FRAC: Recursive Art Compiler

- **Anne Zhang** `az2350`
- **Kunal Kamath** `kak2211`
- **Calvin Li** `ctl2124`
- **Justin Chiang** `jc4127`

Table of Contents

1. Introduction

FRAC is a domain-specific programming language that enables the programmer to generate fractals in the form of bitmap image files (BMPs). The language is meant for those interested in the mathematical manipulation of recursively generated images like fractals. We designed this language to be simple, intuitive, and a joy to use!

Our language uses the L-system method, which uses grammars (similar to those we learned in COMS W3261) to generate fractals. A basic FRAC program is composed of function and grammar declarations. The true power of FRAC is its ability to use simple L-system grammars to generate both static and dynamic fractal images. As you will see in examples of our compiled C code, the code for generating a fractal in C is incredibly verbose and repetitive. Grammar declarations in FRAC allow the programmer to easily play around with different rules and commands to see what fun new fractals they can generate.

2. Language Tutorial

2.1 Compiling and Running

Run `make` in the top level directory of our source code to compile the `frac` compiler. Then, simply run the shell script `run.sh` with a filename argument to compile and run your FRAC code. The filename must have extension .frac.

```
$ ./run.sh test.frac
```

If your FRAC program generates an image, it requires the turtle graphics C library (the necessary files, `turtle.c` and `turtle.h` are included in our source code directory). Additionally, if you are using the `grow()` function to generate an animated GIF, it requires the `GraphicsMagick` and `gifsicle` libraries. The GIF generation libraries are somewhat large, so we did not include them in our submitted directory. Please refer to our README to find instructions on downloading and installing those tools.

The idea of turtle graphics is that there is a “turtle” that walks around the screen with a pen, and it is given commands to move around, drawing an image based on its movement. On compilation, a C program is generated in which each gram is separated into two separate functions, one which determines the start state based on the init string, and one that recursively evaluates the rules based on the number of iterations and the symbol being used. The terminals of each rule map to one of three functions: `turtle_turn_right()`, `turtle_turn_left`, and `turtle_forward()`, corresponding to `rturn()`, `lturn()` and `move()` from FRAC, respectively. The main function generates a 2000x2000 grid of pixels by default, which should be enough space for most drawings. It also saves a static bmp image if `draw()` is called on the gram from the FRAC program, or it strings together multiple bmp files based on each iteration if `grow()` is called, in which case gifsicle is used to turn them into a GIF. Finally, it cleans up the memory used with `turtle_cleanup()`.

2.2 Writing a FRAC Program

At a high level, FRAC programs are composed of function and grammar declarations. A main function is required in every program. Grammars must be declared and defined outside of functions, thus giving every grammar a global scope. The main function is the entry point of the program.

The two system functions `draw(gram g, int n)` and `grow(gram g, int n)` are called on defined grammars to generate images. Grammars can be defined without being used in a system function call, but if you do that your grammar will be lost in the ether, which we do not recommend. `draw()` can be called multiple times, but `grow()` must only be called once. Both functions can only be called in the main function. A call to the `draw(gram g, int n)` system function will generate a BMP image from the specified grammar g. The integer n specifies how many times the recursive rules defined in g are evaluated, which affects the size and complexity of the generated fractal. A call to the `grow(gram g, int n)` function will generate a series of BMPs that will be linked into a GIF, showing the growth of the fractal generated by grammar g. Each frame of the GIF is an image from iteration i of the fractal generation, where i <= n.

Below are several example programs, from basic to somewhat complex, which will help give you a sense of how FRAC works.

2.3 Example Programs

2.3.1 Hello World

This simple program prints "hello world":

```
main() {
      print("hello world");
}
```

