outputting successes and differences in case of failure. This was key to check if new additions caused prior tests to fail.

For remote testing we used CircleCI, a continuous integration pipeline allowing us to run tests every time that commits were pushed to GitHub. This allowed us several advantages: all can know when the regression test suite is failing and at what particular points (delivered as a notification by email). Every commit shows whether the test suite is failing or successful and the stage of the failure is displayed.

**Roles:**
Pedro set up the CircleCI environment on Github and the entire team worked on test cases depending on what feature branch they were working on. It was everyone’s responsibility to add to the test suite every time that they added a feature.

### 7.2 Test Cases

Expected output is shown above test code.

#### Fail Accessing

```
1 Fatal error: exception Failure("cannot access type: int in i.x")

int main()
{
    /* access an integer id */
    int i;
    i = 1;
    printf(i.x);
}
```

#### Fail Decode

```
1 Fatal error: exception Failure("illegal argument int expected lint in 0")

int main()
{
    decode(0);
    return 0;
}
```

#### Fail Decode2

```
1 Fatal error: exception Failure("illegal assignment lint = string in x = decode(67)")

int main()
{
    lint x;
    x = decode(671);
    return 0;
}
```

#### Fail Encode

```
1 Fatal error: exception Failure("illegal argument lint expected string in 34")

int main()
{
    string x;
    x = "fail";
    /* encode(34, x); UNDEFINED: set but can’t reference */
    encode(341);
}
```
Fail Encode2

Fatal error: exception Failure("expecting 1 arguments in encode")

```c
int main()
{
    lint x;
    printl(encode(x, "3"));
}
```

Fail If else statements

Fatal error: exception Failure("expected integer expression in ")

```c
int main()
{
    int test;
    int test2;
    test2 = 2;
    test = 0;
    if("s") {
        print(1);
    } else{
        print(0);
    }
    return 0;
}
```

Fail Inverse for Ints

Fatal error: exception Failure("illegal binary operator int int in test1 test2")

```c
int main()
{
    int test1;
    int test2;
    test1 = 2;
    test2 = 7;
    print(test1 ' test2);
    return 0;
}
```

Fail Lint Casting

Fatal error: exception Failure("illegal argument lint expected int in 12")

```c
int main()
{
    lint x;
    x = tolint(121);
}
```

Fail Lint Operations

Fatal error: exception Failure("illegal binary operator lint lint in 21973469182365874353456 / 2")

```c
int main()
{
    /* for pow we raise by unsigned int */
    printl(21973469182365874353456L / 2L);
    return 0;
}
```
Fail Modular Power Op

Fatal error: exception Failure("illegal ternary operator lint ^ int @ lint in 3 ^ 5 @ 17")

```c
int main(){
    printl(31 ^ 5 @ 17l);
    return 0;
}
```

Fail Point Access

Fatal error: exception Failure("invalid access element z in a.z")

```c
int main(){
    pt a;
    a = [11, 21] & [(41, 51) : 71];
    printl(a.z);
}
```

Fail Point Type Mismatch

Fatal error: exception Failure("points must have Lint coordinates and be defined under a Poly")

```c
int main(){
    pt x;
    x = ["s", 21] & [(41, 51) : 71];
    return 0;
}
```

Fail Point Type Assign

Fatal error: exception Failure("points must have Lint coordinates and be defined under a Poly")

```c
int main(){
    curve x;
    x = [(1, 21) : 31];
    return 0;
}
```

Fail Poly Given Types

Fatal error: exception Failure("Polynomials must have Lint coefficients and a Lint modulus")

```c
int main(){
    curve crv;
}
```

Fail Point Multiplication

Fatal error: exception Failure("illegal binary operator Point * Point in p1 * p1")

```c
int main(){
    /* creation and assignment */
    pt p1;
    pt p2;
    curve crv;
}
```
crv = [(5l, 12l) : 13l];
p1 = [2l, 2l] & crv;
p2 = [10l, 10l] & crv;
printpt(p1 * p1);
printpt(p1 * 8l);
return 0;
}

Fail Return Statements
int main()
{
  int x;
  x = 15;
  return x;
  print x;
}

 Fail Variable assignment and operations
int main()
{
  int test;
  test = "hi";
  return 1;
}

Fail While Loop
int main()
{
  int i;
  while ("s") { /* should be integer expression */
    i = i + 1;
  }
  return 0;
}

Fail While Loop2
int main()
{
  int i;
  while (i < 2) {
    i = i + 1;
    int a;
    a = 7;
  }
  print(a);  // out of scope */
  return 0;
}
Test Addition

```c
int main()
{
    int test;
    printf(1+2);
    return 0;
}
```

Test Assignment

```c
int main()
{
    int test;
    test = 5;
    printf(test);
    return 0;
}
```

Test Lints on big Curves

```c
int main()
{
    /* creation variables */
    lint p;
    lint a;
    lint b;
    lint x1;
    lint y1;
    lint x2;
    lint y2;
    lint x3;
    lint y3;
    pt q;
    pt r;
    pt s;
    curve crv;
    /* create the curve */
    /* prime number from Microsoft Digital Rights Management*/
    /* As seen in Lecture Slides 13 - MATH UN3025 Prof. Dorial Goldfeld - November, 2020 */
    /* coefficients and point q generated on */
    /* http://www.christelbach.com/ECCalculator.aspx */
    p = 7859631023794288223766947894468973962074985689511;
    a = 31768908125132550347631746138276932727469559271;48571406791775727346184082881005620597346426652);
    b = 485714067917757273461840828810056205973464266521;
    crv = [(a, b) : p];
    /* use subgroup generator q */
    x1 = 7715072162626498261706482685655798899077692541761;
    y1 = 3901575102465566285252794592665149955625331966551;
    q = [x1, y1] & crv;
}
```
printpt(q + q);
return 0;
}

Test Large Lint
100000000000000000
300000000000000000
20000000000000000000000000000000000
100000000000000000

int main()
{
lint l1;
lint l2;
l1 = 1000000000000000001;
l2 = 2000000000000000001;
printl(l1);
printl(l1 + l2);
printl(l1*l2);
printl(l1^ l1@ (l1 *3l));
}

Test Decode
HelloWorld

int main()
{
lint x;
string out;
x = 72101108108111087111141081001;
out = decode(x);
printf(out);
return 0;
}

Test elseif Case

int main()
{
int test;
test = 0;
if(test){
print(1);
}
else if(1){
print(1);
}
return 0;
}

Test Encode
HelloWorld
7210110810811108711114108100
HelloWorld

int main()
{
string in;
lint encoded;
in = "HelloWorld";
printf(in);
encoded = encode(in);
printl(encoded);
prints(decode(encoded));
return 0;

Test For Loop

int main()
{
    int a;
    int b;
    for(a = 0; a < 5; a = a+1){
        print(a);
    }
    for(a = 5; a < 5; a = a+1){
        print(a);
    }
    return 0;
}

Test Function

int add(int a, int b)
{
    return a + b;
}

int main()
{
    int a;
    a = add(39, 3);
    print(a);
    return 0;
}

Hello World Program

int main()
{
    int test;
    print(0);
    return 0;
}

Test Basic If Statement

int main()
{
    int test;
    test = 1;
    if(test) {
        print(1);
    }
    return 0;
}
Test IfElse Statement

```c
int main ()
{
    int test;
    int test2;
    test2 = 2;
    test = 0;
    if(test) {
        print(1);
    } else {
        print(0);
    }
    if(test2){
        print(1);
    } else{
        print(0);
    }
    if(test2 < 0){
        print(1);
    } else{
        print(0);
    }
    if(test2 > 0){
        print(1);
    } else{
        print(0);
    }
    if(0 > test2){
        print(1);
    } else{
        print(0);
    }
    if(test != test2){
        print(1);
    } else{
        print(0);
    }
    if(test == test2){
        print(1);
    } else{
        print(0);
    }
    return 0;
}
```

Test If With Variable Condition

```c
int main (){
    int test;
    test = 5;
    if(test == 5){
        print(1);
    } else {
```
```c
int main ()
{
    lint test1;
    lint test2;
    test2 = 7l;
    test1 = 2l;
    printl (2l ' 7l);
    printl (2l ' test2);
    printl ( test1 ' 7l);
    printl ( test1 ' test2);
    return 0;
}
```

Test Lint Operations

```c
int main ()
{
    print(21 > 11);
    print(11 >= 11);
    print(21 == 21);
    print(21 != 11);
    print(11 < 21);
    print(11 <= 11);
    return 0;
}
```

Test Lint And

```c
int main ()
{
    lint test;
    lint test2;
```
int main()
{
    lint test;
    lint test2;
    test = 1l;
    test2 = 7l;
    if(test && test2) {
        printl(1);
    }
    else{
        printl(0);
    }
    return 0;
}
# Test Lint Unary Op

```c
int main()
{
    printl(!2394653278456834765 l);
    return 0;
}
```

# Test Lint Unary Op 1

```c
int main()
{
    /* Unary operators should not work on lints */
    printl(!2394653278456834765 l);
    return 0;
}
```

# Test Lint Return

```c
lint retLint(lint a)
{
    printl(a);
    return a;
}
```

```c
int main()
{
    printl(retLint(42 l));
    return 0;
}
```

# Test Lint Negative

```c
int main()
{
    lint l1;
    lint l2;
    l1 = 1l;
    l2 = -l1;
    printl(12);
    printl(-11);
    printl(-11);
    return 0;
}
```
```c
int main()
{
    lint l1;
    lint l2;
    l1 = -11l;
    l2 = !l1;
    printf(12);
    printf(!l2);
    printf(!-23l);
    printf(!0 l);
    return 0;
}
```

```c
int main()
{
    lint test;
    lint test2;
    test = 1l;
    test2 = 0l;
    if(test || test2) {
        printf(1);
    }
    else {
        printf(0);
    }
    return 0;
}
```

```c
int main()
{
    lint test;
    lint test2;
    test = 0l;
    test2 = 0l;
    if(test || test2) {
        printf(1);
    }
    else {
        printf(0);
    }
    return 0;
}
```

```c
int main()
{
    int test;
    printf((5+25)*3%4);
    return 0;
}
```
Test Ternary

```c
int main()
{
    lint l1;
    l1 = 3l ^ 5l @ 17l;
    printf1(l1);
    printf1(3l ^ 5l @ 17l);
    printf1(3l ^ 5l + 1l @ 17l);
    printf1(3l ^ 3l ^ 5l + 1l @ 17l @ 17l);
    return 0;
}
```

Test Negative

```c
int main()
{
    int test;
    printf(-1*3);
    return 0;
}
```

Test Not

```c
int main()
{
    int test;
    int test2;
    test2 = 5;
    test = 0;
    printf(!1);
    printf(!0);
    printf(!5);
    printf(!test);
    printf(!test2);
    return 0;
}
```

Test Operations

```c
int main()
{
    printf(1 == 1);
    printf(0 != 1);
    printf(1 >= 0);
    printf(1 <= 2);
    printf(2 > 1);
    printf(0 < 1);
}
Test Point

```plaintext
int main()
{
    /* creation and assignment */
    pt x;
    curve mycurve;
    printpt([[11, 21] & [(1001,2001) : 4001]]);

    mycurve = [(5001,6001) : 7001];
    x = [51, 61] & mycurve;
    printpt(x);
    return 0;
}
```

Test Point Access

```plaintext
int main()
{
    pt a;
    a = [11, 21] & [(41, 51) : 71];
    println(a.x);
    println(a.y);
    /* Will add the poly element soon */
}
```

Test Point Return

```plaintext
pt retPoint(pt a)
{
    printpt(a);
    return a;
}

int main()
{
    curve mycurve;
    pt b;
    mycurve = [(11, 21) : 21];
    b = [01, 11] & mycurve;
    printpt(retPoint(b));
    return 0;
}
```

Test Polynomial

```plaintext
int main()
{
    curve x;
    x = [(11, 21) : 31];
    printc(x);
}
```
```c
printc([(41, 51) : 61]);
return 0;
}

Test Precedence

/* Check integer variable assignment and binops */
int main()
{
    print(1 + 2 * 4 / 2 - 1);
    return 0;
}

Test Print

int main()
{
    int test;
    print("test");
    return 0;
}

Test Print If

int main()
{
    int test;
    print((1==1) + 1);
    if (1==1){
        print(1==1);
    }
    if (!(1!=1)) {
        print(1==2);
    }
    if ((1<2) +1) {
        print(1<2);
    }
    if (1>=2) {
        print(404);
    }
    else {
        print(1);
    }
    return 0;
}

Test Point Add

[0,5] & [(6,12) : 13]
[0,5] & [(6,12) : 13]
[0,5] & [(6,12) : 13]

int main()
{
    /* creation and assignment */
    pt p1;
    pt p2;
    pt p3;
    curve crv;
}
```
int main() {
    /* creation and assignment */
    pt p1;
    pt p2;
    pt p3;
    lint m1;
    curve crv;
    m1 = 11 - 21;
    crv = [(5l, 12l) : 13l];
    p1 = [m1, m1] & crv;
    p2 = [7l, 0l] & crv;
    printpt(p2 + p1);
    p3 = p1 + p2;
    printpt(p3);
    printpt([(5l, 12l) : 13l] + [7l, 0l] & [(5l, 12l) : 13l]);
    return 0;
}

Test Point Add Inf

int main() {
    /* creation and assignment */
    pt p1;
    pt p2;
    pt p3;
    lint m1;
    curve crv;
    m1 = 11 - 21;
    crv = [(5l, 12l) : 13l];
    p1 = [2l, 2l] & crv;
    p2 = [7l, 0l] & crv;
    printpt(p2 + p1);
    p3 = p1 + p2;
    printpt(p3);
    printpt([(2l, 2l) & [(5l, 12l) : 13l] + [7l, 0l] & [(5l, 12l) : 13l]]);
    return 0;
}

Test Point Add Inv

int main() {
    /* creation and assignment */
    pt p1;
    pt p2;
    pt p3;
    curve crv;
    crv = [(5l, 12l) : 13l];
    p1 = [2l, 2l] & crv;
    p2 = [7l, 0l] & crv;
    printpt(p2 + p1);
    p3 = p1 + p2;
    printpt(p3);
    printpt([(2l, 2l) & [(5l, 12l) : 13l] + [7l, 0l] & [(5l, 12l) : 13l]]);
    return 0;
}

Test Point Add Same
int main()
{
    /* creation and assignment */
    pt p1;
    pt p2;
    pt p3;
    curve crv;

    crv = [(5l, 12l) : 13l];
    p1 = [2l, 11l] & crv;
    p2 = [2l, 11l] & crv;
    printpt(p2 + p1);
    p3 = p1 + p2;
    printpt(p3);
    printpt([2l, 11l] & [(5l, 12l) : 13l] + [2l, 11l] & [(5l, 12l) : 13l]);

    return 0;
}
Test Point Neg

```c
int main()
{
    /* creation and assignment */
    pt p1;
    pt p2;
    pt p3;
    curve crv;

    crv = [(5l, 12l) : 13l];
    p1 = [2l, 2l] & crv;
    p2 = -p1;
    printpt(-p1);
    printpt(p2);
    printpt([-2l, 2l] & crv);
    printpt([-1l, -1l] & crv);
    return 0;
}
```

Test Point Neq

```c
int main()
{
    /* creation and assignment */
    pt p1;
    pt p2;
    pt p3;
    curve crv;

    crv = [(5l, 12l) : 13l];
    p1 = [2l, 21l] & crv;
    p2 = -p1;
    printpt(-p1);
    printpt(p2);
    printpt([-2l, 21l] & crv);
    printpt([-1l, -1l] & crv);
    return 0;
}
```

Test Random

```c
int main()
{
    lint l1;
    lint l2;
    lint l3;
    l1 = 1l;
    l2 = 100l;
    l3 = 1000000000000000000000000000000000000000000l;
    println(random(l1,l2));
    println(random(l1,l3));
}
```
```c
printl(random(11,100001));
return 0;
}

Test Recursion

int gcd(int a, int b) {
  if(b != 0) {
    return gcd(b, a % b);
  } else {
    return a;
  }
}

int main() {
  int a;
  int b;
  int c;
  a = 25;
  b = 100;
  c = gcd(a, b);
  print(c);
  return 0;
}

Test Return

int main() {
  int i;
  i = 15;
  print(i);
  return 0;
  /* Notes after a return are ok */
}

Test String

int main() {
  string s;
  s = "hello";
  s = "hi";
  print(s);
  return 0;
}

Test Variable

/* Check integer variable assignment and binops */
int main() {
  int test;
  test = 1;
  test = 1 + 2 * 4 / 2 - 1;
  print(test);
  return 0;
}
```
Test Variable And Or

```c
int main (){
    int i;
    int j;
    i = 5;
    j = 0;
    print(i && j);
    print(i || j);
    print(0 || j);
    print(i && 1);
    print(j && 1);
    return 0;
}
```

Test While

```c
int main (){
    int test;
    int a;
    a = 0;
    test = 0;
    while (a < 5) {
        test = test + 1;
        a = a + 1;
    }
    print(test);
    return 0;
}
```

7.3 Demonstration

Demonstration code is shown above sample generated LLVM code.

Diffie-Hellman Key Exchange on Elliptic Curves

```c
pt alice_cpk(pt q) /* Alice computes public key */
{
    lint alpha;
    alpha = 5371;
    return alpha*q;
}

pt bob_cpk(pt q) /* Bob computes public key */
{
    lint beta;
    beta = 7921;
    return beta*q;
}

int alice_csecret(pt b_public_key) /* Alice computes shared secret */
{
    lint alpha;
    alpha = 5371; /* Alice still has access to alpha */
    prints("Alice's Computed Shared Secret:");
    printpt(alpha*b_public_key);
    return 0;
}
```
```c
int bob_csecret(pt a_public_key) /* Bob computes shared secret */
{
    lint beta;
    beta = 7921; /* Bob still has access to beta */
    printf("Bob's Computed Shared Secret:");
    printpt(beta*a_public_key);
    return 0;
}

int main()
{
    /* Diffie-Hellman Key Exchange on Elliptic Curves */
    /* prime number from Microsoft Digital Rights Management
    * As seen in Lecture Slides 13 - MATH UN3025 Prof. Dorial Goldfeld - November, 2020
    * coefficients and point q generated on
    * http://www.christelbach.com/ECCalculator.aspx
    */