2.3.2 GCD

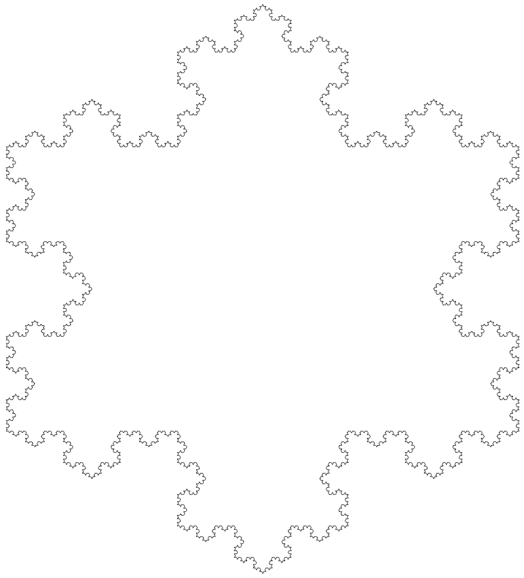
This program calculates and prints the greatest common divisor (GCD) of 8 and 12:

```
gcd(int x, int y) {
    while(x != y) {
        if(x > y) {
          a = a-b;
        }
        else {
            b = b-a;
        }
    }
    return a;
}

main() {
    int n = gcd(8, 12);
    print(n);
}
```

2.3.3 Koch Snowflake (Static BMP)

As our language is designed to facilitate generation of fractals, the best way to program in FRAC is to define grammars and draw them! For example, you can create a Koch snowflake using the following FRAC program:

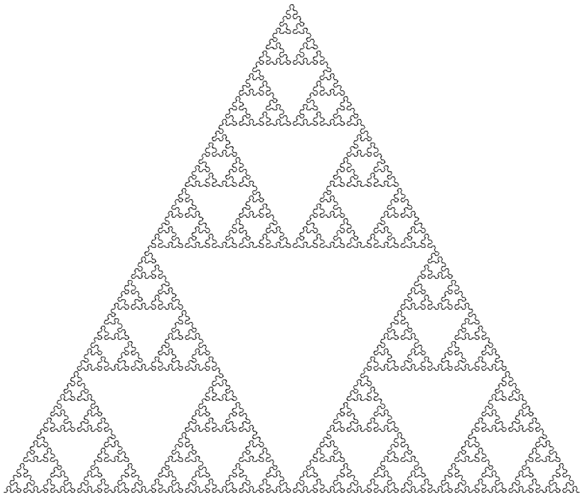
```
gram koch = {
    alphabet: [F, p, m],
    init: 'F p p F p p F',
    rules: {
        'F' -> 'F m F p p F m F',
        'F' -> move(1),
        'm' -> rturn(60),
        'p' -> lturn(60)
    }
}

main() {
    draw(koch, 6);
}
```



2.3.4 Sierpinski Triangle (Static BMP)

Another fun fractal is the Sierpinski triangle. The grammar for generating a Sierpinski triangle is somewhat more complex than the Koch snowflake (but still easy to write):

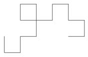
```
gram sierp = {
    alphabet: [A, B, p, m],
    init: 'A',
    rules: {
        'A' -> 'p B m A m B p',
        'A' -> move(1),
        'B' -> 'm A p B p A m',
        'B' -> move(1),
        'p' -> lturn(60),
        'm' -> rturn(60)
    }
}

main() {
    draw(sierp, 9);
}
```

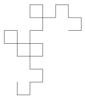


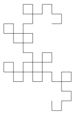
2.3.5 Heighway Dragon (Animated GIF)

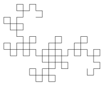
One of the most exciting things that FRAC can do is create animated GIFs depicting the growth of the fractal that you are generating. Here is an example of a FRAC program that uses the `grow()` system function to create a dynamic GIF of the Heighway dragon fractal (one of our favorites!). Each frame of the resulting GIF is included here, but in our source code you can open up the GIF in a browser to view the animated version.

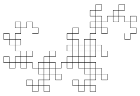
```
gram dragon = {
    alphabet: [F, X, Y, p, m],
    init: 'F X',
    rules: {
        'X' -> 'X p Y F',
        'Y' -> 'F X m Y',
        'F' -> move(5),
        'p' -> lturn(90),
        'm' -> rturn(90)
    }
}

main() {
  grow(dragon, 15);
}
```

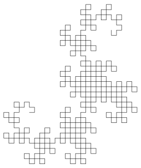


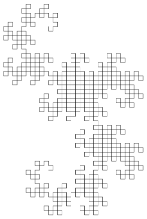


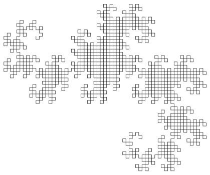


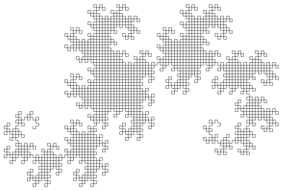


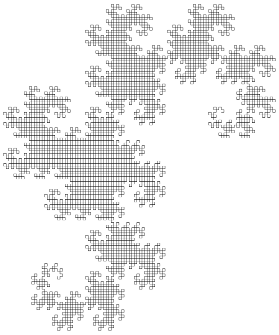


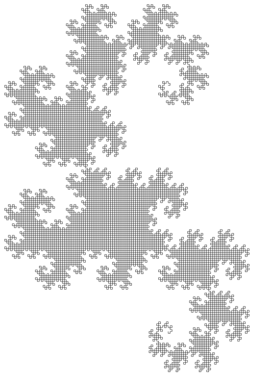


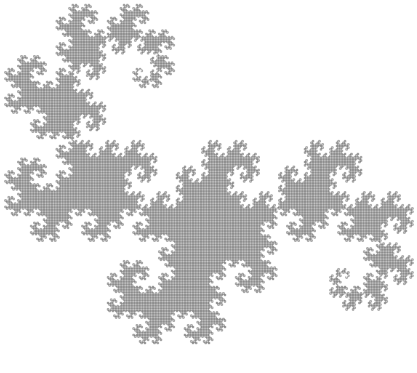












3. Language Reference Manual

3.1. Data Types & Data Structures

The language supports two categories of data types: primitive types and complex types.

3.1.1 Primitive types

The primitive types in FRAC are `int`, `double`, `string`, and `bool`.

```
int x = 3.5;
double y = x;
string s = "Stephen";
bool b = true;
```

3.1.2 Complex types

A complex type contains multiple named fields and requires a larger and variable amount of memory to store a structured collection of values. A complex type is similar to the familiar object type in so far as it contains fields, but a complex type does not contain any methods. In fact, all instantiated complex types are immutable; operating on them requires the use of functions.

Two complex types are supported in the language: `gram` and `rule`.

A `gram` represents a formal grammar that is used to specify a fractal that can be drawn. It contains an `alphabet`, an `init` string, and a set of `rules`. A `rule` represents a production rule that is a part of the formal grammar. A recursive rule consists of a rule symbol and a successor string, while a terminal rule consists of a rule symbol and a terminal function. Later sections expand on how such grammars and rules can be declared in code.

3.2. Lexical Conventions

3.2.1 Identifiers

An identifier is a sequence of alphanumerics and underscores. An identifier may begin with neither a digit nor an underscore. Both uppercase and lowercase letters are permitted. The following are valid identifiers: `kunal_43`, `hello_ANNIE`, and `do_this____justin`. The following are invalid: `helloworld&`, `_dothis`, and `4calvin`.

3.2.2 Keywords

The following are a list of reserved keywords in the language:

```
rule
gram
if
else
while
return
true
false
```

as well as the literal types:

```
int
double
bool
string
```

No keyword may be used as an identifier.

3.2.3 Literals

A literal is a notation that represents the value itself as written. Literals can only be of one of the primitive types, which are discussed below. A literal may not be used as an identifier.

Integer constants

An integer consists of a sequence of digits not containing a decimal point.

```
int x = 10;
```

Floating point constants

A floating point constant consists of two sequences of digits, where one may be the empty sequence, separated by a decimal point.

```
double y = 4.55;
```

Boolean constants

There exist only two boolean constants:

```
bool is_there = true;
bool is_there = false;
```

String constants

A string constant consists of a sequence of characters enclosed by single quotes.

```
string name = 'Anne Zhang’;
```

3.2.4 Comments

Just like in Java, `//` are used for single line comments and `/* */` for nested or multi-line comments.

```
// This is a single line comment
/* This is
   a multi-line
   comment
 */
```

In a single line, all characters after `//` are ignored by the compiler.

With multi-line comments, the compiler will ignore everything from `/*` to `*/`. Note, however, that multi-line comments cannot be nested within one another like so:

```
/* Multi-line comments
   /* cannot be nested */
   like in this example!
 */
```

This will result in a syntax error, as the compiler will treat the first `*/` as the end of the comment.

3.2.5 Operators

Operators specify logical or mathematical operations to be performed.