    /* create variables */
    lint p;
    lint a;
    lint b;
    lint x1;
    lint y1;
    pt q;
    pt a_public_key;
    pt b_public_key;

    curve crv;
    /* create the curve */
    p = 7859631023794288223766947894468973962074985689511l;
    a = 317689081251325503476317464138276932727469559271l;
    b = 4857140679177577273461840828810056205973454266521l;
    crv = [(a, b) : p];
    /* use subgroup generator q */
    x1 = 7715072162626498261706682685655798899077692541761l;
    y1 = 390157510246556285252794592665149955625319666551l;
    printf("Elliptic Curve E:");
    printc(crv);
    /* create the point */
    q = [x1, y1] & crv;
    printf("Point q:");
    printpt(q);
    /* Alice computes a Public Key using private alpha */
    a_public_key = alice_cpk(q);
    /* Bob computes a Public Key using private beta */
    b_public_key = bob_cpk(q);
    /* Alice and Bob compute their shared secret using the public transmission from the other */
    alice_csecret(b_public_key);
    bob_csecret(a_public_key);
    return 0;
}
```
4 \%mpz_t = type { i32, i32, i64* }
5 \%point = type { \%mpz_t, \%mpz_t, \%poly }
6 \%poly = type { \%mpz_t, \%mpz_t, \%mpz_t }

8 \@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
9 \@fmt.1 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
10 \@string = private unnamed_addr constant [49 x i8] c"7859631023794288223766947894468897396207498568951\00"
11 \@string.1 = private unnamed_addr constant [48 x i8] c"485714066791775727346180482881005620597365426652\00"
12 \@string.2 = private unnamed_addr constant [49 x i8] c"7150721626649826170648268565579889907769254176\00"
13 \@string.3 = private unnamed_addr constant [49 x i8] c"3901575102465566285257945926681499556253196655\00"
14 \@string.4 = private unnamed_addr constant [49 x i8] c"Elliptic Curve E:\00"
15 \@string.5 = private unnamed_addr constant [9 x i8] c"Point q:\00"
16 \@string.6 = private unnamed_addr constant [49 x i8] c"Bob's Computed Shared Secret:\00"
17 \@string.7 = private unnamed_addr constant [49 x i8] c"Alice's Computed Shared Secret: \00"
18 \@string.8 = private unnamed_addr constant [49 x i8] c"537\00"
19 \@string.9 = private unnamed_addr constant [49 x i8] c"537\00"
20 \@string.10 = private unnamed_addr constant [49 x i8] c"792\00"
21 \@string.11 = private unnamed_addr constant [49 x i8] c"792\00"
22 \@string.12 = private unnamed_addr constant [49 x i8] c"792\00"
23 \@string.13 = private unnamed_addr constant [49 x i8] c"792\00"
24 \@string.14 = private unnamed_addr constant [49 x i8] c"792\00"
25 \@string.15 = private unnamed_addr constant [49 x i8] c"792\00"
26 \@string.16 = private unnamed_addr constant [49 x i8] c"792\00"
27 \@string.17 = private unnamed_addr constant [49 x i8] c"792\00"
28 declare i32 @printf (i8*, ...)
declare i32 @or_func (% mpz_t *, % mpz_t *)
declare i32 @and_func (% mpz_t *, % mpz_t *)
declare i32 @geq_func (% mpz_t *, % mpz_t *)
declare i32 @rand_func (% mpz_t *, % mpz_t *, % mpz_t *)
declare % point * @Point (% point *, % mpz_t *, % mpz_t *, % poly *)
declare % point * @printpt (% point *)
declare % point * @ptadd (% point *, % point *)
declare % point * @ptmul (% mpz_t *, % point *)
declare % point * @ptneg (% point *)
declare i32 @pteq (% point *, % point *)
declare i32 @ptneq (% point *, % point *)
declare i32 @Poly (% poly *, % mpz_t *, % mpz_t *, % mpz_t *)
declare i32 @printc (% poly *)
declare i32 @encode (% mpz_t *, i8 *)
declare i8* @decode (% mpz_t *)
declare i32 @main () {
  entry :
  %p = alloca % mpz_t
  %a = alloca % mpz_t
  %b = alloca % mpz_t
  %x1 = alloca % mpz_t
  %y1 = alloca % mpz_t
  %q = alloca % point
  % a_public_key = alloca % point
  % b_public_key = alloca % point
  % crv = alloca % poly
  %0 = alloca % mpz_t
  % __gmpz_init_set_str = call i32 @__gmpz_init_set_str (% mpz_t * %1 , i8* getelementptr inbounds ([49 x i8], [49 x i8]* @string , i32 0, i32 0), i32 10)
  %2 = getelementptr inbounds % mpz_t , % mpz_t * %0 , i32 0
  %p1 = getelementptr inbounds % mpz_t , % mpz_t * %p, i32 0
  %3 = call i32 @__gmpz_init (% mpz_t * %a3 , % mpz_t * %6)
  %4 = alloca % mpz_t
  %5 = getelementptr inbounds % mpz_t , % mpz_t * %4, i32 0
  % __gmpz_init_set_str2 = call i32 @__gmpz_init_set_str (% mpz_t * %9 , i8* getelementptr inbounds ([48 x i8], [48 x i8]* @string.2 , i32 0, i32 0), i32 10)
  %6 = getelementptr inbounds % mpz_t , % mpz_t * %4, i32 0
  %a3 = getelementptr inbounds % mpz_t , % mpz_t * %a, i32 0
  %7 = call i32 @__gmpz_init_set (% mpz_t * %a3 , % mpz_t * %6)
  %8 = alloca % mpz_t
  %9 = getelementptr inbounds % mpz_t , % mpz_t * %8, i32 0
  % __gmpz_init_set_str4 = call i32 @__gmpz_init_set_str (% mpz_t * %9 , i8* getelementptr inbounds ([48 x i8], [48 x i8]* @string.3 , i32 0, i32 0), i32 10)
  %10 = getelementptr inbounds % mpz_t , % mpz_t * %8, i32 0
  %b5 = getelementptr inbounds % mpz_t , % mpz_t * %b, i32 0
  %11 = call i32 @__gmpz_init_set (% mpz_t * %b5 , % mpz_t * %10)
  %a6 = getelementptr inbounds % mpz_t , % mpz_t * %a, i32 0
  %b7 = getelementptr inbounds % mpz_t , % mpz_t * %b, i32 0
  %p8 = getelementptr inbounds % mpz_t , % mpz_t * %p, i32 0
  %tmp_poly = alloca % poly
  %Poly = call i32 @Poly(% poly*, % mpz_t*, % mpz_t*, % a6, % mpz_t*, %b7, % mpz_t*, %p8)
  %12 = getelementptr inbounds % poly, % poly* %tmp_poly, i32 0
  %13 = load % poly, % poly* %12
  store % poly %13 , % poly* %crv
  %14 = alloca % mpz_t
  %15 = getelementptr inbounds % mpz_t , % mpz_t * %14, i32 0
}
%_gmpz_init_set_str9 = call i32 @__gmpz_init_set_str (% mpz_t * %15 , i8* getelementptr inbounds ([49 x i8], [49 x i8]* @string .4 , i32 0, i32 0), i32 10)
%16 = getelementptr inbounds % mpz_t , % mpz_t * %14 , i32 0
% x110 = getelementptr inbounds % mpz_t , % mpz_t * %x1 , i32 0
%17 = call i32 @__gmpz_init_set (% mpz_t * %x110 , % mpz_t* %16)
%18 = alloca % mpz_t
%19 = getelementptr inbounds % mpz_t , % mpz_t * %18 , i32 0
% __gmpz_init_set_str11 = call i32 @__gmpz_init_set_str (% mpz_t * %19 , i8* getelementptr inbounds ([49 x i8], [49 x i8]* @string .5 , i32 0, i32 0), i32 10)
%20 = getelementptr inbounds % mpz_t , % mpz_t * %18 , i32 0
% y112 = getelementptr inbounds % mpz_t , % mpz_t * %y1 , i32 0
%21 = call i32 @__gmpz_init_set (% mpz_t * %y112 , % mpz_t* %20)
% printf = call i32 (i8*, ...) @printf (i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt .1 , i32 0, i32 0) , i8* getelementptr inbounds ([18 x i8], [18 x i8]* @string .6 , i32 0, i32 0))
% crv13 = getelementptr inbounds %poly , % poly * %crv , i32 0
% printc = call i32 @printc (% poly * % crv13 )
% x114 = getelementptr inbounds % mpz_t , % mpz_t * %x1 , i32 0
% y115 = getelementptr inbounds % mpz_t , % mpz_t * %y1 , i32 0
%22 = call i32 @__gmpz_init_set (% mpz_t * %y115 , % mpz_t * %x114 , % mpz_t * %y115 , % poly* % crv16)
%23 = load % poly , % poly * %22
store % poly %23 , % poly * %q
% printf17 = call i32 (i8*, ...) @printf (i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt .1 , i32 0, i32 0) , i8* getelementptr inbounds ([9 x i8], [9 x i8]* @string .7 , i32 0, i32 0))
% q18 = getelementptr inbounds % point , % point * %q, i32 0
% printpt = call i32 @printpt (% point * % q18 )
% pt_param = load % point , % point * % q19
% a_public_key25 = getelementptr inbounds % point , % point * % a_public_key , i32 0
% pt_param26 = load % point , % point * % a_public_key25
% ret i32 0
}
}

define i32 @bob_csecret (% point % a_public_key ) {
entry :
% a_public_key1 = alloca % point
store % point % a_public_key , % point * % a_public_key1
% beta = alloca % mpz_t
%0 = alloca % mpz_t
%1 = getelementptr inbounds % mpz_t , % mpz_t * %0 , i32 0
% _gmpz_init_set_str = call i32 @__gmpz_init_set_str(%mpz_t* %1 , 18 , i8* getelementptr inbounds ([4 x i8], [4 x i8]* @string .10 , i32 0, i32 0), i32 10)
%2 = getelementptr inbounds % mpz_t , % mpz_t * %0 , i32 0
% beta2 getelementptr inbounds % mpz_t , % mpz_t* %beta , i32 0
%3 = call i32 @__gmpz_init_set (% mpz_t* %beta2 , % mpz_t* %2)
% printf = call i32 (i8*, ...) @printf (i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt .9 , i32 0, i32 0) , i8* getelementptr inbounds ([30 x i8], [30 x i8]* @string .11 , i32 0, i32 0))
% beta3 = getelementptr inbounds % mpz_t , % mpz_t* %beta , i32 0
% public_key23 = getelementptr inbounds % point , % point* % public_key , i32 0
% pt_param24 = load % point , % point * % public_key23
% crv24 = getelementptr inbounds % point , % point* % public_key25
% a_public_key26 = getelementptr inbounds % point , % point* % a_public_key26
% pt_param25 = load % point , % point * % a_public_key26
% bob_csecret_result = call i32 @bob_csecret (% point % pt_param25 )
% ret i32 0
}

#define i32 @bob_csecret (% point % a_public_key ) {
%printpt = call i32 @printpt(%point* %pt_mul)
ret i32 0
}

define i32 @alice_csecret(%point %b_public_key) {
entry:
    %b_public_key1 = alloc %point
    %b = getelementptr inbounds %mpz_t, %mpz_t* %b_public_key1, i32 0
    %__gmpz_init_set_str = call i32 @__gmpz_init_set_str(%mpz_t* %b, %i8* @string.14)
    %beta = getelementptr inbounds %mpz_t, %mpz_t* %b, i32 0
    %pt_mul = call %point* @ptmul(%mpz_t* %beta, %point* %b_public_key1)
    %printpt = call i32 @printpt(%point* %pt_mul)
ret i32 0
}

define %point* @bob_cpk(%point* %sret , %point %q) {
entry:
    %sret1 = alloc %point*
    %q2 = alloc %point
    store %point %sret , %point* %sret1
    store %point %q, %point* %q2
    %alpha = alloc %mpz_t
    %__gmpz_init_set_str = call i32 @__gmpz_init_set_str(%mpz_t* %alpha, %i8* @string.17)
    %pt_mul = call %point* @ptmul(%mpz_t* %alpha, %point* %b_public_key1)
    %printpt = call i32 @printpt(%point* %pt_mul)
ret i32 0
}

define %point* @alice_cpk(%point* %sret , %point %q) {
entry:
    %sret1 = alloc %point*
    %q2 = alloc %point
    store %point %sret , %point* %sret1
    store %point %q, %point* %q2
    %alpha = alloc %mpz_t
    %__gmpz_init_set_str = call i32 @__gmpz_init_set_str(%mpz_t* %alpha, %i8* @string.20)
    %pt_mul = call %point* @ptmul(%mpz_t* %alpha, %point* %b_public_key1)
    %printpt = call i32 @printpt(%point* %pt_mul)
ret i32 0
}

define %point* @bob_cpk(%point* %sret , %point %q) {
entry:
    %sret1 = alloc %point*
    %q2 = alloc %point
    store %point %sret , %point* %sret1
    store %point %q, %point* %q2
    %alpha = alloc %mpz_t
    %__gmpz_init_set_str = call i32 @__gmpz_init_set_str(%mpz_t* %alpha, %i8* @string.23)
    %pt_mul = call %point* @ptmul(%mpz_t* %alpha, %point* %b_public_key1)
    %printpt = call i32 @printpt(%point* %pt_mul)
ret i32 0
}

define %point* @alice_cpk(%point* %sret , %point %q) {
entry:
    %sret1 = alloc %point*
    %q2 = alloc %point
    store %point %sret , %point* %sret1
    store %point %q, %point* %q2
    %alpha = alloc %mpz_t
    %__gmpz_init_set_str = call i32 @__gmpz_init_set_str(%mpz_t* %alpha, %i8* @string.26)
    %pt_mul = call %point* @ptmul(%mpz_t* %alpha, %point* %b_public_key1)
    %printpt = call i32 @printpt(%point* %pt_mul)
ret i32 0
}
int main()
{
    /* creation variables */
    lint p;
    lint a;
    lint b;
    lint x1;
    lint y1;
    lint x2;
    lint y2;
    lint x3;
    lint y3;
    pt q;
    pt r;
    pt s;
    curve crv;
    /* create the curve */
    /* prime number from Microsoft Digital Rights Management */
    /* As seen in Lecture Slides 13 - MATH UN3025 Prof. Dorial Goldfeld - November, 2020 */
    /* coefficients and point q generated on */
    /* http://www.christelbach.com/ECCalculator.aspx */
    p = 785963102379428822376694784468973962074985689511;
    a = 317689081251325503476317464138276932727469559271;
    b = 48571408791775775273461840828810056205973454266521;
    crv = [(a, b) : p];
    /* use subgroup generator q */
    x1 = 771507216262649826170648268565798899077692541761;
    y1 = 390157510246556285252794592665149955625331966551;
    q = [x1, y1] & crv;
    printc(crv);
    print("Elliptic Curve E: ");
    printpt(q);
    r = q + q;
    print("Point r = q + q");
    printpt(r);
    s = i231*q;
    print("123*q");
    printpt(s);
    return 0;
}
0fmt.1 = private unnamed_addr constant [4 x i8] c "%s\0A\00"
0string.2 = private unnamed_addr constant [48 x i8] c "78596310237942882237669478944689739620748568951\00"
0string.3 = private unnamed_addr constant [48 x i8] c "845716067917757727346184082881005620597365426652\00"
0string.4 = private unnamed_addr constant [49 x i8] c "771507216262649826170648268565579889907769254176\00"
0string.5 = private unnamed_addr constant [49 x i8] c "3901575102465566285252794592665149955626253196655100"
0string.6 = private unnamed_addr constant [30 x i8] c "Bob's Computed Shared Secret: \00"
0string.7 = private unnamed_addr constant [32 x i8] c "Alice's Computed Shared Secret: \00"
0fmt.8 = private unnamed_addr constant [4 x i8] c "%d\0A\00"
0fmt.9 = private unnamed_addr constant [4 x i8] c "%s\0A\00"
0fmt.10 = private unnamed_addr constant [4 x i8] c "%s\0A\00"
0fmt.11 = private unnamed_addr constant [4 x i8] c "%s\0A\00"
0fmt.12 = private unnamed_addr constant [4 x i8] c "%s\0A\00"
0fmt.13 = private unnamed_addr constant [4 x i8] c "%s\0A\00"
0fmt.14 = private unnamed_addr constant [4 x i8] c "%s\0A\00"
0fmt.15 = private unnamed_addr constant [4 x i8] c "%s\0A\00"
0fmt.16 = private unnamed_addr constant [4 x i8] c "%s\0A\00"
0fmt.17 = private unnamed_addr constant [4 x i8] c "%s\0A\00"
0fmt.18 = private unnamed_addr constant [4 x i8] c "%s\0A\00"
0fmt.19 = private unnamed_addr constant [4 x i8] c "%s\0A\00"
0fmt.20 = private unnamed_addr constant [4 x i8] c "%s\0A\00"
0string.19 = private unnamed_addr constant [4 x i8] c "%d\0A\00"
0fmt.1 = private unnamed_addr constant [4 x i8] c "%d\0A\00"
0fmt.2 = private unnamed_addr constant [4 x i8] c "%d\0A\00"
0string.3 = private unnamed_addr constant [4 x i8] c "Point q: \00"
0string.4 = private unnamed_addr constant [4 x i8] c "Point q: \00"
0string.5 = private unnamed_addr constant [18 x i8] c "Elliptic Curve E: \00"
0string.6 = private unnamed_addr constant [18 x i8] c "Elliptic Curve E: \00"
0string.7 = private unnamed_addr constant [18 x i8] c "Elliptic Curve E: \00"
0string.8 = private unnamed_addr constant [9 x i8] c "Point q: \00"
0string.9 = private unnamed_addr constant [9 x i8] c "Point q: \00"
0string.10 = private unnamed_addr constant [9 x i8] c "Point q: \00"
0string.11 = private unnamed_addr constant [9 x i8] c "Point q: \00"
0string.12 = private unnamed_addr constant [9 x i8] c "Point q: \00"
0string.13 = private unnamed_addr constant [9 x i8] c "Point q: \00"
0string.14 = private unnamed_addr constant [9 x i8] c "Point q: \00"
0string.15 = private unnamed_addr constant [9 x i8] c "Point q: \00"
0string.16 = private unnamed_addr constant [9 x i8] c "Point q: \00"
0string.17 = private unnamed_addr constant [9 x i8] c "Point q: \00"
0string.18 = private unnamed_addr constant [9 x i8] c "Point q: \00"
0string.19 = private unnamed_addr constant [9 x i8] c "Point q: \00"
0string.20 = private unnamed_addr constant [9 x i8] c "Point q: \00"
0string.21 = private unnamed_addr constant [9 x i8] c "Point q: \00"
%18 = alloca %mpz_t
%19 = getelementptr inbounds %mpz_t, %mpz_t* %18, i32 0
%__gmpz_init_set_str11 = call i32 @__gmpz_init_set_str (%mpz_t* %19, i8* getelementptr inbounds ([49 x i8], [49 x i8]* @string.5, i32 0, i32 0), i32 10)
%20 = getelementptr inbounds %mpz_t, %mpz_t* %18, i32 0
%__gmpz_init_set_str12 = call i32 @__gmpz_init_set_str (%mpz_t* %19, i8* getelementptr inbounds ([49 x i8], [49 x i8]* @string.5, i32 0, i32 0), i32 10)
%y112 = getelementptr inbounds %mpz_t, %mpz_t* %y1, i32 0
%21 = call i32 @__gmpz_init_set (%mpz_t* %y112, %mpz_t* %20)
%printf = call i32 (i8*, ...) @printf (i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt.1, i32 0, i32 0), i8* getelementptr inbounds ([18 x i8], [18 x i8]* @string.6, i32 0, i32 0))
%crv13 = getelementptr inbounds %poly, %poly* %crv, i32 0
%printc = call i32 @printc (%poly* %crv13)
%x114 = getelementptr inbounds %mpz_t, %mpz_t* %x1, i32 0
%y115 = getelementptr inbounds %mpz_t, %mpz_t* %y1, i32 0
%crv16 = getelementptr inbounds %poly, %poly* %crv, i32 0
%tmp_pt = alloca %point
%Point = call i32 @Point (%point* %tmp_pt, %mpz_t* %x114, %mpz_t* %y115, %poly* %crv16)
%22 = getelementptr inbounds %point, %point* %tmp_pt, i32 0
%23 = load %point, %point* %22
store %point %23, %point* %q
%printf17 = call i32 (i8*, ...) @printf (i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt.1, i32 0, i32 0), i8* getelementptr inbounds ([9 x i8], [9 x i8]* @string.7, i32 0, i32 0))
%q18 = getelementptr inbounds %point, %point* %q, i32 0
%printpt = call i32 @printpt (%point* %q18)
%q19 = getelementptr inbounds %point, %point* %q, i32 0
%pt_param = load %point, %point* %q19
%sret_space = alloca %point
%24 = getelementptr inbounds %point, %point* %sret_space, i32 0
%alice_cpk_result = call %point* @alice_cpk (%point* %pt_param)
%25 = getelementptr inbounds %point, %point* %alice_cpk_result, i32 0
%26 = load %point, %point* %25
%27 = getelementptr inbounds %point, %point* %sret_space22, i32 0
%bob_cpk_result = call %point* @bob_cpk (%point* %27, %point* %pt_param21)
%28 = load %point, %point* %28
%29 = load %point, %point* %29
%printpt20 = call i32 @printpt (%point* %pt_param21, %point* %pt_param20)
%2a = getelementptr inbounds %point, %point* %2a, i32 0
%pt_param24 = load %point, %point* %2a
%2b = getelementptr inbounds %point, %point* %2b, i32 0
%pt_param26 = load %point, %point* %2b
%alice_csecret_result = call i32 @alice_csecret (%point* %pt_param26)
%2c = getelementptr inbounds %point, %point* %2c, i32 0
%beta3 = getelementptr inbounds %point, %point* %beta3, i32 0
%beta4 = getelementptr inbounds %point, %point* %beta4, i32 0
%printpt27 = call i32 @printpt (%point* %pt_mull)
%2d = getelementptr inbounds %point, %point* %2d, i32 0
%alice_csecret = call i32 @alice_csecret (%point* %alice_csecret)
%2e = getelementptr inbounds %point, %point* %2e, i32 0
%2f = getelementptr inbounds %point, %point* %2f, i32 0
%alice_csecret = call i32 @alice_csecret (%point* %alice_csecret)
%2g = getelementptr inbounds %point, %point* %2g, i32 0
%beta3 = getelementptr inbounds %point, %point* %beta3, i32 0
%beta4 = getelementptr inbounds %point, %point* %beta4, i32 0
%printpt27 = call i32 @printpt (%point* %pt_mull)
%2h = getelementptr inbounds %point, %point* %2h, i32 0
%beta3 = getelementptr inbounds %point, %point* %beta3, i32 0
%beta4 = getelementptr inbounds %point, %point* %beta4, i32 0
%printpt27 = call i32 @printpt (%point* %pt_mull)
%2i = getelementptr inbounds %point, %point* %2i, i32 0
%beta3 = getelementptr inbounds %point, %point* %beta3, i32 0
%beta4 = getelementptr inbounds %point, %point* %beta4, i32 0
%printpt27 = call i32 @printpt (%point* %pt_mull)
lint encrypt(lint n, lint e){
    string plntxt; lint encotxt; lint ciphtxt;
    plntxt = "Hey Professor Edwards";
    ret i32 0;
}
encotxt = encode(plntxt);
ciphtxt = encotxt ^ e @ n;
return ciphtxt;

int main () { /* Primes taken from RSA Factor Challenge - RSA 250*/
lint p; lint q; lint n; lint e; lint d; lint phi; lint max; lint ciphtxt; lint encotxt;
string mess;
p =
6413528947707158027879019017057738908482501474294344720811685963202453234463023862359875266634770871;
q =
3337202759497815655622601060535511422794076034476755466678452098702384172921003708025744867329688181;

n = p * q;
max = 1000000000000000000000000000000000000000000;
e = random(101, max);
ciphtxt = encrypt(n,e);
phi = (p -1l)*(q -1l);
d = e' phi;
encotxt = ciphtxt ^ d @ n;
mess = decode(encotxt);
prints(mess);
return 0;
}
%encottxt = alloca %mpz_t
%mess = alloca i8*
%0 = alloca %mpz_t
%1 = getelementptr inbounds %mpz_t, %mpz_t* %0, i32 0
%gmzp_init_set_str = call i32 __gmpz_init_set_str(%mpz_t* %1, i8* getelementptr inbounds ([126 x i8], [126 x i8]* @string, i32 0, i32 0), i32 10)
%2 = getelementptr inbounds %mpz_t, %mpz_t* %0, i32 0
%0 = alloca %mpz_t
%1 = getelementptr inbounds %mpz_t, %mpz_t* %0, i32 0
%__gmpz_init_set_str2 = call i32 @__gmpz_init_set_str(%mpz_t* %1, i8* getelementptr inbounds ([126 x i8], [126 x i8]* @string.2, i32 0, i32 0), i32 10)
%2 = getelementptr inbounds %mpz_t, %mpz_t* %0, i32 0
%__gmpz_init_set = getelementptr inbounds %mpz_t, %mpz_t* %0, i32 0
%6 = getelementptr inbounds %mpz_t, %mpz_t* %6, i32 0
%3 = call i32 __gmpz_init_set(%mpz_t* %6, %mpz_t* %2)
%4 = alloca %mpz_t
%5 = getelementptr inbounds %mpz_t, %mpz_t* %4, i32 0
%__gmpz_init_set_str6 = call i32 @__gmpz_init_set_str(%mpz_t* %5, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.3, i32 0, i32 0), i32 10)
%9 = getelementptr inbounds %mpz_t, %mpz_t* %4, i32 0
%12 = alloca %mpz_t
%13 = getelementptr inbounds %mpz_t, %mpz_t* %13, i32 0
%__gmpz_init_set_str8 = call i32 @__gmpz_init_set_str(%mpz_t* %13, i8* getelementptr inbounds ([44 x i8], [44 x i8]* @string.4, i32 0, i32 0), i32 10)
%16 = getelementptr inbounds %mpz_t, %mpz_t* %16, i32 0
%19 = alloca %mpz_t
%20 = getelementptr inbounds %mpz_t, %mpz_t* %20, i32 0
%__gmpz_init_set_str11 = call i32 @__gmpz_init_set_str(%mpz_t* %20, i8* getelementptr inbounds ([3 x i8], [3 x i8]* @string.5, i32 0, i32 0), i32 10)
%21 = getelementptr inbounds %mpz_t, %mpz_t* %21, i32 0
%30 = getelementptr inbounds %mpz_t, %mpz_t* %30, i32 0
%max9 = getelementptr inbounds %mpz_t, %mpz_t* %max9, i32 0
%10 = call i32 __gmpz_init_set(%mpz_t* %10, %mpz_t* %9, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.3, i32 0, i32 0), i32 10)
%15 = alloca %mpz_t
%16 = null
%17 = getelementptr inbounds %mpz_t, %mpz_t* %17, i32 0
%10 = call i32 __gmpz_init_set(%mpz_t* %10, %mpz_t* %9, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.3, i32 0, i32 0), i32 10)
%18 = getelementptr inbounds %mpz_t, %mpz_t* %18, i32 0
%19 = alloca %mpz_t
%12 = getelementptr inbounds %mpz_t, %mpz_t* %12, i32 0
%13 = getelementptr inbounds %mpz_t, %mpz_t* %13, i32 0
%14 = getelementptr inbounds %mpz_t, %mpz_t* %14, i32 0
%15 = getelementptr inbounds %mpz_t, %mpz_t* %15, i32 0
%16 = getelementptr inbounds %mpz_t, %mpz_t* %16, i32 0
%17 = getelementptr inbounds %mpz_t, %mpz_t* %17, i32 0
%18 = getelementptr inbounds %mpz_t, %mpz_t* %18, i32 0
%19 = getelementptr inbounds %mpz_t, %mpz_t* %19, i32 0
%15 = call i32 __gmpz_init_set(%mpz_t* %15, %mpz_t* %14)
%10 = alloca %mpz_t
%16 = null
%17 = getelementptr inbounds %mpz_t, %mpz_t* %17, i32 0
%10 = call i32 __gmpz_init_set(%mpz_t* %10, %mpz_t* %9, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.3, i32 0, i32 0), i32 10)
%15 = call i32 __gmpz_init_set(%mpz_t* %15, %mpz_t* %14)
%16 = alloca %mpz_t
%17 = getelementptr inbounds %mpz_t, %mpz_t* %17, i32 0
%10 = call i32 __gmpz_init_set(%mpz_t* %10, %mpz_t* %9, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.3, i32 0, i32 0), i32 10)
%15 = call i32 __gmpz_init_set(%mpz_t* %15, %mpz_t* %14)
%16 = alloca %mpz_t
%17 = getelementptr inbounds %mpz_t, %mpz_t* %17, i32 0
%10 = call i32 __gmpz_init_set(%mpz_t* %10, %mpz_t* %9, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.3, i32 0, i32 0), i32 10)
%15 = call i32 __gmpz_init_set(%mpz_t* %15, %mpz_t* %14)
%16 = alloca %mpz_t
%17 = getelementptr inbounds %mpz_t, %mpz_t* %17, i32 0
%10 = call i32 __gmpz_init_set(%mpz_t* %10, %mpz_t* %9, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.3, i32 0, i32 0), i32 10)
%15 = call i32 __gmpz_init_set(%mpz_t* %15, %mpz_t* %14)
%16 = alloca %mpz_t
%17 = getelementptr inbounds %mpz_t, %mpz_t* %17, i32 0
...
```c
inbounds ([2 x i8], [2 x i8]* @string.9, i32 0, i32 0), i32 10)

33 = getelementptr inbounds %mpz_t, %mpz_t* %31, i32 0
34 = alloc %mpz_t
35 = getelementptr inbounds %mpz_t, %mpz_t* %34, i32 0
% ____init_set_str23 = call i32 @__gmpz_init_set_str (% mpz_t* %35, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.10, i32 0, i32 0), i32 10)
37 = alloc %mpz_t
38 = getelementptr inbounds %mpz_t, %mpz_t* %37, i32 0
% ____init_set_str25 = call i32 @__gmpz_init_set_str (% mpz_t* %38, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.11, i32 0, i32 0), i32 10)
39 = getelementptr inbounds %mpz_t, %mpz_t* %38, i32 0
% ____init_set_str30 = call i32 @__gmpz_init_set_str (% mpz_t* %39, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.12, i32 0, i32 0), i32 10)
41 = alloc %mpz_t
42 = getelementptr inbounds %mpz_t, %mpz_t* %41, i32 0
% ____init_set_str35 = call i32 @__gmpz_init_set_str (% mpz_t* %42, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.13, i32 0, i32 0), i32 10)
43 = alloc %mpz_t
44 = getelementptr inbounds %mpz_t, %mpz_t* %44, i32 0
% ____init_set_str40 = call i32 @__gmpz_init_set_str (% mpz_t* %45, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.14, i32 0, i32 0), i32 10)
45 = alloc %mpz_t
46 = getelementptr inbounds %mpz_t, %mpz_t* %46, i32 0
% ____init_set_str45 = call i32 @__gmpz_init_set_str (% mpz_t* %47, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.15, i32 0, i32 0), i32 10)
47 = alloc %mpz_t
48 = getelementptr inbounds %mpz_t, %mpz_t* %48, i32 0
% ____init_set_str50 = call i32 @__gmpz_init_set_str (% mpz_t* %49, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.16, i32 0, i32 0), i32 10)
49 = alloc %mpz_t
50 = getelementptr inbounds %mpz_t, %mpz_t* %50, i32 0
% ____init_set_str55 = call i32 @__gmpz_init_set_str (% mpz_t* %51, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.17, i32 0, i32 0), i32 10)
51 = alloc %mpz_t
52 = getelementptr inbounds %mpz_t, %mpz_t* %52, i32 0
% ____init_set_str60 = call i32 @__gmpz_init_set_str (% mpz_t* %53, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.18, i32 0, i32 0), i32 10)
53 = alloc %mpz_t
54 = getelementptr inbounds %mpz_t, %mpz_t* %54, i32 0
```
RSA

```c
lint encrypt(lint n, lint e)
{
    string plntxt;
    lint encotxt;
    lint ciphtxt;

    plntxt = "Hey Professor Edwards";
    encotxt = encode(plntxt);
    prints(""); prints("Encoded text: ");
    printl(encotxt);
    ciphtxt = encotxt ^ e @ n;
    prints(""); prints("Cipher text: ");
    printl(ciphtxt);
    return ciphtxt;
}
```

```c
int main ()
{
    /* RSA Algorithm Demonstration */
    /* Primes taken from RSA Factor Challenge - RSA 250 */

    /* declaration */
    lint p;
    lint q;
    lint n;
    lint e;
    lint d;
    lint phi;
    lint max;
    lint ciphtxt;
    lint decotxt;
    string mess;
    int cp;

    /* Select p and q, compute n = pq, and phi(n) */
    p = 6413528947707158027879019017057738908482501474294344720811685963202453234463023862359875266834770871;
    q = 33372027594978156556226010605355114227940760344767554466678452098702384172921003708025748467329688181;
    n = p * q;
    phi = (p-1l)*(q-1l);

    /* randomly select e */
    max = 1000000000000000000000000000000000000000000000;
    e = random(101, max);
    prints("Public key (n, e): ");
    printl(n);
    prints(" "); printl(e);
    /* n, e are the public key */
    ciphtxt = encrypt(n,e);
    /* find phi(n) and use that to find d */
    d = e'phi;
}```
/* decrypt message */
decotxt = ciphtxt ^ d @ n;
prints("Decrypt encoded text:");
printl ( decotxt );

/* decode message */
prints("Decoded message:");
mess = decode ( decotxt );
prints ( mess );
return 0;

; ModuleID = 'Prime'
source_filename = "Prime"

mpz_t = type { i32, i32, i64 * }
point = type { mpz_t, mpz_t, poly }
poly = type { mpz_t, mpz_t, mpz_t }