Arithmetic operators:
`+` addition
`-` subtraction
`*` multiplication
`/` division
`%` modulo

`=` assignment

Logical operators:
`!` negation
`==` equivalence
`!=` non-equivalence
`<` less than
`>` greater than
`&&` AND
`||` OR

The arrow `->` is a special operator used in rule definitions in grammars. In a rule, the string to the left of the arrow can be replaced by the string or system function to the right of the arrow. For example:

```
'F' -> 'F l F r r F l F',
'r' -> turn(60)
```

are both valid rules. The arrow has no meaning outside of rule definitions, and an error will be thrown if it is used outside of this context.

3.2.6 Punctuators

`;`

- terminate statements

`,`

- separate function parameters, separate key-value pairs in grammar definitions

`'`

- string literal declaration

`{}`

- grammar definitions
- scope

`()`

- function arguments
- expression precedence
- type casting
- conditional parameters

3.3 Syntax

3.3.1 Program Structure

FRAC programs should be written in a single file. A FRAC program consists of grammar definitions, function definitions, and a `main()` function. Functions and grammars are defined first and subsequently used in the `main()` function, although they cannot be defined within the `main()` function itself.

The `main()` function is the entry point for the program. It may contain variable and literal declarations, expressions, and statements. It may also use any previously defined functions and grammars. In addition, the `main()` function must use one, and only one, of the following system functions: `draw()`, `grow()`. This function specifies the type of image output that the program will create.

The following is an example of a valid FRAC program:

```
gram my_grammar = {
    alphabet: [F, r, l],
    init: 'F r r F r r F',
    rules: {
        'F' -> 'F l F r r F l F',
        'r' -> rturn(60),
        'l' -> lturn(-60),
        'F' -> move(1)
    }
}

main() {
    grow(my_function(my_grammar), 2);
}
```

In this example, the program will construct a grammar given in the declaration of my_grammar. Then, it will output a GIF showing the growth of the fractal generated by that grammar (the fractal will have undergone 2 iterations, as specified by the second parameter to the `grow()` function).

3.3.2 Expressions

Variable Declarations

Variables can be declared and assigned to a value simultaneously, or declared without assignment and assigned to a value later on. Variable can only be declared at the top of functions, before any other statements. If a variable is declared and not defined, but used later in the program, our compiler will not throw an error (and neither will `gcc`), but the variable will be evaluated to a garbage value. Declarations take the form:

```
// declaration without assignment
var_type var_name;
var_name = value;

// declaration with assignment
var_type var_name = value;
```

where `var_type` is any of the four literal type keywords (`int` , `double` , `bool` , `string`), `var_name` is any valid identifier as defined in 3.1, and `value` is either a literal of type `var_type` or an expression that evaluates to a literal of that type.

Function Definitions

Functions are declared and defined simultaneously - unlike variables, they cannot be declared without definition and defined later. All functions must return a value, although the return type is not be specified in the function declaration. Functions have any return type, except for `gram`. Any function except for the `main()` function is defined as follows:

```
my_name(params) {
    // function body
}
```

while the `main()` function must not have any formal parameters:

```
main() {
    // main function body
}
```

Additionally, the `main()` function should not contain any return statements.

Function Calls

All functions except for the `main()` function must be called explicitly, with the correct number of arguments as specified in the function definition. Functions can take arguments of any type, except type `gram`. The `main()` function is called implicitly at the start of every program run, and calling `main()` explicitly in the program will throw an error.

```
// valid function call
my_func(args);

// this will throw an error
main();
```

Function calls may be placed on the right-hand side of an assignment expression, in which case the identifier on the left-hand side will be assigned the return value of the function call.

```
// n is assigned the return value of my_func(args)
```

```
int n = my_func(args);
```

Function calls may also be nested. They can be passed as arguments into other functions, in which case the return value of the inner function call will be passed as an argument to the outer function call. The return value of the inner function call must match the argument type specified in the outer function’s definition. A type mismatch will throw an error.

```
/* my_func must return an object of type gram, otherwise this
   expression will throw an error */
draw(my_func(args), 2);
```

Grammar Definitions

Grammar definitions are similar to function definitions, but the grammars themselves are more similar to objects. Grammars are defined as follows:

```
gram my_gram {
    alphabet: [// comma-separated symbol list]
    init: // init string here,
    rules: {
      // symbol -> end string
      // OR
      // symbol -> terminal function
    }
}
```

Grammars are defined with three comma-separated fields: alphabet, init, and rules. The alphabet specifies the symbols that will be used in the rules. The init string specifies the start state of the grammar. The rules specify how the init string will be evaluated.