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@string = private unnamed_addr constant [126 x i8] c "64135289477071580278790190170577389084825014742943447208116859632024532344630238623598752668347708737661925585694639798853367\00"
@string.1 = private unnamed_addr constant [126 x i8] c "333720275949781565562260106053551142279407603447675546666784520987023841729210037080257486732968110\00"
@string.2 = private unnamed_addr constant [126 x i8] c "000000000000000000000000000000000000000000\00"
@string.3 = private unnamed_addr constant [2 x i8] c"0\0"
@string.4 = private unnamed_addr constant [2 x i8] c"1\0"
@string.5 = private unnamed_addr constant [2 x i8] c"0\0"
@string.6 = private unnamed_addr constant [2 x i8] c"1\0"
@string.7 = private unnamed_addr constant [2 x i8] c"0\0"
@string.8 = private unnamed_addr constant [2 x i8] c"0\0"
@string.9 = private unnamed_addr constant [2 x i8] c"0\0"
@string.10 = private unnamed_addr constant [2 x i8] c"0\0"
@string.11 = private unnamed_addr constant [3 x i8] c"10\0"
@string.12 = private unnamed_addr constant [19 x i8] c"Public key (n, e):\00"
@string.13 = private unnamed_addr constant [1 x i8] zeroinitialize
@string.14 = private unnamed_addr constant [2 x i8] c"0\0"
@string.15 = private unnamed_addr constant [2 x i8] c"0\0"
@string.16 = private unnamed_addr constant [1 x i8] zeroinitialize
@string.17 = private unnamed_addr constant [23 x i8] c"Decrypted encoded text:\00"
@string.18 = private unnamed_addr constant [1 x i8] zeroinitialize
@string.19 = private unnamed_addr constant [17 x i8] c"Decoded message:\00"
@fmt.20 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.21 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@string.22 = private unnamed_addr constant [22 x i8] c"Hey Professor Edwards\00"
@string.23 = private unnamed_addr constant [1 x i8] zeroinitialize
@string.24 = private unnamed_addr constant [14 x i8] c"Encoded text:\00"
@string.25 = private unnamed_addr constant [2 x i8] c"0\0"
@string.26 = private unnamed_addr constant [1 x i8] zeroinitialize
@string.27 = private unnamed_addr constant [13 x i8] c"Cipher text:\00"
declare i32 @printf (i8*, ...)
declare i32 @__gmpz_tdiv_q(%mpz_t*, %mpz_t*, %mpz_t*)
declare i32 @__gmpz_tdiv_r(%mpz_t*, %mpz_t*, %mpz_t*)
declare i32 @__gmpz_pow_ui(%mpz_t*, %mpz_t*, i32)
declare i32 @__gmpz_invert(%mpz_t*, %mpz_t*, %mpz_t*)
declare i32 @__gmpz_powm(%mpz_t*, %mpz_t*, %mpz_t*, %mpz_t*)
declare i32 @lnot_func(%mpz_t*, %mpz_t*)
declare i32 @eq_func(%mpz_t*, %mpz_t*)
declare i32 @neq_func(%mpz_t*, %mpz_t*)
declare i32 @lth_func(%mpz_t*, %mpz_t*)
declare i32 @gth_func(%mpz_t*, %mpz_t*)
declare i32 @leq_func(%mpz_t*, %mpz_t*)
declare i32 @or_func(%mpz_t*, %mpz_t*)
declare i32 @and_func(%mpz_t*, %mpz_t*)
declare i32 @geq_func(%mpz_t*, %mpz_t*)
declare % point * @ptadd(% point *, % point *)
declare % point * @ptmul(% mpz_t *, % point *)
declare % point * @ptneg(% point *)
declare i32 @pteq(% point *, % point *)
declare i32 @ptneq(% point *, % point *)
declare i32 @Poly(% poly *, % mpz_t *, % mpz_t *, % mpz_t *)
declare i32 @Point(% point *, % mpz_t *, % mpz_t *, % poly *)
declare i32 @printpt(% point *)
declare % point * @ptadd(% point*, % point*)
declare % point * @ptmul(% mpz_t*, % point*)
declare % point * @ptneg(% point*)
declare i32 @pteq(% point*, % point*)
declare i32 @ptneq(% point*, % point*)
declare i32 @Poly(% poly*, % mpz_t*, % mpz_t*, % mpz_t*)
declare i32 @printc(% poly*)
declare i32 @encode(% mpz_t*, i8*)
declare i8* @decode(% mpz_t*)
declare i32 @main() {
    entry:
    %p = alloca % mpz_t
    %q = alloca % mpz_t
    %n = alloca % mpz_t
    %e = alloca % mpz_t
    %d = alloca % mpz_t
    %phi = alloca % mpz_t
    %max = alloca % mpz_t
    %ciphertext = alloca % mpz_t
    %decotext = alloca % mpz_t
    %mess = alloca i8*
    %cp = alloca i32
    %p = alloca % mpz_t
    %1 = getelementptr inbounds % mpz_t, % mpz_t* %0, i32 0
    %__gmpz_init_set_str = call i32 @__gmpz_init_set_str(% mpz_t* %1, i8* getelementptr inbounds ([126 x i8], [126 x i8]* @string, i32 0, i32 0), i32 10)
    %2 = getelementptr inbounds % mpz_t, % mpz_t* %0, i32 0
    %phi = getelementptr inbounds % mpz_t, % mpz_t* %p, i32 0
%39 = getelementptr inbounds %mpz_t, %mpz_t* %n, i32 0
%printl1 = call i32 @printf1(%mpz_t* %n)
%printl2 = call i32 @printf1(%mpz_t* %n)
%printl3 = call i32 @printf1(%mpz_t* %n)
%printl4 = call i32 @printf1(%mpz_t* %n)
%printl5 = call i32 @printf1(%mpz_t* %n)
%printl6 = call i32 @printf1(%mpz_t* %n)
%printl7 = call i32 @printf1(%mpz_t* %n)
%printl8 = call i32 @printf1(%mpz_t* %n)
%printl9 = call i32 @printf1(%mpz_t* %n)
%printl10 = call i32 @printf1(%mpz_t* %n)
%printl11 = call i32 @printf1(%mpz_t* %n)
%printl12 = call i32 @printf1(%mpz_t* %n)
%printl13 = call i32 @printf1(%mpz_t* %n)
%printl14 = call i32 @printf1(%mpz_t* %n)
%printl15 = call i32 @printf1(%mpz_t* %n)
%printl16 = call i32 @printf1(%mpz_t* %n)
%printl17 = call i32 @printf1(%mpz_t* %n)
%printl18 = call i32 @printf1(%mpz_t* %n)
%printl19 = call i32 @printf1(%mpz_t* %n)
%printl20 = call i32 @printf1(%mpz_t* %n)
%printl21 = call i32 @printf1(%mpz_t* %n)
%printl22 = call i32 @printf1(%mpz_t* %n)
%printl23 = call i32 @printf1(%mpz_t* %n)
%printl24 = call i32 @printf1(%mpz_t* %n)
%printl25 = call i32 @printf1(%mpz_t* %n)
%printl26 = call i32 @printf1(%mpz_t* %n)
%printl27 = call i32 @printf1(%mpz_t* %n)
%printl28 = call i32 @printf1(%mpz_t* %n)
%printl29 = call i32 @printf1(%mpz_t* %n)
%printl30 = call i32 @printf1(%mpz_t* %n)
%printl31 = call i32 @printf1(%mpz_t* %n)
%printl32 = call i32 @printf1(%mpz_t* %n)
%printl33 = call i32 @printf1(%mpz_t* %n)
%printl34 = call i32 @printf1(%mpz_t* %n)
%printl35 = call i32 @printf1(%mpz_t* %n)
%printl36 = call i32 @printf1(%mpz_t* %n)
%printl37 = call i32 @printf1(%mpz_t* %n)
%printl38 = call i32 @printf1(%mpz_t* %n)
%printl39 = call i32 @printf1(%mpz_t* %n)
%printl40 = call i32 @printf1(%mpz_t* %n)
%printl41 = call i32 @printf1(%mpz_t* %n)
%printl42 = call i32 @printf1(%mpz_t* %n)
%printl43 = call i32 @printf1(%mpz_t* %n)
%printl44 = call i32 @printf1(%mpz_t* %n)
%printl45 = call i32 @printf1(%mpz_t* %n)
%printl46 = call i32 @printf1(%mpz_t* %n)
%printl47 = call i32 @printf1(%mpz_t* %n)
%printl48 = call i32 @printf1(%mpz_t* %n)
%printl49 = call i32 @printf1(%mpz_t* %n)
%printl50 = call i32 @printf1(%mpz_t* %n)
%printl51 = call i32 @printf1(%mpz_t* %n)
%printl52 = call i32 @printf1(%mpz_t* %n)
%printl53 = call i32 @printf1(%mpz_t* %n)
%printl54 = call i32 @printf1(%mpz_t* %n)
%printl55 = call i32 @printf1(%mpz_t* %n)
%printl56 = call i32 @printf1(%mpz_t* %n)
%printl57 = call i32 @printf1(%mpz_t* %n)
%printl58 = call i32 @printf1(%mpz_t* %n)
%printl59 = call i32 @printf1(%mpz_t* %n)
%printl60 = call i32 @printf1(%mpz_t* %n)
%printl61 = call i32 @printf1(%mpz_t* %n)
%printl62 = call i32 @printf1(%mpz_t* %n)
%printl63 = call i32 @printf1(%mpz_t* %n)
%printl64 = call i32 @printf1(%mpz_t* %n)
%printl65 = call i32 @printf1(%mpz_t* %n)
%printl66 = call i32 @printf1(%mpz_t* %n)
%printl67 = call i32 @printf1(%mpz_t* %n)
%printl68 = call i32 @printf1(%mpz_t* %n)
%printl69 = call i32 @printf1(%mpz_t* %n)
%printl70 = call i32 @printf1(%mpz_t* %n)
%printl71 = call i32 @printf1(%mpz_t* %n)
%printl72 = call i32 @printf1(%mpz_t* %n)
%printl73 = call i32 @printf1(%mpz_t* %n)
%printl74 = call i32 @printf1(%mpz_t* %n)
%printl75 = call i32 @printf1(%mpz_t* %n)
%printl76 = call i32 @printf1(%mpz_t* %n)
%printl77 = call i32 @printf1(%mpz_t* %n)
%printl78 = call i32 @printf1(%mpz_t* %n)
%printl79 = call i32 @printf1(%mpz_t* %n)
%printl80 = call i32 @printf1(%mpz_t* %n)
%printl81 = call i32 @printf1(%mpz_t* %n)
%printl82 = call i32 @printf1(%mpz_t* %n)
%printl83 = call i32 @printf1(%mpz_t* %n)
%printl84 = call i32 @printf1(%mpz_t* %n)
%printl85 = call i32 @printf1(%mpz_t* %n)
%printl86 = call i32 @printf1(%mpz_t* %n)
%printl87 = call i32 @printf1(%mpz_t* %n)
%printl88 = call i32 @printf1(%mpz_t* %n)
%printl89 = call i32 @printf1(%mpz_t* %n)
%printl90 = call i32 @printf1(%mpz_t* %n)
%printl91 = call i32 @printf1(%mpz_t* %n)
%printl92 = call i32 @printf1(%mpz_t* %n)
%printl93 = call i32 @printf1(%mpz_t* %n)
%printl94 = call i32 @printf1(%mpz_t* %n)
%printl95 = call i32 @printf1(%mpz_t* %n)
%printl96 = call i32 @printf1(%mpz_t* %n)
%printl97 = call i32 @printf1(%mpz_t* %n)
%printl98 = call i32 @printf1(%mpz_t* %n)
%printl99 = call i32 @printf1(%mpz_t* %n)
%printl100 = call i32 @printf1(%mpz_t* %n)
%ciphertext = alloca %mpz_t
store i8* getelementptr inbounds ([22 x i8], [22 x i8]* @string.22, i32 0, i32 0), i8** %plaintext
%plaintext4 = load i8*, i8** %plaintext
%0 = alloca %mpz_t
%1 = getelementptr inbounds %mpz_t, %mpz_t* %0, i32 0
%enctxt5 = getelementptr inbounds %mpz_t, %mpz_t* %enctxt5, i32 0
%2 = call i32 @__gmpz_init_set(%mpz_t* %enctxt5, %mpz_t* %1)
%printf = call i32 (i8*, ...) @printf (i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt.21, i32 0, i32 0), i8* getelementptr inbounds ([1 x i8], [1 x i8]* @string.23, i32 0, i32 0))
%plaintext6 = call i32 (i8*, ...) @printf (i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt.21, i32 0, i32 0), i8* getelementptr inbounds ([14 x i8], [14 x i8]* @string.24, i32 0, i32 0))
%3 = getelementptr inbounds %mpz_t, %mpz_t* %enctxt5, i32 0
%printf1 = call i32 @printf (i8*, ...) @printf (i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt.21, i32 0, i32 0), i8* getelementptr inbounds ([14 x i8], [14 x i8]* @string.24, i32 0, i32 0))
%4 = alloca %mpz_t
%5 = getelementptr inbounds %mpz_t, %mpz_t* %enctxt5, i32 0
%__gmpz_init_set_str = call i32 @__gmpz_init_set_str (%mpz_t* %5, i8* getelementptr inbounds ([2 x i8], [2 x i8]* @string.25, i32 0, i32 0), i32 10)
%6 = getelementptr inbounds %mpz_t, %mpz_t* %enctxt5, i32 0
%__gmpz_powm = call i32 @__gmpz_powm (%mpz_t* %6, %mpz_t* %enctxt7, %mpz_t* %e8, %mpz_t* %enctxt7, %mpz_t* %enctxt7)
%7 = call i32 @printf (i8*, ...) @printf (i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt.21, i32 0, i32 0), i8* getelementptr inbounds ([1 x i8], [1 x i8]* @string.26, i32 0, i32 0))
%printf12 = call i32 (i8*, ...) @printf (i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt.21, i32 0, i32 0), i8* getelementptr inbounds ([13 x i8], [13 x i8]* @string.27, i32 0, i32 0))
%8 = getelementptr inbounds %mpz_t, %mpz_t* %ciphertext, i32 0
%printf13 = call i32 @printf (i8*, ...) @printf (i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt.21, i32 0, i32 0), i8* getelementptr inbounds ([13 x i8], [13 x i8]* @string.27, i32 0, i32 0))
%9 = getelementptr inbounds %mpz_t, %mpz_t* %ciphertext, i32 0
%__gmpz_init_set = call i32 @__gmpz_init_set (%mpz_t* %10, %mpz_t* %9)
%10 = getelementptr inbounds %mpz_t, %mpz_t* %ciphertext, i32 0
%ret_ptr = load %mpz_t*, %mpz_t** %sret1
%ret_set = call i32 @__gmpz_init_set (%mpz_t* %ret_ptr, %mpz_t* %ciphertext14)
ret %mpz_t* %ret_ptr

8 Lessons Learned

8.1 Team Advice
For future teams, make sure you check in often with your team to get the ball rolling and keep up some momentum. Go to OH as many times as necessary, particularly when getting started so you can be pointed in the right direction, this was an early sticking point for us.

8.2 Alexander Liebeskind
This project was quite demanding, but it’s incredibly fulfilling to see it all come together. I learned a lot from PRIME about everything from functional programming to group collaboration.

One of my most significant takeaways from the project is the importance of designing around the user. Some of the ideas we had at the outset made sense from a mathematical perspective, but weren’t the most effective way to make the life of the programmer easy. For example, we had initially conceived of rings as their own data type, but incorporating rings into curves facilitated elliptic curve cryptography. Realizing an initial proposal is not final and being willing to rethink ideas during implementation is crucial.

Also, communication with the group is vital. Every part of a language relies on the others, so making sure everyone is on the same page is essential.
8.3 Nikhil Mehta

It was a rough semester and I definitely came to realize the importance of getting enough sleep.

Writing robust test cases for every feature in your language as soon as they’re implemented is vital to success. If you implement a feature without testing it in 3 or 5 or 10 different ways it might well be broken. When we didn’t test features as well as we could have, we would later find out that putting in literals would throw weird behavior or that certain expressions would cause a segmentation fault.

In addition, I learned, over time, to appreciate OCaml and functional programming. This required a very different kind of thinking than what has been required in all my prior coursework. Through this project I have come to appreciate functional programming a lot more than I did after the first homework.

Lastly, it’s essential to write down and constantly update both your individual tasks and the group’s tasks. The team was always handling multiple different tasks across different branches at any given moment over the course of the semester and it became very easy for things to slip through the cracks.

8.4 Thomas Tran

LLVM documentation is awful, so understanding how to read it from the start is crucial. Placing an emphasis on understanding the end to end from scanner to executable is especially important, and developing features end to end helped me to understand what was going on low level. Also, using git best practices and setting up continuous integration helps manage version history for large projects like this. (shoutout pedro)

8.5 Pedro B T Santos

This semester’s been difficult. Made more so by the almost intractable nature of LLVM and its sparse documentation.

However, I did learn a lot from experimenting and seeing the project build and take shape was very rewarding. I’d always wanted to learn more functional programming, and was glad to be able to do so with OCaml. I learned the importance of planning and communication when managing a remote work environment. This semester especially when none of us were able to meet one another it was especially important to check in every so often and to remain efficient in what we did.

9 Appendix

Attach a complete code listing of your translator with each module signed by its author
All modules were written and edited by all team members. MicroC was used as a starting point for OCaml files.

9.1 ast.ml

```ocaml
(* Create a new operator for assignment and create a new expression*)
(* sequences of expressions *)

type operator = Add | Sub | Mul | Div | Mod | Pow | Beq | Bneq | Leq | Geq | Lth | Gth | And | Or | Inv

type eqsign = Eq

type uoperator = Neg | Not

type accessor = Access

type toperator = Lpw | Pmd

type typ = Int | Lint | Chr | Ring | String | Point | Poly | Void

type bind = typ * string

type expr =
    Strlit of string
  | Lit of int
  | Litlitr of string
  | Ptlit of expr * expr * expr
  | Polylit of expr * expr * expr
  | Id of string
  | Binop of expr * operator * expr
```
Relop of expr * operator * expr
| Trnop of expr * toperator * expr * toperator * expr
| Unop of uoperator * expr
| Access of string * expr
| Call of string * string (* we will use the second string to convert to gep*)
| Noexpr

type stmt =
  | Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt (*need for pretty print, temp*)
  | For of expr * expr * expr * stmt
  | While of expr * stmt

type func_decl = {
  typ : typ;
  name : string;
  params : bind list;
  locals : bind list;
  body : stmt list;
}

(* Essentially means variable declarations followed by function defs *)
type program = bind list * func_decl list

let string_of_op = function
  Add -> "+"
  | Sub -> "-"
  | Mul -> "*
  | Div -> "/"
  | Mod -> "%
  | Pow -> "\^""\^
  | Inv -> "'
  | Beq -> "=="
  | Bneq -> "!="
  | Leq -> "<="
  | Geq -> ">="
  | Lth -> "<"
  | Gth -> ">
  | And -> "&&"
  | Or -> "||"

let string_of_uop = function
  Neg -> "-
  | Not -> "!

let string_of_top = function
  Lpw -> "^"
  | Pmd -> "@

let rec string_of_expr = function
  Strlit(l) -> "\"" ^ l ^ "\"
  | Id(s) -> s
  | Lit(l) -> string_of_int l
  | Lintlit(l) -> l
  | Ptlit(i, j, p) -> "[" ^ string_of_expr i ^ "," ^ string_of_expr j ^ "] & ^ string_of_expr p
  | Polylit(i,j, m) -> 
  | Binop(e1, o, e2) -> string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Relop(e1, o, e2) -> string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_expr o ^ string_of_expr e
  | Trnop(e1, e2, e3) -> string_of_expr e1 ^ " " ^ string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Access(v, s) -> v ^ "." ^ s
let rec string_of_stmt = function
  | Block(stmts) ->
    "{\n" ^ string_of_stmt stmts ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ "\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ "\n";
  | If(e, s, Block([])) -> "if (\n" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) -> "if (\n" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s1 ^ " else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
    "for (\n" ^ string_of_expr e1 ^ " ; \n" ^ string_of_expr e2 ^ " ; \n" ^ string_of_expr e3 ^ " ) \n" ^ string_of_stmt s
  | While(e, s) -> "while (\n" ^ string_of_expr e ^ " ) \n" ^ string_of_stmt s

let string_of_vdecl (t, id) = string_of_typ t ^ " \n" ^ id ^ " ; \n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " \n" ^ fdecl.name ^ " (\n" ^
  String.concat " (\n" ^ List.map string_of_vdecl fdecl.locals ^ " ) ^ " ) \n"
  ^
  String.concat " \n" ^
  String.concat " \n" ^
  String.concat " \n" ^
  String.concat " \n" ^
  String.concat " \n" ^

let string_of_program (vars, funcs) =
  String.concat " \n" ^
  String.concat " \n" ^
  String.concat " \n" ^
  String.concat " \n" ^
  String.concat " \n" ^
  String.concat " \n" ^

9.2 sast.ml
let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ ( match e with
    | SStrlit (l) -> "" ^ l ^ ""
    | SLintlit (l) -> l
    | SPtlit (i, j, p) -> "[" ^ string_of_sexpr i ^ " , " ^ string_of_sexpr j ^ "] & " ^ string_of_sexpr p
    | SPolylit (i, j, m) -> "[(" ^ string_of_sexpr i ^ " , " ^ string_of_sexpr j ^ ") : " ^ string_of_sexpr m ^ "]" ^ "]
    | SAccess (s, i) -> s ^ ". " ^ string_of_int i
    | SLit (l) -> string_of_int l
    | SId (s) -> s
    | SBinop (e1 , o, e2) -> string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
    | SRelop (e1 , o, e2) -> string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
    | SUnop (o, e) -> string_of_uop o ^ string_of_sexpr e
    | STrnop (e1 , o1 , e2 , o2 , e3) -> string_of_sexpr e1 ^ " " ^ string_of_top o1 ^ " " ^ string_of_sexpr e2 ^ " " ^ string_of_sexpr e3
    | SCall (f, el) -> f ^ "(" ^ String.concat " " (List.map string_of_sexpr el) ^ ")"
    | SAssign (v, e) -> v ^ " = " ^ string_of_sexpr e
    | SNoexpr -> ""
  ) ^ ")"

let rec string_of_sstmt = function
  | SBlock (stmts) -> "{
    String.concat " " (List.map string_of_sstmt stmts) ^ "}\n"
  | SExpr (expr) -> string_of_sexpr expr ^ " ; ";\n"
  | SReturn (expr) -> "return " ^ string_of_sexpr expr ^ " ; ";\n"
  | SIf (e, s, SBlokc ([])) -> "if (" ^ string_of_sexpr e ^ ")\n"
  | SIf (e, s1, e2) -> "if (" ^ string_of_sexpr e ^ ")\n"
  | SIf (e, s1, s2) -> "else\n"
  | SIf (e, s1, s2) -> "if (" ^ string_of_sexpr e ^ " )\n"
  | SFor (e1, e2, e3, s) -> "for (" ^ string_of_sexpr e1 ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^ string_of_sexpr e3 ^ " )\n"
  | SWhile (e, s) -> "while (" ^ string_of_sexpr e ^ " )\n"

let string_of_sfdecl fdecl =
  string_of_typ fdecl.styp ^ " " ^ fdecl.sname ^ "(" ^ (List.map string_of_vdecl fdecl.sparams) ^ ")\n"

let string_of_sprogram (vars, funcs) =
  String.concat " " (List.map string_of_sfdecl vars) ^ "\n"
  String.concat " " (List.map string_of_sfdecl funcs)
%token ACCESS
%token RETURN IF ELSE WHILE FOR INT LINT POLY POINT RING CHAR STRING //(*add float/void here if wanted*)
%token <int> LITERAL
%token <string> STRLIT LINTLIT ID
%token EUF
%start program
%type <Ast . program> program //(* Add in later when we define the AST *)

// (*precedence*)
%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left LTH GTH LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE //(* Change this order later if necessary \r moved mod up -l.guru*)
%right INVERT
%right POWER
%nonassoc AMP
%right PMOD
%left LPOWER
%left ACCESS // Built in access methods
%

//(* All the semantic action braces will be empty for this submission *)
program:
  decls EOF { $1 }

// fst gets the var decs, snd gets the function decs
decls:
  /* nothing */ { ([], []) } // Building up list of variable decs and list of function decs
  | decls declare_init { (($2 :: fst $1), snd $1) } //(* No external variables ? or keep as is*)
  | decls fdecl { ( fst $1 , ( snd $1 :: $2 ) ) }

// create the record denoted by AST
// the body will then contain declarations (allowed by expr)
fdecl:
  typ ID LPAREN params_opt RPAREN LBRACE seq_stmts RBRACE
  { typ = $1;
    name = $2;
    params = List . rev $4;
    locals = List . rev (fst $7);
    body = List . rev (snd $7) (* Might have to split this for hello world *)
  }

params_opt:
  /* nothing */ { [] }
  | params_list { $1 }

// Have the lists on the left
params_list:
  typ ID
  { ($1, $2) }
  | params_list COMMA typ ID { ($3, $4 :: $1) }

typ:
  INT { Int }
  | LINT { Lint }
  | POINT { Point }
  | POLY { Poly }
  | STRING { String }

vars:
79 /* nothing */ { [] } 
80 | vars declare_init { $2 :: $1 } 
81 declare_init: 
82 typ declarator SEMI { ($1, $2) } 
83 declarator: 
84   ID { $1 } 
85 seq_stmts: 
86   vars stmt_list { ($1, $2) } 
87 stmt_list: 
88 /* nothing */ { [] } 
89 | stmt_list stmt { $2 :: $1 } 
90 stmt: 
91 expr_opt SEMI { Expr $1 } // (* Expr - stmt *) 
92 | RETURN expr_opt SEMI { Return $2 } // (* Return stmt *) 
93 | LBRACE stmt_list RBRACE { Block ( List . rev $2) } 
94 | IF LPAREN expr RPAREN stmt % prec NOELSE { If($3, $5, Block([[]])) } 
95 | IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) } 
96 | FOR LPAREN expr SEMI expr SEMI expr_opt RPAREN stmt { For($3, $5, $7, $9) } 
97 | WHILE LPAREN expr RPAREN stmt { While($3, $5) } 
98 expr_opt: 
99 /* nothing */ { Noexpr } 
100 | expr { $1 } 
101 expr: 
102 ID { Id($1) } 
103 | LITERAL { Lit($1) } 
104 | STRLIT { Strlit($1) } 
105 | LINTLIT { Lintlit($1) } 
106 | LBRACK expr COMMA expr RBRACK AMP expr { Ptlit($2, $4, $7) } 
107 | LBRACK LPAREN expr COMMA expr RPAREN COLON expr RBRACK { Polylit($3, $5, $8)} 
108 | expr MOD expr { Binop($1, Mod, $3) } 
109 | expr POWER expr { Binop($1, Pow, $3) } 
110 | expr PLUS expr { Binop($1, Add, $3) } 
111 | expr MINUS expr { Binop($1, Sub, $3) } 
112 | expr TIMES expr { Binop($1, Mul, $3) } 
113 | expr DIVIDE expr { Binop($1, Div, $3) } 
114 | expr INVERT expr { Binop($1, Inv, $3) } 
115 | expr BEQ expr { Relop($1, Beq, $3) } 
116 | expr BNEQ expr { Relop($1, Bneq, $3)} 
117 | expr LTH expr { Relop($1, Lth, $3) } 
118 | expr LEQ expr { Relop($1, Leq, $3) } 
119 | expr GTH expr { Relop($1, Gth, $3) } 
120 | expr GEQ expr { Relop($1, Geq, $3) } 
121 | expr AND expr { Relop($1, And, $3) } 
122 | expr OR expr { Relop($1, Or, $3) } 
123 | expr LPOWER expr PMOD expr { Trnop($1, Lpw, $3, Pmd, $5) } 
124 | MINUS expr %prec NOT { Unop(Not, $2) } 
125 | NOT expr { Unop(Not, $2) } 
126 | ID ASSIGN expr { Assign($1, $3) } 
127 | ID LPAREN args_opt RPAREN { Call($1, $3) } 
128 | LPAREN expr RPAREN { $2 } 
129 args_opt: 
130 /* nothing */ { [] } 
131 | args_list { List . rev $1 } 
132 args_list: 
133 expr { [$1] } 
134 | args_list COMMA expr { $3 :: $1 } 

9.4 scanner.mll

/* Ocamllex scanner for PRIME
Many thanks to the MicroC compiler example created by
Professor Edwards

Many of the symbols here are directly from or follow that.

let digit = ['0' - '9']
let digits = digit+

rule token = parse
  {
    [ ' ' '	' '' '
' ] { token lexbuf } /* add comments */
    | '{'   { LBRACE } /* Grouping operators */
    | ']'   { RBRACE }
    | '{'   { LBRACK } /* More needs to be done here */
    | '}'   { RBRACK }
    | ','   { COMMA }
    | '='   { ASSIGN } /* Binary Operators (semi perhaps not) */
    | ';'   { SEMI }
    | ':'   { COLON }
    | '+'   { PLUS }
    | '-'   { MINUS }
    | '*'   { TIMES }
    | '/'   { DIVIDE }
    | '^\^' { POWER }
    | '%'   { MOD }
    | '''   { INVERT }
    | '.'   { ACCESS }
    | '&'   { AMP }
    | '@'   { PMOD }
    | '^'   { LPOWER }
    | '==' { BEQ } /* Relational Ops (which ones of these do we want?) */
    | '!'   { BNEQ }
    | '<'   { LTH }
    | '>=' { LEQ }
    | '>'   { GTH }
    | '&&' { AND }
    | '||' { OR }
    | '!'  { NOT }
    | 'if' { IF } /* Keywords and types */
    | "else" { ELSE }
    | "for"  { FOR }
    | "while" { WHILE }
    | "return" { RETURN }
    | "int" { INT }
    | "char" { CHAR }
    | "string" { STRING }
    | "lint" { LINT }
    | "poly" { POLY }
    | "pt" { POINT }
    | "ring" { RING }
    | ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9' '_']* as name { ID(name) } /* ids can be
      alpha followed by alphanum and _ */
    | digits as lit { LITERAL(int_of_string lit) }
    | digits as lit '"' { LISTRULIT(lit) }
    | digits as lit '"' { STRLIT(lit) } Make a separate rule for looking through string
      literals and comment literals *
    | eof  { EOF } /* any
      other character is not allowed */
  }

(* part of rule for ending comments *)
and comment = parse
  {
    [ '/' '
' ] { token lexbuf } /* back to normal scanning */
    | _ { comment lexbuf } /* keep reading comments */
  }
9.5 semant.ml

(* Semantic checking file *)
open Ast
open Sast

(* Make a map to keep track of globals *)
module StringMap = Map.Make(String)

(* String hashmap for lint string conversion *)
(* e.g. RSA *)
module StringHash = Hashtbl.Make(struct
type t = string
let equal x y = x = y
let hash = Hashtbl.hash
end);

let vals : int StringHash.t = StringHash.create 10;;

(* Begin Semantic checking sast if good else error *)
let check (globals, functions) =
(* Check binds have types and ids are unique *)
let check_binds (kind : string) (binds : bind list) =
List.iter (function
(Void , b) -> raise (Failure ("missing/wrong type in declaration " ^ kind ^ " " ^ " ^ b))
| _ -> ()) binds;
let rec dups = function
[ ] -> () (* No name found here *)
| ((_, n1) :: (_, n2) :: _) when n1 = n2 -> (* check if same in order because
sorted *)
  raise (Failure ("duplicate " ^ " " ^ " n1))
| _ :: t -> dups t (* check the tail of the binds *)
in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)

(* Now actually perform the checks first variables then functions *)
check_binds "global" globals;

(* Start with function declarations for built-ins (just print for now)*)
(* Just call the formal parameter ID of our inbuilt functions x*)
let built_in_decls =
  let add_bind map (name , ty) = StringMap.add name {
    typ = Void; (* Our built in print functions will return string*)
    name = name;
    params = [(ty , "x")];
    locals = []; body = [] (* In-built don't have body. Determine semantics here *)
  } map
  in let void_decls = List.fold_left add_bind StringMap.empty [ ("print", Int);
    ("prints", String);
    ("print1", Lint);
    ("printpt", Point);
    ("printc", Poly);]

  and add_cast map (name , ty) = StringMap.add name {
    typ = Lint;
    name = name;
    params = [(ty , "x")];
    locals = []; body = []
  } map
  in let void_decls = List.fold_left add_cast void_decls [ ("tolint", Int) ]

and add_rand map (name , ty) = StringMap.add name {
  typ = Lint;
  name = name;
  params = [(ty , ("x")); (ty , ("y"))];
  locals = []; body = []
} map
in let void_decls = List.fold_left add_rand void_decls ["random", Lint] 

and add_decode map (name, ty) = StringMap.add name {
  typ = String;
  name = name;
  params = [(ty, "x")];
  locals = []; body = []
} map

and add_encode map (name, _) = StringMap.add name {
  typ = Lint;
  name = name;
  params = [[(Lint, "x"); (String, "y")];
  locals = []; body = []
} map

in let built_decls = List.fold_left add_decode void_decls ["decode", Lint]
in List.fold_left add_encode built_decls ["encode", (String)]

(* We likely don't need the GMP functions here because they are not called directly (in fact should not be) *)

in

(* Now keep track of these named built-in funcs in the top-level symbol table *)

let add_func map fd =

(* Define what errors we might have *)

let built_in_err = " function " ^ fd.name ^ " not defined "

and dup_err = " duplicate function found : " ^ fd.name

and make_err er = raise (Failure er) (* Helper to throw error with msg = er *)

in

(* Make the symbol table starting with the built-in functions *)

let function_decls = List.fold_left add_func built_in_decls functions

(* Returning the added function *)

let find_func s =
  try StringMap.find s function_decls
  with Not_found -> raise (Failure("function not found: " ^ s))

(* check function bodies *)

let check_function func =

(* All #TODO: *)

(* check type and identifiers in formal parameters and local vars *)

(* check all assignments are valid types. Should we coerce? *)

let check_assign lvaltype rvaltype err =

(* print_string (" param: " ^ (string_of_typ lvaltype) ^ " actual: " ^ (string_of_typ rvaltype) ^ "\n"); *)

match rvaltype with

(* Lint -> if rvaltype = String || rvaltype = Lint then lvaltype else raise (Failure err) *)

| _ -> if lvaltype = rvaltype then lvaltype else raise (Failure err)

(* if lvaltype is lint and rvaltype is string then lvaltype else raise failure*)

in

(* make local symbol table and functions to use it*)

(* Build local symbol table of variables for this function *)

let symbols = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
  StringMap.empty (globals @ func.params @ func.locals)

(* Return a variable from our local symbol table *)

let type_of_identifier s =

try StringMap.find s symbols
with Not_found -> raise (Failure("undeclared identifier " ^ s))
let rec expr = function
  Lit l -> (Int , SLit l)
| Id s -> ( type_of_identifier s, SId s)
| Strlit l -> ( String , SStrlit l) (* String literals *)
| Lintlit l -> (Lint , SLintlit l)
| Noexpr -> (Void , SNoexpr)
| Assign(var , e) as ex ->
  let lt = type_of_identifier var
  and (rt , e') = expr e in
  let err = " illegal assignment " ^ string_of_typ lt ^ " = " ^
  string_of_typ rt ^ " in " ^ string_of_expr ex
  in (check_assign lt rt err , SAssign (var , (rt , e')))
| Ptlit (e1 , e2 , e3) ->
  let (t1 , e1') = expr e1
  and (t2 , e2') = expr e2
  and (t3 , e3') = expr e3 in
  let ty = match t1 , t2 , t3 with
    Lint , Lint , Poly -> Point
  | _ -> raise ( Failure ("points must have Lint coordinates and be defined under a Poly"))
  in (ty , SPtlit ((t1 , e1') , (t2 , e2') , (t3 , e3')))
| Access (var , e2) as ex -> (* Will give us the right index for gep from string *)
  let lt = type_of_identifier var in
  ( match lt with
    Point -> ( match e2 with
      "x" -> (Lint , SAccess (var , 0))
    | "y" -> (Lint , SAccess (var , 1))
    | _ -> raise ( Failure ("invalid access element " ^ e2 ^ " in " ^
                         string_of_expr ex)))
  | _ -> raise ( Failure ("cannot access type : " ^ string_of_typ lt ^
                         " in " ^ string_of_expr ex)))
| Polylit (e1 , e2 , e3) ->
  let (t1 , e1') = expr e1
  and (t2 , e2') = expr e2
  and (t3 , e3') = expr e3 in
  let ty = match t1 , t2 , t3 with
    Lint , Lint , Lint -> Poly
  | _ -> raise ( Failure ("Polynomials must have Lint coefficients and a Lint modulus")
  )
  in (ty , SPolylit ((t1 , e1') , (t2 , e2') , (t3 , e3')))
| Unop (op , e) as ex ->
  let (t, e') = expr e in
  let ty = match op with
    Neg when t = Int -> Int
  | Not when t = Int -> Int
  | Neg when t = Lint -> Lint
  | Not when t = Lint -> Lint
  | Neg when t = Point -> Point
  | _ -> raise ( Failure ("illegal unary operator " ^
                           string_of_uop op ^ string_of_typ t ^
                           " in " ^ string_of_expr ex))
  in (ty , SUNop (op , (t, e')))
| Binop (e1 , op , e2) as e ->
  let (t1 , e1') = expr e1
  and (t2 , e2') = expr e2 in
  (* All binary operators require operands of the same type *)
  let same = t1 = t2 in
  (* Determine expression type based on operator and operand types *)
  let ty = match op with
    Add | Sub | Mul | Div | Mod | Pow when same && t1 = Int -> Int
  | Add | Sub | Mul | Div | Mod | Inv when same && t1 = Lint -> Lint
  | Add when same && t1 = Point -> Point
  | _ -> raise ( Failure ("illegal binary operator " ^
                           string_of_typ t1 ^ " " ^
                           string_of_op op ^ " " ^
                           string_of_typ t2 ^ " in " ^ string_of_expr e))
  in (ty , SBinop (op , (t1 , e1') , (t2 , e2')))

(* semantic expression checking *)
206 in (ty, SRelop ((t1, e1'), op, (t2, e2')))
207 | Rlo(e1, op, e2) as e ->
208 let (t1, e1') = expr e1
209 and (t2, e2') = expr e2 in
210 let same = t1 = t2 in
211 let ty = match op with
212 | Beq | Bneq | Leq | Geq | Lth | Gth | And | Or when same && t1 = Int -> Int
213 | Beq | Bneq when same && t1 = Point ->
214 Int
215 | _ -> raise (Failure "illegal relational operator " ^
216 string_of_typ t1 ^ " string_of_op op ^ " ^
217 string_of_typ t2 ^ " in " ^
218 string_of_expr e))
219 in (ty, SRelop ((t1, e1'), op, (t2, e2')))
220 | Tlno(e1, o1, e2, o2, e3) as e ->
221 let (t1, e1') = expr e1
222 and (t2, e2') = expr e2
223 and (t3, e3') = expr e3 in
224 let ty = match o1, o2 with
225 Lpw, Pmd when t1 = Lint && t2 = Lint && t3 = Lint -> Lint
226 | _ -> raise (Failure "illegal ternary operator " ^
227 string_of_typ t1 ^ " string_of_top o1 ^ " ^
228 string_of_typ t2 ^ " string_of_top o2 ^ " ^
229 string_of_typ t3 ^ " in " ^
230 string_of_expr e))
231 in (ty, STlno ((t1, e1'), o1, (t2, e2'), o2, (t3, e3')))
232 | Call(name, args) (* as call *) ->
233 let fd = find_func name in
234 let param_length = List.length fd.params in
235 if List.length args != param_length then
236 raise (Failure "expecting " ^
237 string_of_int param_length ^ " arguments in " ^
238 name))
239 else let check_call (param_typ, _) e = (validate call *)
240 let (et, e') = expr e in (recursively semantic check expr *)
241 let err = "illegal argument " ^
242 string_of_typ param_typ ^ " in " ^
243 string_of_expr e
244 in (check_assign param_typ et err, e')
245 in
246 let args' = List.map2 check_call fd.params args
247 in (fd.typ, SCall (name, args'))
248
249 let check_int_expr e =
250 let (t', e') = expr e
251 and err = "expected integer expression in " ^
252 string_of_expr e
253 in if t' != Int then raise (Failure err) else (t', e')
254 in
255
256(* Here is where we check statements (only expr and Block for now) *)
257 let rec check_stmt = function
258 Expr e -> SExpr (expr e) (* recursive check *)
259 | Return e -> let (t, e') = expr e in
260 if t = func.typ then SReturn (t, e') (* Correct return type for function *)
261 else raise (Failure "wrong return type")
262 | Block sl ->
263 let rec check_stmt_list = function (* Maybe add other return checks here *)
264 [Return _ as s] -> [check_stmt s]
265 | _ -> raise (Failure "nothing may follow a return")
266 | Block sl :: ss -> check_stmt_list (sl @ ss) (* Flatten blocks *)
267 s :: ss -> check_stmt_list ss (* one statement at a time *)
268 | [] -> [] (* done *)
269 in SBlock(check_stmt_list sl)
270 | If(p, b1, b2) -> SIf(check_int_expr p, check_stmt b1, check_stmt b2)
271 | For(e1, e2, e3, st) ->
272 SFor(expr e1, check_int_expr e2, expr e3, check_stmt st)
273 | While(p, s) -> SWhile (check_int_expr p, check_stmt s)
274 in
275 { styp = func.typ;
276 sname = func.name;
277 sparams = func.params;
278 slocals = func.locals;
279 sbody = check_stmt (Block func.body) with
SBloc(k sl) -> sl
| _ -> raise (Failure ("blocking failed"))
in (globals, List.map check_function functions)