Every grammar must contain at least one recursive (string-to-string) rule - it wouldn’t generate a fractal otherwise! Every symbol in the alphabet must have at least one, and at most two, rules corresponding to it. Every symbol in the init string and in the rule list must be included in the alphabet. If a symbol has two rules, one rule must be recursive and the other must be non-recursive. There cannot be two recursive rules of the same name, or two terminal rules of the same name. Any other combination of rules is ambiguous and will throw an error.

Grammars are evaluated when they are passed into a drawing system function (`draw()` or `grow()`). Grammar evaluations start with the `init` string, which is then evaluated recursively for the number of times specified in the second argument to the drawing function call. For every recursive evaluation, the compiler will look for a recursive rule for each symbol, and will only use a terminal rule for a symbol if it cannot find a recursive rule, or if it has reached the end of its required iterations.

Arithmetic Expressions

Arithmetic expressions are expressions that contain an arithmetic operator, and evaluate to a literal value. They can be placed on the right-hand side of variable assignments, or passed as arguments to function calls.

```
int x = 3;
int y = 8;
int z = x + y; // z = 11
my_func(x + y); // 11 is passed into my_func
```

Boolean Expressions

Boolean expressions are expressions that contain logical operators, and evaluate to a boolean value `true` or `false`. They are used to evaluate conditional and loop statements.

```
bool isTrue = true;
bool isFalse = false;
if(isTrue || isFalse) {
    print(“truth”);
}
```

3.3.3 Statements

A statement is a complete instruction that can be interpreted by the computer. Statements are executed sequentially within a function.

Expression Statements

Expression statements are the most common type of statement, and can include any of the previously covered expressions. In FRAC, all statements are terminated with a semicolon `;`.

Conditional Statements

Conditional statements first check the truth condition of a boolean expression, and then execute a set of statements depending on the result. Here is an example `if` / `else` conditional statement:

```
if (expression) {
    statement
}
else if (expression) {
    statement
}
else {
    statement
}
```

Only the `if` clause of the conditional statement is required. The `else` statement is executed only if none of the previous conditions return true.

Loop Statements

Loop statements are constructed using the `while` and `for` keywords, which allow you to iterate over blocks of code.

```
while (expression) {
    statement
    ...
}
for(int i = 0; i < 5, i=i+1) {
    statement
    ...
}
```

In the case of `while` loops, the truth condition of the boolean expression is checked before every execution of the body of the `while` loop, which is executed only if expression returns true.

In the case of `for` loops, there are three expressions within the parentheses, separated by semicolons. However, only the middle expression is required, and it must be an expression that evaluates to a boolean value, which is used to check if the `for` loop should continue running or if it should terminate.

Return Statements

Ends the execution of a function with the use of the keyword `return`. If a function does not have a `return` statement at the end, it is assumed to be a void function without a return type.

3.4. Special Functions

3.4.1 Terminal Functions

There are three possible terminal functions that are used in grammar declarations in FRAC: `move()`, `rturn` and `lturn`. There should be at least one rule in your grammar that evaluates to a terminal function, in order to generate a fractal image. These functions correspond to the turtle graphics "pen", which draws the image that you are generating.

`move()`

This is one of two possible terminals in a FRAC grammar:

```
move(int distance)
```

The function draws a line of length `distance`.

`turn()`

The other two possible terminals in a FRAC grammar:

```
rturn(int angle)
```

or

```
lturn(int angle)
```

The function indicates to the grammar that the current line being drawn should be re-oriented by `angle` degrees, which can be in the positive or negative direction (abiding by the right hand rule).

3.4.2 System Functions

`draw()`

This is one of two functions in a FRAC program that generates a fractal image:

```
draw(gram g, int n)
```

The function creates a static BMP image of the fractal described by the grammar `g` over `n` number of iterations.

`grow()`

```
grow(gram g, int n)
```

The function resembles `draw()`, except instead of creating a static image, it creates a dynamically “growing” GIF (by linking together a collection of static BMP images) of the fractal described by the given grammar `g` over `n` iterations.