9.6 codegen.ml

(* This file will be used to get LLVM to work for our compiler as an IR *)
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR
LLVM tutorial: Make sure to read the OCaml version of the tutorial
http://llvm.org/docs/tutorial/index.html
Detailed documentation on the OCaml LLVM library:
http://llvm.moe/
http://llvm.moe/ocaml/ *)

module L = Llvm
module A = Ast
open Sast

module StringMap = Map.Make (String)

(* translate : Sast.program -> Llvm.module *)
let translate (globals, functions) =
let context = L.global_context () in

(* Create the LLVM compilation module into which
we will generate code *)
let the_module = L.create_module context "Prime" in

(* Get types from the context *)
let i32_t = L.i32_type context
and i8_t = L.i8_type context
and void_t = L.void_type context
and string_t = L.pointer_type (i8_t)
and mpz_t = L.named_struct_type context "mpz_t"
in let mpz_t = L.named_struct_type context "mpz_t"
in let poly_t = L.named_struct_type context "poly"
in let point_t = L.named_struct_type context "point"
in

(* Return the LLVM type for a MicroC type *)
let ltype_of_typ = function
| A.String -> string_t
| A.List -> mpz_t
| A.Point -> point_t
| A.Poly -> poly_t
| A.Int -> i32_t
| A.Void -> void_t
| _ -> void_t
in

(* Create a map of global variables after creating each *)
let global_vars : L.llvalue StringMap.t =
let global_var m (t, n) =
let init = match t with
| L.const_int (ltype_of_typ t) 0
in StringMap.add n (L.define_global n init the_module) m in
List.fold_left global_var StringMap.empty globals in

let printf_t : L.lltype =
L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in

let printf_func : L.llvalue =
L.declare_function "printf" printf_t the_module in

(* Declare our external functions here *)

(* LINTS *)

let linit_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t; string_t; i32_t |] in
let linit_func : L. llvalue =
  L. declare_function "__gmpz_init_set_str" linit_t the_module in

let lcast_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t; i32_t |] in
let lcast_func : L. llvalue =
  L. declare_function "__gmpz_init_set_si" lcast_t the_module in

let linitdup_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t; L. pointer_type mpz_t |] in
let linitdup_func : L. llvalue =
  L. declare_function "__gmpz_init_set" linitdup_t the_module in

(let The following would be needed in future work to free mpz memory *)

(* let lclear_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t |] in
let lclear_func : L. llvalue = (* free lints - define usage *)
  L. declare_function "__gmpz_clear" lclear_t the_module in *)

(* We don't use the mpz_out_str because FILE* is a pain *)

let lprint_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t |] in
let lprint_func : L. llvalue =
  L. declare_function "printl" lprint_t the_module in

let ladd_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t; L. pointer_type mpz_t; L. pointer_type mpz_t |] in
let ladd_func : L. llvalue =
  L. declare_function "__gmpz_add" ladd_t the_module in

let lsub_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t; L. pointer_type mpz_t; L. pointer_type mpz_t |] in
let lsub_func : L. llvalue =
  L. declare_function "__gmpz_sub" lsub_t the_module in

let lmul_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t; L. pointer_type mpz_t; L. pointer_type mpz_t |] in
let lmul_func : L. llvalue =
  L. declare_function "__gmpz_mul" lmul_t the_module in

let ldiv_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t; L. pointer_type mpz_t; L. pointer_type mpz_t |] in
let ldiv_func : L. llvalue =
  L. declare_function "__gmpz_tdiv_q" ldiv_t the_module in

let lmod_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t; L. pointer_type mpz_t; L. pointer_type mpz_t |] in
let lmod_func : L. llvalue =
  L. declare_function "__gmpz_tdiv_r" lmod_t the_module in

(let This power function will be used to raise to an unsigned int power *)

let lpow_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t; L. pointer_type mpz_t; i32_t |] in
let lpow_func : L. llvalue =
  L. declare_function "__gmpz_pow_ui" lpow_t the_module in

let linv_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t; L. pointer_type mpz_t; L. pointer_type mpz_t |] in
let linv_func : L. llvalue =
  L. declare_function "__gmpz_invert" linv_t the_module in

let lpowmod_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t; L. pointer_type mpz_t; L. pointer_type mpz_t |] in
let lpowmod_func : L. llvalue =
  L. declare_function "__gmpz_powm" lpowmod_t the_module in

let lneg_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t; L. pointer_type mpz_t; |] in
let lneg_func : L. llvalue =
  L. declare_function "__gmpz_neg" lneg_t the_module in

let lnot_t : L. lltype =
  L. function_type i32_t [| L. pointer_type mpz_t; L. pointer_type mpz_t; |] in
let lnot_func : L. llvalue =
  L. declare_function "__gmpz_neg" lnot_t the_module in

81
let lnot_func : L.llvalue =
    L.declare_function "lnot_func" lnot_t the_module in

(* comparator operators *)
let l_eq_t : L.lltype =
    L.function_type i32_t [| L.pointer_type mpz_t; L.pointer_type mpz_t |] in
let l_eq_func : L.llvalue =
    L.declare_function "eq_func" l_eq_t the_module in
let l_neq_t : L.lltype =
    L.function_type i32_t [| L.pointer_type mpz_t; L.pointer_type mpz_t |] in
let l_neq_func : L.llvalue =
    L.declare_function "neq_func" l_neq_t the_module in
let l_lth_t : L.lltype =
    L.function_type i32_t [| L.pointer_type mpz_t; L.pointer_type mpz_t |] in
let l_lth_func : L.llvalue =
    L.declare_function "lth_func" l_lth_t the_module in
let l_gth_t : L.lltype =
    L.function_type i32_t [| L.pointer_type mpz_t; L.pointer_type mpz_t |] in
let l_gth_func : L.llvalue =
    L.declare_function "gth_func" l_gth_t the_module in
let l_leq_t : L.lltype =
    L.function_type i32_t [| L.pointer_type mpz_t; L.pointer_type mpz_t |] in
let l_leq_func : L.llvalue =
    L.declare_function "leq_func" l_leq_t the_module in
let l_or_t : L.lltype =
    L.function_type i32_t [| L.pointer_type mpz_t; L.pointer_type mpz_t |] in
let l_or_func : L.llvalue =
    L.declare_function "or_func" l_or_t the_module in
let l_and_t : L.lltype =
    L.function_type i32_t [| L.pointer_type mpz_t; L.pointer_type mpz_t |] in
let l_and_func : L.llvalue =
    L.declare_function "and_func" l_and_t the_module in
let l_geq_t : L.lltype =
    L.function_type i32_t [| L.pointer_type mpz_t; L.pointer_type mpz_t |] in
let l_geq_func : L.llvalue =
    L.declare_function "geq_func" l_geq_t the_module in
let l_rand_t : L.lltype =
    L.function_type (L.pointer_type point_t) [| L.pointer_type point_t; L.pointer_type mpz_t |] in
let l_rand_func : L.llvalue =
    L.declare_function "rand_func" l_rand_t the_module in

(* points and printing points *)
let init_lintpoint_t : L.lltype =
    L.function_type i32_t [| L.pointer_type point_t; L.pointer_type mpz_t;
                           L.pointer_type mpz_t; L.pointer_type poly_t |] in
let init_point_func : L.llvalue =
    L.declare_function "Point" init_lintpoint_t the_module in
let print_point_t : L.lltype =
    L.function_type i32_t [| L.pointer_type point_t; L.pointer_type point_t |] in
let print_point_func : L.llvalue =
    L.declare_function "printpt" print_point_t the_module in
let pt_add_t : L.lltype =
    L.function_type (L.pointer_type point_t) [| L.pointer_type point_t;
                                              L.pointer_type point_t_t |] in
let pt_add_func : L.llvalue =
    L.declare_function "ptadd" pt_add_t the_module in
let pt_mul_t : L.lltype =
    L.function_type (L.pointer_type point_t) [| L.pointer_type mpz_t;
                                              L.pointer_type point_t_t |] in
let pt_mul_func : L.llvalue =
    L.declare_function "ptmul" pt_mul_t the_module in
let pt_neg_t : L.lltype =
    L.function_type (L.pointer_type point_t) [| L.pointer_type point_t_t |] in
let pt_neg_func : L.llvalue =
    L.declare_function "ptneg" pt_neg_t the_module in
let pt_eq_t : L.lltype =
    L.function_type i32_t [| L.pointer_type point_t; L.pointer_type point_t_t |] in
let pt_eq_func : L.llvalue =
    L.declare_function "pteq" pt_eq_t the_module in
let pt_neq_t : L.lltype =
    L.function_type i32_t [| L.pointer_type point_t; L.pointer_type point_t_t |] in
let pt_neq_func : L.llvalue =
L.declare_function "pteq" pt_neq_t the_module in

(* polys and printing polys *)
let init_poly_t : L.lltype =
L.function_type i32_t [| L.pointer_type poly_t; L.pointer_type mpz_t; L.pointer_type mpz_t; L.pointer_type mpz_t |] in
let init_poly_func : L.llvalue =
L.declare_function "Poly" init_poly_t the_module in

let print_poly_t : L.lltype =
L.function_type i32_t [| L.pointer_type poly_t |] in
let print_poly_func : L.llvalue =
L.declare_function "printc" print_poly_t the_module in

(* Encoding and decoding strings for encryption *)
let encode_t : L.lltype =
L.function_type i32_t [| L.pointer_type mpz_t; string_t |] in
let encode_func : L.llvalue =
L.declare_function "encode" encode_t the_module in
let decode_t : L.lltype =
L.function_type string_t [| L.pointer_type mpz_t |] in
let decode_func : L.llvalue =
L.declare_function "decode" decode_t the_module in

(* Define each function (arguments and return type) so we can
call it even before we've created its body *)
let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
let function_decl m fdecl =
let name = fdecl.sname (* sfname *)
and formal_types =
Array.of_list ( let new_params = ( match fdecl.styp with
A. Lint -> (A.Lint, "sret") :: fdecl.sparams
| A. Point -> (A.Point, "sret") :: fdecl.sparams
| _ -> fdecl.sparams
) in
List.map (fun (t, n) -> match t with
A. Lint when n = "sret" -> L.pointer_type (ltype_of_typ t)
| A. Point when n = "sret" -> L.pointer_type (ltype_of_typ t)
| _ -> ltype_of_typ t) new_params)

in let ftype = L.function_type (match fdecl.styp with
A. Lint -> L.pointer_type mpz_t
| A. Point -> L.pointer_type point_t
| _ -> ltype_of_typ fdecl.styp) formal_types

in StringMap.add name (L.define_function name ftype the_module, fdecl) m in
List.fold_left function_decl StringMap.empty functions in

(* Fill in the body of the given function *)
let build_function_body fdecl =
let (the_function, _) = StringMap.find fdecl.sname function_decls in
let builder = L.builder_at_end context (L.entry_block the_function) in
let new_params = (match fdecl.styp with
A. Lint -> (A.Lint, "sret") :: fdecl.sparams
| A. Point -> (A.Point, "sret") :: fdecl.sparams
| _ -> fdecl.sparams
) in

List.map (fun (t, n) -> match t with
A. Lint when n = "sret" -> L.pointer_type (ltype_of_typ t)
| A. Point when n = "sret" -> L.pointer_type (ltype_of_typ t)
| _ -> ltype_of_typ t) new_params

in let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
and string_format_str = L.build_global_stringptr "%s\n" "fmt" builder
(* and point_format_str = L.build_global_stringptr "%[\n\n" "fmt" builder *)
in

(* Construct the function's "locals": formal arguments and locally
declared variables. Allocate each on the stack, initialize their
value, if appropriate, and remember their values in the "locals" map *)
let local_vars =
let add_formal m (t, n) p =
L.set_value_name n p;
let local = L.build_alloca (match t with
A. Lint when n = "sret" -> L.pointer_type (ltype_of_typ t)
| A. Point when n = "sret" -> L.pointer_type (ltype_of_typ t)
| _ -> ltype_of_typ t) n builder in
ignore (L.build_store p local builder);
StringMap.add n local m

(* Allocate space for any locally declared variables and add the *
 * resulting registers to our map *)
and add_local m (t, n) =
let local_var = L.buildalloca (ltype_of_typ t) n builder

StringMap.add n local_var m

let formals = List.fold_left2 add_formal StringMap.empty new_params (*fdecl.
sparams*)

(List.fold_left add_local formals fdecl.slocals in

(* Return the value for a variable or formal argument. *
Check local names first, then global names *)

let lookup n = try StringMap.find n local_vars

with Not_found -> StringMap.find n global_vars

in

(* Helper function to deal with unassigned lint lits *
Returns: mpz_t pointer to be used for function args *)

let llit_helper i =

let lstr = L.buildglobalstringptr i "string" builder

and space = L.buildalloca (ltype_of_typ A.Lint) "" builder in

let calls = ignore (L.buildcall linit_func
[| L.buildinboundsgep space [| L.constint i32_t 0 |] "" builder; lstr; L.
const_int i32_t 10 |]
"__gmpz_init_set_str" builder);

L.buildinboundsgep space [| L.constint i32_t 0 |] "" builder

in calls

(* how to free after done using *)

in

(* Helpful when writing geps *)

let zero = L.constint i32_t 0
in

(* Construct code for an expression; return its value *)

let rec expr builder ((stype, e) : sexpr) = match e with

SSTrlit i -> L.buildglobalstringptr i "string" builder

| SSSllit i -> llit_helper i (* Pointer to new mpz*)

| SSLit i -> L.constint i32_t i

| SSpollit (i, j, m) -> (* call our struct initialiser passing in loc of
initialisation *)

let e1 ' = expr builder i

and e2 ' = expr builder j

and e3 ' = expr builder p

and space = L.buildalloca point_t "tmp_pt" builder

in ignore (L.buildcall init_point_func [| space; e1 '; e2 '; e3 ' |] "Point"
builder); space

| SSpollit (i, j, m) -> (* call our struct initialiser passing in loc of
initialisation *)

let e1 ' = expr builder i

and e2 ' = expr builder j

and e3 ' = expr builder m

and space = L.buildalloca poly_t "tmp_poly" builder

in ignore (L.buildcall init_poly_func [| space; e1 '; e2 '; e3 ' |] "Poly"
builder); space

| SSexpr -> L.constint i32_t 0

| SId s -> (match stype with (* Might be better just to have StructType
adt? *)

A.Lint -> L.buildinboundsgep (lookup s) [| zero |] s

builder

| A.Point -> L.buildinboundsgep (lookup s) [| zero |] s

builder

| A.Poly -> L.buildinboundsgep (lookup s) [| zero |] s

builder

| _ -> L.buildload (lookup s) s builder

(* Here we have a pointer to mpz val *)

ignore (L.buildcall linitdup_func

in
```
| 341 | L.build_in_bounds_gep (lookup s) [ ] zero [ ] s builder; e1' | **
| 342 | builder); e1' |
| 343 | | SAssign (s, ((A.Point, _) as e1)) ->
| 344 | (* For point lits that already have stack allocated, we get element pointer
| 345 | then store *)
| 346 | let e1' = expr builder e1 in
| 347 | let val_ptr = L.build_in_bounds_gep e1' [ ] zero [ ] "" builder in
| 348 | let loaded = L.build_load val_ptr "" builder in
| 349 | ignore(L.build_store (lookup s) builder); e1' |
| 350 | | SAssign (s, ((A.Poly, _) as e1)) ->
| 351 | let e1' = expr builder e1 in
| 352 | let val_ptr = L.build_in_bounds_gep e1' [ ] zero [ ] "" builder in
| 353 | let loaded = L.build_load val_ptr "" builder in
| 354 | ignore(L.build_store (lookup s) builder); e1' |
| 355 | | SAssign (s, e) -> let e' = expr builder e in
| 356 | ignore (L.build_store e' (lookup s) builder); e' |
| 357 | | SAccess (s, idx) ->
| 358 | let outer_ptr = L.build_in_bounds_gep (lookup s) [ ] zero [ ]; L.const_int i32_t
| 359 | idx [ ] "outer" builder
| 360 | in
| 361 | L.build_in_bounds_gep outer_ptr [ ] zero [ ] "inner" builder
| 362 | | SBinop ((A.Point, _) as e1, operator, e2) ->
| 363 | let e1' = expr builder e1
| 364 | and e2' = expr builder e2 in
| 365 | (match operator with
| 366 | A.Add -> (L.build_call pt_add_func [ ] e1'; e2' ] "pt_add" builder)
| 367 | | A.Mul -> L.build_call pt_mul_func [ ] e2'; e1' ] "pt_mul" builder
| 368 | | _ -> raise (Failure "Operator not implemented for Point")
| 369 | (* special binop for lint times pt*)
| 370 | | SBinop (((A.Lint, _) as e1), operator, ((A.Point, _) as e2)) ->
| 371 | let e1' = expr builder e1
| 372 | and e2' = expr builder e2
| 373 | and tmp = llit_helper "0" in
| 374 | ignore(( match operator with
| 375 | A.Add -> L.build_call ladd_func [ ] tmp; e1'; e2' ] "__gmpz_add"
| 376 | builder
| 377 | | A.Sub -> L.build_call lsub_func [ ] tmp; e1'; e2' ] "__gmpz_sub"
| 378 | builder
| 379 | | A.Mul -> L.build_call lmul_func [ ] tmp; e1'; e2' ] "__gmpz_mul"
| 380 | builder
| 381 | | A.Div -> L.build_call ldiv_func [ ] tmp; e1'; e2' ] "__gmpz_tdiv_q"
| 382 | builder
| 383 | | A.Mod -> L.build_call lmod_func [ ] tmp; e1'; e2' ] "__gmpz_tdiv_r"
| 384 | builder
| 385 | | A.Pow -> L.build_call lpow_func [ ] tmp; e1'; e2' ] "__gmpz_pow_ui"
| 386 | builder
| 387 | | A.Inv -> L.build_call linv_func [ ] tmp; e1'; e2' ] "__gmpz_invert"
| 388 | builder (* add handling for inv does not exist *)
| 389 | | _ -> raise (Failure "Binary operator not implemented for Lint")
| 390 | ); tmp
| 391 | | SRelop ((A.Lint, _) as e1, operator, e2) ->
| 392 | let e1' = expr builder e1
| 393 | and e2' = expr builder e2 in (match operator with
| 394 | A.Beq -> L.build_call l_eq_func [ ] e1'; e2' ] "eq_func" builder
```
| A. Bneq -> L. build_call l_neq_func [| e1'; e2' |] "neq_func" builder |
| A. Lth -> L. build_call l_lth_func [| e1'; e2' |] "lth_func" builder |
| A. Gth -> L. build_call l_gth_func [| e1'; e2' |] "gth_func" builder |
| A. Leq -> L. build_call l_leq_func [| e1'; e2' |] "leq_func" builder |
| A. Geq -> L. build_call l_geq_func [| e1'; e2' |] "geq_func" builder |
| A. And -> L. build_call l_and_func [| e1'; e2' |] "and_func" builder |
| A. Or -> L. build_call l_or_func [| e1'; e2' |] "or_func" builder |
| _ -> raise (Failure "Relational operator not implemented for Lint") |