`print()`

```
print(string s)
```

The function prints out the string `s` to the standard output. The same escape sequences as Java would be interpreted correspondingly (i.e. `\n` for newline).

4. Project Plan

4.1 Process & Timeline

We decided that we wanted to create a fractal-generating language in late September. We began to design the language in early October, and continued to flesh out ideas until the beginning of November. Programming of the compiler began in earnest in November. We completed a basic front-end and basic code generator in mid-November. We worked on building the semantic checker and code generator, as well as expanding the front-end, well into December. Finally, we linked all the parts together, obtaining our first successful fractal images in mid-December.

While building our language and the compiler, members of our group tried to work together as much as possible. Pair programming was a crucial part of our strategy - we recognized that this allowed us to catch errors much more easily, as well as write clearer and more readable code.

4.2 Team Roles

- **Anne Zhang**: Manager / Language Guru / System Architect
- **Kunal Kamath**: Language Guru / System Architect / Tester
- **Calvin Li**: System Architect

Throughout the semester members of our group stepped up to take on various tasks, so no member had a static role. Anne was the primary author of the front-end (scanner, parser, AST). She and Kunal worked together on the semantic checker and the code generator, the core parts of our compiler. Kunal worked on building out a robust test suite for the language, as well as using libraries to generate dynamic GIFs. Calvin worked on using the turtle graphics library to write the C code into which FRAC is generated.

The following is an extensive log of our `git commits`, which show the work that each team member put into each part of the project.

```
commit a97091a7f3bda4e37d98781e80fbf3496df3df60
Author: Annie Zhang
Date:   Tue Dec 22 15:09:05 2015 -0500

    Finished final report

commit e1a0876bb684184b19b430c3d66d57d6325896f7
Author: Annie Zhang
Date:   Tue Dec 22 04:48:18 2015 -0500

    Added to final report
```

```
commit 2eb906746e9b6fdb53072f8494851b526a1e1e17
Author: Annie Zhang
Date:   Tue Dec 22 00:50:27 2015 -0500

    Started color stuff

commit 4c25b61803afa2698a49e4c23213a6998a80e888
Merge: 8835df8 045aff8
Author: Annie Zhang
Date:   Mon Dec 21 16:40:28 2015 -0500

    Merge branch 'master' of github.com:kunalkamath/FRAC

commit 8835df821cbc0b497c4786923d21af7b3e4bd220
Author: Annie Zhang
Date:   Mon Dec 21 16:40:18 2015 -0500

    Merging

commit 045aff8898640480026dfa770dcad9ef1fcca0a7
Author: Kunal Kamath
Date:   Mon Dec 21 16:36:05 2015 -0500

    Added config files

commit 766e4e781f078feeb3392a052ce8279e65189ee1
Author: Kunal Kamath
Date:   Mon Dec 21 06:58:32 2015 -0500

    Made sutre test suite works for demo

commit ecf51d9eb59ecceca759296de1aa7a13c0fd99c1
Author: Kunal Kamath
Date:   Mon Dec 21 06:36:29 2015 -0500

    Now really ready for demo

commit 156fcf98f365b9a5d79511278ac62f7705cc50db
Author: Kunal Kamath
Date:   Mon Dec 21 06:14:51 2015 -0500

    Ready for demo

commit 622f1206fc3b653a1fcf46a86e06fef5980b2dc9
Author: Kunal Kamath
```

```
Date:   Mon Dec 21 03:10:14 2015 -0500

    Finished tests

commit 898319875067f90c2431e6dfde11fa13af4cb517
Merge: f9ce528 5c3e367
Author: Kunal Kamath
Date:   Mon Dec 21 02:31:35 2015 -0500

    Merge branch 'master' of https://github.com/kunalkamath/FRAC

commit f9ce5283724e31495789c6508542b0cd0309a2d4
Author: Kunal Kamath
Date:   Mon Dec 21 02:31:30 2015 -0500

    More tests still

commit 5c3e36760bcfe0ebac20c0e34a70fad56b022fc7
Author: Annie Zhang
Date:   Mon Dec 21 02:29:31 2015 -0500

    Fixed another small error

commit 034e6bb7b7ebdcb0e9abc8ea2a79460b36b54745
Merge: d429566 3c1a1bc
Author: Annie Zhang
Date:   Mon Dec 21 02:28:47 2015 -0500

    Merge branch 'master' of github.com:kunalkamath/FRAC

commit d429566d21be5452a77cb80b5f1e673081b2eaa6
Author: Annie Zhang
Date:   Mon Dec 21 02:28:39 2015 -0500

    Fixed small error

commit cbe6ddaf4c47b4cf9fb1b61f0e3a0f9f30ca4e5e
Author: Annie Zhang
Date:   Mon Dec 21 02:07:46 2015 -0500

    IM SO SORRY

commit 3c1a1bcbe0d079480349d6c5585187f32c00d164
Merge: 611dc4f 607aaa9
Author: Kunal Kamath
Date:   Mon Dec 21 01:39:15 2015 -0500
```

```
    Merge branch 'master' of https://github.com/kunalkamath/FRAC

commit 611dc4fdf8f9254ff416a0a83be3e214e2eab27a
Author: Kunal Kamath
Date:   Mon Dec 21 01:39:06 2015 -0500

    Added more failure tests

commit 607aaa9674b2b897219b666f86de76042567d0a1
Author: Annie Zhang
Date:   Mon Dec 21 01:03:37 2015 -0500

    Void print checking

commit 3fb6901950d83278a8045d499f9ff1308b431ae6
Author: Kunal Kamath
Date:   Mon Dec 21 00:19:03 2015 -0500

    Failure tests

commit b9c6cfdfc7a5a6b5bfc6ff15249ca8fa90ebbd2f
Merge: a7ffa6b 8c5ea0f
Author: Annie Zhang
Date:   Sun Dec 20 23:47:04 2015 -0500

    Merge branch 'master' of github.com:kunalkamath/FRAC

commit a7ffa6b6e87fee56168b3a15728c679953997148
Author: Annie Zhang
Date:   Sun Dec 20 23:46:11 2015 -0500

    Parser warnings fixed

commit 8c5ea0f6de9e83e8c186bc428d92fb9239f66249
Author: Kunal Kamath
Date:   Sun Dec 20 23:26:57 2015 -0500

    Did some restructuring

commit 8651d1e5c2944d9ed79f496bfce09eef006be392
Merge: e18a9f4 229815c
Author: Kunal Kamath
Date:   Sun Dec 20 23:11:31 2015 -0500

    Merge branch 'master' of https://github.com/kunalkamath/FRAC
```

```
commit e18a9f412583fc47c63497a5066e1f34e1f148d7
Author: Kunal Kamath
Date:   Sun Dec 20 23:11:24 2015 -0500

    More testing

commit 229815c50da7bfe6239fc8d2e12c4f17cb256c7b
Merge: f72a5f3 bacd55b
Author: Annie Zhang
Date:   Sun Dec 20 23:09:37 2015 -0500