| SRelop ((A. Point , _) as e1 , operator , e2) -> |
| let e1' = expr builder e1 |
| and e2' = expr builder e2 in |
| (match operator with |
| A. Beq -> L. build_call pt_eq_func [| e1'; e2' |] "eq_func" builder |
| A. Bneq -> L. build_call pt_neq_func [| e1'; e2' |] "neq_func" builder |
| _ -> raise (Failure "Relational operator not implemented for Point")) |

| SRelop (e1 , operator , e2) -> |
| let e1' = expr builder e1 |
| and e2' = expr builder e2 in |
| (match operator with |
| A. And -> L. build_zext (L. build_icmp L. Icmp .Ne e1' (L. const_int i32_t 0) "tmp") |
| A. Bneq -> L. build_zext (L. build_icmp L. Icmp .Eq e1' e2' "tmp") |
| A. Lth -> L. build_zext (L. build_icmp L. Icmp .Slt e1' e2' "tmp") |
| A. Leq -> L. build_zext (L. build_icmp L. Icmp .Sle e1' e2' "tmp") |
| A. Gth -> L. build_zext (L. build_icmp L. Icmp .Sgt e1' e2' "tmp") |
| A. Geq -> L. build_zext (L. build_icmp L. Icmp .Sge e1' e2' "tmp") |
| _ -> raise (Failure "Relational operator not implemented")) |

| SBinop (e1 , operator , e2) -> |
| let e1' = expr builder e1 |
| and e2' = expr builder e2 in |
| (match operator with |
| A. Add -> L. build_add e1' e2' "tmp" builder |
| A. Sub -> L. build_sub e1' e2' "tmp" builder |
| A. Mul -> L. build_mul e1' e2' "tmp" builder |
| A. Div -> L. build_sdiv e1' e2' "tmp" builder |
| A. Mod -> L. build_srem e1' e2' "tmp" builder |
| _ -> raise (Failure "Binary operator not implemented")) |

| SUnop (op , ((A.Lint , _) as e)) -> |
| let e' = expr builder e |
and tmp = llit_helper "0" in
ignore (match op with
  | A. Neg -> L. build_call lneg_func [| tmp; e'|] "__gmpz_neg" builder
  | A. Not -> L. build_call lnot_func [| tmp; e'|] "lnot_func" builder
); tmp
| SUnop (op, (A. Point, _) as e)) ->
  let e' = expr builder e in
  (match op with
    | A. Neg -> (L. build_call pt_neg_func [| e' |] "ptneg" builder)
    | _ -> raise (Failure "Unary operator not implemented")
  )
| SUnop (op, (_, _) as e)) ->
  let e' = expr builder e in
  (match op with
    | A. Neg -> L. build_neg e' "tmp" builder
    | A. Not -> (L. build_zext
                  (L. build_icmp L. Icmp .Eq e' (L. const_int i32_t 0) "tmp"
                   builder)
                  i32_t "tmp" builder)
    | STrnop (e1, o1, e2, o2, e3) ->
      let e1' = expr builder e1
      and e2' = expr builder e2
      and e3' = expr builder e3
      and out = llit_helper "0" in
      ignore ((match o1, o2 with
        | A. Lpw, A. Pmd ->
          L. build_call lpowmod_func [| out; e1'; e2'; e3' |] "__gmpz_powm"
          builder
        | _ -> raise (Failure "Trinary operator not implemented")))
      out
    | SCall ("print", [e]) -> (* keep print delete printb printf*)
      L. build_call printf_func [| int_format_str; (expr builder e) |] "printf" builder
    | SCall ("prints", [e]) -> (*print string*)
      L. build_call printf_func [| string_format_str; (expr builder e) |] "printf" builder
    | SCall ("printf", [(_, e) as e1]) -> (* print pt *)
      let e1' = expr builder e1 in
      (match e with
        | SPlit _ -> L. build_call print_point_func [| L. build_in_bounds_gep e1' [| zero |] "" builder |
        | SLintlit i -> llit_helper i []
        | _ -> expr builder e1 in
      )
    | SCall ("printc", [(_, e) as e1]) -> (* print poly *)
      let e1' = expr builder e1 in
      (match e with
        | SPlit _ -> L. build_call print_poly_func [| L. build_in_bounds_gep e1' [| zero |] "" builder |
        | SLintlit i -> llit_helper i []
        | _ -> expr builder e1 in
      )
    | SCall ("printl", [(_, e) as ptr]) ->
      (* L. build_call lprintf_func [| expr builder e |] "printf" builder *)
      L. build_call lprintf_func (match e with
        | SId s -> [| L. build_in_bounds_gep (lookup s)
                     ([L. const_int i32_t 0] " builder) |]
        | SIntlit i -> [| llit_helper i |]
        | _ -> [| expr builder ptr |]) "printf" builder
    | SCall ("tolint", [e]) -> (* allocate some lint space and init with value *)
      let space = L. build_alloca mpz_t "tmp_lint" builder in
      let ptr = L. build_in_bounds_gep space [| zero |] "" builder
      and e' = expr builder e in
      ignore (L. build_call lcast_func [| ptr; e' |] "__gmpz_init_set_si" builder); ptr
    | SCall ("random", [e1;e2]) ->
      let rnd = llit_helper "0"
      and sed = expr builder e1
      and max = expr builder e2 in
      ignore (lbuild lrandom_func [| rnd; sed; max |] "rand_func" builder); rnd
    | SCall ("decode", [e]) ->
      let e' = expr builder e in
      L. build_call decode_func [| e' |] "decode" builder
    | SCall ("encode", [e]) ->
      let e' = expr builder e in
      and ret_space = L. build_alloca mpz_t " builder in
let ret_ptr = L.build_in_bounds_gep ret_space [| zero |] "" builder in
  ignore(L.build_call encode_func [| ret_ptr; e' |] "encode" builder); ret_ptr
| SCall (f, args) ->
  let (def, fdecl) = StringMap.find f function_decls in
  (* let args = match fdecl.styp with
      A.Lint -> (A.Lint, "sret") :: args
    | _ -> args in *)
  let llargs = List.rev (List.map (fun (ty, se) -> match ty with
    A.Lint -> L.build_load (expr builder (ty, se)) " lint_param " builder
    | A.Point -> L.build_load (expr builder (ty, se)) " pt_param " builder
    | _ -> expr builder (ty, se) ) (List.rev args )) in
  let result = ( match fdecl.styp with
    A.Void -> ""
  | _ -> f " _result") in
  let llargs = (match fdecl.styp with
    A.Lint -> let space = L.build_alloca mpz_t "sret_space" builder
               in
               L.build_in_bounds_gep space [| zero |] "" builder ::
               llargs
    | A.Point -> let space = L.build_alloca point_t "sret_space" builder
                 in
                 L.build_in_bounds_gep space [| zero |] "" builder ::
                 llargs
    | _ -> llargs ) in
  L.build_call fdef (Array.of_list llargs) result builder
| _ -> L.const_int i32_t 0 *)

(* LLVM insists each basic block end with exactly one "terminator" instruction that transfers control. This function runs "instr builder" if the current block does not already have a terminator. Used, e.g., to handle the "fall off the end of the function" case. *)
let add_terminal builder instr =
  match L.block_terminator (L.insertion_block builder ) with
    Some _ -> ()
  | None -> ignore (instr builder) in

(* Build the code for the given statement; return the builder for the statement’s successor (i.e., the next instruction will be built after the one generated by this call) *)
let rec stmt builder = function
  SBlock sl -> List.fold_left stmt builder sl
  | SExpr e -> ignore(expr builder e); builder
  | SReturn e -> ignore(match fdecl.styp with
    (* Special "return nothing" instr *)
      A.Void -> L.build_ret_void builder
    (* Add return statements for structs *)
    | A.Lint -> (let local_val = match (snd e) with
                  SId s -> L.build_load (expr builder e) " val_ptr "
                  and loaded = L.build_load (lookup (snd (List.hd new_params)))) "ret_ptr" builder
      | _ -> expr builder e;(*)
    let local_val = expr builder e
    and loaded = L.build_load (lookup (snd (List.hd new_params)))) "ret_ptr" builder
    ignore(L.build_call linitdup_func
          [loaded; local_val |] "ret_set" builder);
    L.build_ret loaded builder
    | A.Point ->
      let local_val = expr builder e in
      let value = L.build_load local_val "" builder
      and loaded = L.build_load (lookup (snd (List.hd new_params))) "ret_ptr" builder
      ignore(L.build_store value loaded builder);
      L.build_ret loaded builder
    (* Build return statement *)
    | _ -> L.build_ret (expr builder e) builder );

  SIf (predicate, then_stmt, else_stmt) ->
    let int_val = expr builder predicate in
    let bool_val = L.build_icmp L.icmp.Ge int_val (L.const_int i32_t 0) "tmp"
builder in
  (*L. const_int i1_t (* (if int_val = (L. const_int i32_t 0) then 0 else 1)*)
  ignore(match int_val with
  | (L. const_int i32_t _) -> 1
  | _ -> raise(Failure "case")
  |)
  *)
let merge_bb = L. append_block context "merge" the_function in
  let build_br_merge = L. build_br merge_bb in (* partial function *)
let then_bb = L. append_block context "then" the_function in
  add_terminal (stmt (L. builder_at_end context then_bb ) then_stmt)
  build_br_merge;
let else_bb = L. append_block context "else" the_function in
  add_terminal (stmt (L. builder_at_end context else_bb) else_stmt)
  build_br_merge;
  ignore(L. build_cond_br bool_val then_bb else_bb builder);
L. builder_at_end context merge_bb

  | SWhile (predicate, body) ->
  let pred_bb = L. append_block context "while" the_function in
  ignore(L. build_br pred_bb builder);
  let body_bb = L. append_block context "while_body" the_function in
  add_terminal (stmt (L. builder_at_end context body_bb) body)
  (L. build_br pred_bb);
  let pred_builder = L. builder_at_end context pred_bb in
  let int_val = expr pred_builder predicate in
  let bool_val = (L. build_icmp L. Icmp .Ne int_val (L. const_int i32_t 0)) "tmp" pred_builder in
  let merge_bb = L. append_block context "merge" the_function in
  ignore(L. build_cond_br bool_val body_bb merge_bb pred_builder);
  let merge_bb = L. append_block context merge_bb
  ((! Implement for loops as while loops *)
  | SFor (e1, e2, e3, body) -> stmt builder
  ( SBlock [SExpr e1 ; SWhile (e2, SBlock [body ; SExpr e3]) ] )
  in

  (* Build the code for each statement in the function *)
let builder = stmt builder (SBlock fdecl . sbody) in
  (**#TODO: We need some code to clear our lints so free all at the end of this
  function
  We will iterate through our locals map and add clears at the end of each of them
  have map that contains all the lints (does local vars work for this?) if not
  we need to make new map.
  Function called for each lint, check elt match type with A. Lint getelementptr
  inbounds
  and pass to lclear_t same way we pass stuff to println.
  *)
  (* Add a return if the last block falls off the end *)
add_terminal builder (match fdecl . styp with
  | A. Void -> L. build_ret_void
  | t -> L. build_ret (L. const_int (ltype_of_typ t) 0))
  in
List.iter build_function_body functions;

the_module

9.7 gmpfunc.c
  // This file will be used to interface with OCaml LLVM
  #include <stdio.h>
  #include <gmp.h>
  #include <stdlib.h>
```c
#include <string.h>
#include <time.h>
#include "structs.h"

void printl(mpz_t n)
{
    mpz_out_str(stdout, 10, n);
    printf("\n");
}

int rand_func(mpz_t rnd, mpz_t seed, mpz_t max)
{
    /* rand() into mpzt */
    // mpz_t newseed;
    // mpz_init(newseed);
    if(mpz_sgn(seed) == 0)
    {
        srand(time(0));
        mpz_set_ui(seed, rand());
    }
    gmp_randstate_t state; /* initialize state */
    gmp_randinit_mt(state); /* set set state to use the Mersenne Twister Algorithm */
    gmp_randseed(state, seed); /* seed the state using user input*/
    mpz_urandomm(rnd, state, max); /* generate random int*/
    gmp_randclear(state);
    return(0);
}

char *sub(char *left, char *right)
{
    mpz_t n1;
    mpz_t n2;
    mpz_init(n1);
    mpz_init(n2);
    if (mpz_set_str(n1, left, 10) != 0){
        printf("Failed to assign number");
        mpz_clear(n1);
        mpz_clear(n2);
        exit(1);
    }
    if (mpz_set_str(n2, right, 10) != 0) {
        printf("Failed to assign number");
        mpz_clear(n1);
        mpz_clear(n2);
        exit(1);
    }
    mpz_sub(n1, n1, n2);
    char *ret_str = mpz_get_str(NULL, 10, n1);
    mpz_clear(n1);
    mpz_clear(n2);
    return ret_str;
}

int eq_func(mpz_t x, mpz_t y){
    if(mpz_cmp(x, y) == 0){
        return 1;
    } else{
        return 0;
    }
}

int neq_func(mpz_t x, mpz_t y){
    if(mpz_cmp(x, y) == 0){
        return 0;
    } else{
        return 1;
    }
```
int lth_func (mpz_t x, mpz_t y){
    if( mpz_cmp (x, y) < 0){
        return 1;
    }
    else{
        return 0;
    }
}

int gth_func (mpz_t x, mpz_t y){
    if( mpz_cmp (x, y) > 0){
        return 1;
    }
    else{
        return 0;
    }
}

int leq_func (mpz_t x, mpz_t y){
    if( mpz_cmp (x, y) <= 0){
        return 1;
    }
    else{
        return 0;
    }
}

int geq_func (mpz_t x, mpz_t y){
    if( mpz_cmp (x, y) >= 0){
        return 1;
    }
    else{
        return 0;
    }
}

int lnot_func (mpz_t out , mpz_t in){
    if( mpz_sgn (in) == 0)
    {
        mpz_set_str (out , "1", 10) ;
    }
    else
    {
        mpz_set_str (out , "0", 10) ;
    }
    return 0;
}

int and_func (mpz_t x, mpz_t y){
    if( mpz_get_si (x) == 0 || mpz_get_si (y) == 0)
    {
        return 0;
    }
    else
    {
        return 1;
    }
}

int or_func (mpz_t x, mpz_t y){
    if( mpz_get_si (x) == 0 && mpz_get_si (y) == 0)
    {
        return 0;
    }
    else
    {
        return 1;
    }
}
```c
# ifdef BUILD_TEST
int main()
{
    // Create a lint through assignment to an id
    // char *id1 = "1934759237458927349587234858395728";
    // printf("n = ");
    // printl(id1);
    // printf("\n");

    // // Do some operation(s) on lint
    // printf("Squaring:\n");
    // char *fun = pow(id1, 2);
    // printf("%s", fun);
    // printf("\n");

    // printf("Adding\n");
    // char *added = add("4035273409750284735027430528934750", "139487619823469187364916398427");
    // printf("%s", added);
    // printf("\n");

    // // clean up
    // free(fun);
    // free(added);
    return 0;
}
#endif
```

9.8 structs.h

```c
#define STRUCTS_H

struct poly {
    mpz_t x_coeff;
    mpz_t c;
    mpz_t mod;
};

struct point {
    mpz_t i;
    mpz_t j;
    struct poly *curve;
};

#define STRUCTS_H
```

9.9 structs.c

```c
#include <stdio.h>
#include <gmp.h>
#include <stdlib.h>
#include <string.h>
#include "structs.h"

// POLYS
void Poly struct poly *p, mpz_t x_coeff, mpz_t c, mpz_t mod)
{
    mpz_init_set(p->x_coeff, x_coeff);
    mpz_init_set(p->c, c);
    mpz_init_set(p->mod, mod);
}

void printc(struct poly *p)
{
```

92
printf ("[=");
mpz_out_str (stdout, 10, p->x_coeff);
printf (",");
mpz_out_str (stdout, 10, p->c);
printf (" : ");
mpz_out_str (stdout, 10, p->mod);
printf ("]\n");
}

void Point (struct point *p, mpz_t i, mpz_t j, struct poly * curve) {
  mpz_init_set(p->i, i);
  mpz_init_set(p->j, j);
  p->curve = curve;
}

void printpt (struct point *p) {
  printf ("[");
  mpz_out_str (stdout, 10, p->i);
  printf (",");
  mpz_out_str (stdout, 10, p->j);
  printf ("] & ");
  printc(p->curve);
}

struct point * ptadd ( struct point *p1 , struct point *p2);
/*
char *printpt (struct point p){
  printf("[");
  mpz_out_str (stdout, 10, p.i);
  printf (" ;");
  mpz_out_str (stdout, 10, p.j);
  printf (" ]\n");
}*/

struct point * ptmul ( mpz_t n, struct point *p1) {
  /* struct poly * curve = ( struct poly *) malloc ( sizeof ( struct poly ));
  struct point * sum = ( struct point *) malloc ( sizeof ( struct point ));
  mpz_t xcoeff ;
  mpz_init_set ( xcoeff , p1 -> curve -> x_coeff );
  mpz_t c;
  mpz_init_set (c, p1 -> curve ->c);
  mpz_t mod;
  mpz_init_set (mod, p1 -> curve ->mod);
  Poly( curve, xcoeff, c, mod);*/
  struct point * product = p1;
  // copy n into new mpz_t
  mpz_t i;
  mpz_init(i);
  mpz_sub_ui(i, n);
  while( mpz_sgn(i) != 0) {
    product = ptadd(product, p1);
    mpz_sub_ui(i, n);
  }
  return product;
}

/*
struct point * ptadd ( struct point *p1 , struct point *p2){
  struct poly * curve = ( struct poly *) malloc ( sizeof ( struct poly ));
  struct point * sum = ( struct point *) malloc ( sizeof ( struct point ));
  mpz_t xcoeff ;
  mpz_init_set ( xcoeff , p1 -> curve -> x_coeff );
  mpz_t ycoeff ;
  mpz_init_set ( ycoeff , p1 -> curve -> y_coeff );
  struct point * result = ( struct point *) malloc ( sizeof ( struct point ));
  Poly ( curve , xcoeff , ycoeff , p1 -> curve -> mod );
  result = ptmul ( 2 , result );
  return result;
}*/

void printpt (struct point *p) {
  printf ("[");
  mpz_out_str (stdout, 10, p->i);
  printf (" ,");
  mpz_out_str (stdout, 10, p->j);
  printf ("] & ");
  printc(p->curve);
}

struct point * ptadd ( struct point *p1 , struct point *p2);
/*
char *printpt (struct point p){
  printf("[");
  mpz_out_str (stdout, 10, p.i);
  printf (" ;");
  mpz_out_str (stdout, 10, p.j);
  printf (" ]\n");
}*/

struct point * ptmul ( mpz_t n, struct point *p1) {
  /* struct poly * curve = ( struct poly *) malloc ( sizeof ( struct poly ));
  struct point * sum = ( struct point *) malloc ( sizeof ( struct point ));
  mpz_t xcoeff ;
  mpz_init_set ( xcoeff , p1 -> curve -> x_coeff );
  mpz_t c;
  mpz_init_set (c, p1 -> curve ->c);
  mpz_t mod;
  mpz_init_set (mod, p1 -> curve ->mod);
  Poly( curve, xcoeff, c, mod);*/
  struct point * product = p1;
  // copy n into new mpz_t
  mpz_t i;
  mpz_init(i);
  mpz_sub_ui(i, n);
  while( mpz_sgn(i) != 0) {
    product = ptadd(product, p1);
    mpz_sub_ui(i, n);
  }
  return product;
}

/*
struct point * ptadd ( struct point *p1 , struct point *p2){
  struct poly * curve = ( struct poly *) malloc ( sizeof ( struct poly ));
  struct point * sum = ( struct point *) malloc ( sizeof ( struct point ));
  mpz_t xcoeff ;
  mpz_init_set ( xcoeff , p1 -> curve -> x_coeff );
  mpz_t ycoeff ;
  mpz_init_set ( ycoeff , p1 -> curve -> y_coeff );
  struct point * result = ( struct point *) malloc ( sizeof ( struct point ));
  Poly ( curve , xcoeff , ycoeff , p1 -> curve -> mod );
  result = ptmul ( 2 , result );
  return result;
}*
mpz_t c;
mpz_init_set(c, p1->curve->c);

mpz_t mod;
mpz_init_set(mod, p1->curve->mod);

Poly(curve, xcoeff, c, mod);
return ptaddhelper(sum, p1, p2);
}*/

int pteq(struct point *p1, struct point *p2)
{
  if (mpz_cmp(p1->i, p2->i) == 0 &&
      mpz_cmp(p1->j, p2->j) == 0 &&
      mpz_cmp(p1->curve->mod, p2->curve->mod) == 0 &&
      mpz_cmp(p1->curve->c, p2->curve->c) == 0 &&
      mpz_cmp(p1->curve->x_coeff, p2->curve->x_coeff) == 0)
  {
    return 1;
  } else {
    return 0;
  }
}

int ptneq(struct point *p1, struct point *p2)
{
  if (mpz_cmp(p1->i, p2->i) != 0 ||
      mpz_cmp(p1->j, p2->j) != 0 ||
      mpz_cmp(p1->curve->mod, p2->curve->mod) != 0 ||
      mpz_cmp(p1->curve->c, p2->curve->c) != 0 ||
      mpz_cmp(p1->curve->x_coeff, p2->curve->x_coeff) != 0)
  {
    return 1;
  } else {
    return 0;
  }
}

struct point *ptneg(struct point *p1)
{
  struct poly *curve = (struct poly *)malloc(sizeof(struct poly));
  struct point *neg = (struct point *)malloc(sizeof(struct point));
  
  mpz_t xcoeff;
  mpz_init_set(xcoeff, p1->curve->x_coeff);
  
  mpz_t c;
  mpz_init_set(c, p1->curve->c);
  
  mpz_t mod;
  mpz_init_set(mod, p1->curve->mod);
  
 Poly(curve, xcoeff, c, mod);
  if (mpz_sgn(p1->i) == -1 && mpz_sgn(p1->j) == -1) {
    Point(neg, p1->i, p1->j, curve);
  } else {
    mpz_t inv;
    mpz_init(inv);
    mpz_neg(inv, p1->j);
    mpz_mod(inv, inv, mod);
    
    Point(neg, p1->i, inv, curve);
  }
  
  mpz_clear(inv);

  return neg;
}

mpz_clear(xcoeff);
struct point *ptadd(struct point *p1, struct point *p2)
{
    struct poly *curve = (struct poly *)malloc(sizeof(struct poly));
    struct point *sum = (struct point *)malloc(sizeof(struct point));

    mpz_t xcoeff;
    mpz_init_set(xcoeff, p1->curve->x_coeff);
    mpz_t c;
    mpz_init_set(c, p1->curve->c);

    mpz_t mod;
    mpz_init_set(mod, p1->curve->mod);

    Poly(curve, xcoeff, c, mod);

    mpz_t zero;
    mpz_init_set_str(zero, "0", 10);

    mpz_t p3x;
    mpz_t p3y;
    mpz_init(p3x);
    mpz_init(p3y);

    /* if pt is -1, -1 -> pt at infinity acts as identity element
     * return other point
     */
    if( mpz_sgn(p1->i) == -1 && mpz_sgn(p1->j) == -1 ){
        mpz_set(p3x, p2->i);
        mpz_set(p3y, p2->j);
        Point(sum, p3x, p3y, curve);
    }
    else if ( mpz_sgn(p2->i) == -1 && mpz_sgn(p2->j) == -1 ){
        mpz_set(p3x, p1->i);
        mpz_set(p3y, p1->j);
        Point(sum, p3x, p3y, curve);
    }
    else{
        /* build local x and y coords */
        mpz_t p1x;
        mpz_t p1y;
        mpz_t p2x;
        mpz_t p2y;
        mpz_init(p1x);
        mpz_init(p1y);
        mpz_init(p2x);
        mpz_init(p2y);

        mpz_mod(p1x, p1->i, mod);
        mpz_mod(p1y, p1->j, mod);
        mpz_mod(p2x, p2->i, mod);
        mpz_mod(p2y, p2->j, mod);

        /* check if they are inverses of one another */
        mpz_t neg;
        mpz_init(neg);
        mpz_neg(neg, p2y);
        if(mpz_congruent_p(p1y, neg, mod))
        {
            mpz_set_str(p3x, "-1", 10);
        }
    }
```c
mpz_set_str(p3y, "-1", 10);
Point( sum, p3x, p3y, curve);

/* mpz_clear(neg);
mpz_clear(p1x);
mpz_clear(p1y);
mpz_clear(p2x);
mpz_clear(p2y);*/

/* mpz_clear(xcoeff);
mpz_clear(c);
mpz_clear(z: qero);
mpz_clear(mod);*/

// return sum;
}
else {
 // slope

mpz_t m;
mpz_init(m);

/* if pts are not the same */
if( mpz_cmp(p1x, p2x) != 0 || mpz_cmp(p1y, p2y) != 0) {
  mpz_t tmpy;
  mpz_t tmpx;
  mpz_init(tmpy);
  mpz_init(tmpx);

  mpz_sub(tmpy, p2y, p1y);
  mpz_sub(tmpy, tmpy, mod);
  mpz_sub(tmpx, p2x, p1x);
  mpz_mod(tmpx, tmpx, mod);

  mpz_invert(tmpx, tmpx, mod);
  mpz_mul(m, tmpy, tmpx);
  mpz_mod(m, m, mod);

  mpz_clear(tmpy);
  mpz_clear(tmpx);
} else { /* if points are same */
  mpz_t tmpx;
  mpz_t tmpy;
  mpz_init(tmpx);
  mpz_init(tmpy);

  mpz_mul(tmpx, m, m);
  mpz_mod(tmpx, tmpx, mod);
  mpz_sub(tmpx, tmpx, m);
  mpz_add(tmpx, tmpx, xcoeff);
  mpz_mul_si(tmpy, p1y, (long) 2);
  mpz_mod(tmpy, tmpy, mod);
  mpz_invert(tmpy, tmpy, mod);
  mpz_mul(m, tmpx, tmpy);
  mpz_mod(m, m, mod);

  mpz_clear(tmpx);
  mpz_clear(tmpy);
}

/* find p3x */
mpz_t tmp;
mpz_init(tmp);
mpz_mod(tmp, m, m);
mpz_mul(tmp, m, m);
mpz_sub(tmp, tmp, p1x);
mpz_sub(tmp, tmp, p2x);
mpz_mod(tmp, tmp, mod);
mpz_set(p3x, tmp);
```

/* find p3y */
mpz_sub(tmp, p1x, p3x);
mpz_mul(tmp, tmp, m);
mpz_sub(tmp, tmp, p1y);
mpz_mod(tmp, tmp, mod);
mpz_set(p3y, tmp);

/* build pt */
Point(sum, p3x, p3y, curve);

mpz_clear(m);
mpz_clear(tmp);
}
mpz_clear(neg);
mpz_clear(pix);
mpz_clear(p1y);
mpz_clear(p2x);
mpz_clear(p2y);
}
mpz_clear(xcoeff);
mpz_clear(c);
mpz_clear(zero);
mpz_clear(mod);
mpz_clear(p3x);
mpz_clear(p3y);
return sum;
/* int i, j;
int m;
int b = 1;

if(p1.i == p2.i && p1.j == p2.j){
m = (3*(p1.i)^2 + b)/(2*p1.j);
}
else{
m = (p2.j-p1.j)/(p2.i-p1.i);
}
i = m^2 - p1.i - p2.i;
j = m*(p1.i - i) - p1.j;
return Point(i, j);*/

9.10 input.c
#include <stdio.h>
#include <gmp.h>
#include <string.h>
#include <stdlib.h>

// takes in pointer to mpz to update
void encode(mpz_t res, char *in)
{
    // keep output buff that handles padding length and null terminator
    // printf("%s\n", in);
    char outBuf[3 * strlen(in) + 1];
    int i;
    outBuf[0] = '\0';
    for (i = 0; i < strlen(in); i++) {
        int c = (int) in[i];
        char temp[4];
        sprintf(temp, "%03d", c);
        strcat(outBuf, temp, strlen(temp));
}
// printf("%s\n", outBuf);
} mpz_init_set_str(res, outBuf, 10);

char *decode(mpz_t in)
{
    int i;
    char *lintStr = mpz_get_str(NULL, 10, in);
    int padLen = 3 - (strlen(lintStr) % 3);
    char *tmp = (char *) malloc(strlen(lintStr)+padLen+1); // will leak unless freed
    for (i = 0; i < padLen; i++)
        tmp[i] = '0';
    tmp[i] = '\0';
    strcat(tmp, lintStr, strlen(lintStr));
    free(lintStr);
    int newlength = strlen(tmp)/3 + 1;
    char *ret = (char *) malloc(newlength);
    for (i = 0; i < newlength; i++) {
        char buf[4];
        strncpy(buf, tmp +3*i, 3);
        tmp[3] = '\0';
        char c = (char) atoi(buf);
        // printf("%c", c);
        ret[i] = c;
    }
    free(tmp);
    return ret;
}

#ifdef BUILD_TEST
int main()
{
    mpz_t res;
    // mpz_init(res);
    char testStr[] = "HelloWorld";
    encode(res, testStr);
    mpz_out_str(stdout, 10, res);
    printf("\n");
    char *retVal = decode(res);
    printf("%s\n", retVal);
    free(retVal);
}
#endif

9.11 prime.ml

(* Compiler command centre: tell sequence of actions here *)

type action = Ast | Sast | LLVM_IR | Compile

let () = (* don't care about return type *)
let action = ref Compile in (* set default? *)
let set_action a () = action := a in
let options = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST"),
    ("-s", Arg.Unit (set_action Sast), "Print the SAST"),
    ("-l", Arg.Unit (set_action LLVM_IR), "Print LLVM"),
    ("-c", Arg.Unit (set_action Compile),
        "Check and print the generated LLVM IR (default)"),
] in (* Only one mode for now *)
let usage_msg = "usage: ./prime.native [-a|-c] <filename>" in
let channel = ref stdin in
(* take the options and a function that takes filename and opens it for reading *)
let parser_options (fun filename -> channel := open_in filename) usage_msg;
(* Start reading input *)
let lexbuf = Lexing.from_channel !channel in (* ! operator dereferences *)
(* Construct AST *)
let ast = Parser.program Scanner.token lexbuf in
match !action with
| Ast  -> print_string (Ast.string_of_program ast)
| _   -> let sast = Semant.check ast in
  match !action with (* add other options to stop at later *)
  | Ast  -> ()
  | Sast -> print_string (Sast.string_of_sprogram sast)
  | LLVM_IR -> let modu = Codegen.translate sast in
               print_string (Llvm.string_of_llmodule modu)
  | Compile -> let modu =
               Codegen.translate sast in
               llvm_analysis.assert_valid_module modu;
               print_string (llvm.string_of_llmodule modu)

9.12 Makefile

.PHONY : test
test : all test_all.sh
   ./test_all.sh

.PHONY : all
all : clean gmp prime.native gmpfunc.o structs.o

# this will serve to install the GNU multiple precision library onto our system
.PHONY : gmp
gmp :
   apt install -y libgmp-dev

# We will now make the compiler
prime.native : codegen.ml sast.ml ast.ml semant.ml scanner.mll parser.mly
   opam config exec -- \ocambuild -use=ocamlfind prime.native

# Test the GMP calls we build
gmpfunc: gmp gmpfunc.c
c c -o gmpfunc -DBUILD_TEST gmpfunc.c -lgmp

gmpfunc.o: gmp gmpfunc.c
c c -c gmpfunc.c

structs : structs.c
c c -o structs -DBUILD_TEST structs.c -lgmp

structs.o : structs.c
c c -c structs.c

input : gmp input.c
c c -o input -DBUILD_TEST input.c -lgmp

input.o : input.c
c c -c input.c

# Some old stuff:
prime : parser.cmo scanner.cmo prime.cmo
   ocamlc -o prime "$"

%.cmo : %.ml
   ocamlc -c $<

%.cmi : %.ml
   ocamlc -c $<

scanner.ml : scanner.mll
   ocamllex "$"

parser.ml parser.mli : parser.mly
   ocamlyacc "$"

# run the tests (without outputting to file)
prime.out : prime prime.tb
   ./prime < prime.tb
# Dependencies from ocamldep
58 prime.cmo : scanner.cmo parser.cmi ast.cmi
59 prime.cmx : scanner.cmx parser.cmx ast.cmi
60 parser.cmo : ast.cmi parser.cmi
61 parser.cmx : ast.cmi parser.cmi
62 scanner.cmo : parser.cmi
63 scanner.cmx : parser.cmx

# TARFILES = README Makefile scanner.mll ast.mli parser.mly prime.ml prime.tb
69 hw1.tar.gz : $(TARFILES)
70 cd .. && tar zcf hw1/hw1.tar.gz $(TARFILES):%=hw1/%

.PHONY : clean
clean :
r -rf *.cmo *.cmx parser.ml parser.mli scanner.ml prime.out prime
77 rm -rf *.exe *.ll *.s *.test *.diff a.out gmpfunc gmpfunc.o structs structs.o input input.o
78 opam config exec -- 
79 ocamlbuild -clean

9.13 test_file.sh
1 #!/bin/bash
2 ./prime.native $1.pr > $1.ll
3 llc -relocation-model=pic $1.ll > $1.s
4 cc -c gmpfunc.c
5 cc -c structs.c
6 gcc -o $1.exe $1.s gmpfunc.o structs.o -lgmp
7 ./$1.exe
8 rm $1.ll;
9 rm $1.s;
10 rm $1.exe;

9.14 test_all.sh
1 #!/bin/bash
2 # time limit on operations
3 ulimit -t 30
4 logfile=tests.log
5 rm -rf $logfile
6 error=0
7 exitcode=0

IsError () {
10 if [ $error -eq 0 ] ; then
11 echo "FAILED"
12 error=1
13 fi
14 # print out what we failed
15 echo "$1"
16 }
17
18 Difference () {
19 echo diff -b -q $1 $2 "">" $logfile 1>&2
20 diff -b "$1" "$2" "$1.diff" 2>&1 || {
21 IsError "Difference in $1"
22 }
23 }
24
25 # Run a command retaining error code
26 Run () {
27 echo $* 1>&2
28 eval $* || {
29 IsError "$1 Failed (cmd: $*)"
30 return 1
31 }
32 }
Test() {
    error=0
    filename="$(basename -- "$1")"
    # filename="$(basename%.*)"
    echo -n "Test: $filename 
    # newline between tests
    echo 1>&2
    echo "##### Testing $1 #####" 1>&2
    # Run the various compilation parts
    Run "./prime.native " "$1" " "$filename.ll" &&
    Run "llc" "-relocation-model=pic" "$filename.ll" " "$filename.s" &&
    Run "cc" "-o" "$filename.exe" "$filename.s" "gmpfunc.o" "structs.o" "input.o" "-lgmp"
    &&
    Run "./$filename.exe" "$filename.test" &&
    Difference $filename.test ./tests/$filename.out
    if [ $error -eq 0 ] ; then
        echo "OK"
        echo "##### Success" 1>&2
    else
        echo "##### FAIL" 1>&2
        exitcode=$error
    fi
}

RunFail() {
    echo $* 1>&2
    # Use short circuit && operator
    eval $* && {
        IsError "failed: $* did not show error"
        return 1
    }
    return 0
}

TestFail() {
    error=0
    filename="$(basename -- "$1")"
    # filename="$(basename%.*)"
    echo -n "Test: $filename 
    # newline between tests
    echo 1>&2
    echo "##### Testing $1 #####" 1>&2
    # This is a file case so should not get past the compiler
    RunFail "./prime.native" "$1" "$filename.test" "$logfile" &&
    Difference $filename.test ./tests/$filename.out
    if [ $error -eq 0 ] ; then
        echo "OK"
        echo "##### Success" 1>&2
    else
        echo "##### FAIL" 1>&2
        exitcode=$error
    fi
}

# make sure C files ready
# Compile/link in gmpfunc file
cc -c gmpfunc.c
cc -c structs.c
cc -c input.c

# Run test_hello.pr
# check if specific files to test
if [ $# -ge 1 ]
then
    # provided specific files to test
    files="@
else
    files="tests/*.pr"
fi

# run positive tests for now
for file in $files
do
    if [[ $file != *fail*.pr ]];
    then
        Test $file 2>> $logfile
    else
        TestFail $file 2>> $logfile
    fi
done

# clean up ()
# rm -rf *.exe *.test *.ll *.s

# print out so we can see return at the end
cat $logfile
exit $exitcode