    Merge branch 'master' of github.com:kunalkamath/FRAC

commit f72a5f39d9f1f685a20da3e92070726d75f2b87f
Author: Annie Zhang
Date:   Sun Dec 20 23:09:22 2015 -0500

    Updated vdecl checking

commit bacd55be230c732bf54644e593a157b2cd0dbb5e
Merge: 9d617b9 4b062b6
Author: Kunal Kamath
Date:   Sun Dec 20 23:01:50 2015 -0500

    Merged

commit 9d617b9aadb3411a5255ddd05bc1f527997fa205
Author: Kunal Kamath
Date:   Sun Dec 20 23:00:45 2015 -0500

    Still testing

commit 4b062b617597bbb6b41d950fb5c515fbc7aeb2e6
Author: Annie Zhang
Date:   Sun Dec 20 22:58:47 2015 -0500

    Checks duplicate variables

commit 3d06b8dcf1a49c98c04022ddbae44e425ab814aa
Author: Annie Zhang
Date:   Sun Dec 20 22:32:09 2015 -0500

    All warnings fixed, parser rules still not reduced tho

commit ff1d32894e48b36a21097e985a1ab85a35109fc0
```

```
Merge: 124cbf4 fdebde1
Author: Annie Zhang
Date:   Sun Dec 20 22:26:47 2015 -0500

    Fixed conflicts

commit 124cbf4ebc1fc278cfa505f5fbe5a0968a81403b
Merge: d06878e 306b315
Author: Annie Zhang
Date:   Sun Dec 20 22:15:52 2015 -0500

    Merge branch 'cleanup'

commit 306b3153e0132007de774991f709f592d196f2e4
Author: Annie Zhang
Date:   Sun Dec 20 22:14:18 2015 -0500

    5 warnings left woo

commit fdebde1cf0d5524f41bfed8b69591be5176e7325
Author: Kunal Kamath
Date:   Sun Dec 20 22:11:05 2015 -0500

    Adding tests

commit 56f302f79fffb4e5ba002204edf21aea8aa295c3
Author: Annie Zhang
Date:   Sun Dec 20 22:10:54 2015 -0500

    5 warnings left

commit f5a62eb8a2eb078f9ff36763064a02e3129bb1b9
Author: Annie Zhang
Date:   Sun Dec 20 21:04:12 2015 -0500

    Working on final report

commit c15c2e29e7fcb565fd6a86621ebf267a200ea364
Author: Kunal Kamath
Date:   Sun Dec 20 20:41:33 2015 -0500

    Grow() working, consider adding step sizes

commit 46afb00540a3f236dbedbfd0fc482c926288b9a1
Author: Kunal Kamath
Date:   Sun Dec 20 18:55:11 2015 -0500
```

```
    Grams fully integrated, basics of Grow() working too

commit ed5069b3d2c68fbba1dad4ca67949eac8454d1ce
Author: Kunal Kamath
Date:   Sun Dec 20 15:24:36 2015 -0500

    Documentation added

commit e822dfd459f25cdc77e6886384086603283fd36e
Merge: d06878e e3384e8
Author: Annie Zhang
Date:   Sun Dec 20 13:53:45 2015 -0500

    Merge pull request #17 from kunalkamath/gram-gen

    Gram gen

commit e3384e81d88680cedb8bff83db6ae0389333464b
Author: Annie Zhang
Date:   Sun Dec 20 13:52:43 2015 -0500

    It works

commit 8f425333e5b31bb7322533eb328667fccb0b25fb
Author: Annie Zhang
Date:   Sat Dec 19 19:19:18 2015 -0500

    Code gen working for draw() function

commit 95e51570d5e1f8cfb891a2ce03a2b39bbff109d9
Author: Annie Zhang
Date:   Sat Dec 19 18:47:15 2015 -0500

    Basic gram code gen WORKING

commit d06878e0126851d0e7f6f7332103ff6629a00021
Merge: 34b123c 5cd899b
Author: Kunal Kamath
Date:   Sat Dec 19 14:54:18 2015 -0500

    Merged gram semantics

commit 34b123c1addc904f95e45af254fc7f54da38fc99
Author: Kunal Kamath
Date:   Sat Dec 19 14:51:00 2015 -0500
```

```
    Refactored var_decls

commit 5cd899bf3ae3c85ba4a6c392871dffb742be7687
Merge: 82d0ff5 c68052e
Author: Annie Zhang
Date:   Sat Dec 19 14:16:13 2015 -0500

    Merging gram

commit c68052eecf2f0036180e0c718bc3e7adcf94844f
Author: Annie Zhang
Date:   Sat Dec 19 14:06:57 2015 -0500

    Draw function semantic checking

commit c5871604398a61dc39be5f6868d9be27c528e003
Author: Annie Zhang
Date:   Sat Dec 19 13:33:19 2015 -0500

    Checks that every element of alphabet has a corresponding rule

commit 85c46e284b059aeb438090442dab8600c48395f3
Author: Annie Zhang
Date:   Sat Dec 19 11:19:58 2015 -0500

    Gram semantics work basically

commit 3cfb38d0a5170f268f0c4a5d2881bf27e1ee8aab
Author: Annie Zhang
Date:   Sat Dec 19 11:11:53 2015 -0500

    BASIC GRAM SEMANTICS WORK

commit e60e85c70adf252c8ff9ec855840f4af78efe9d0
Author: Annie Zhang
Date:   Sat Dec 19 10:52:18 2015 -0500

    Alphabet and init checking

commit 82d0ff5caf11da61168fd08b2088969108bea9d6
Author: Kunal Kamath
Date:   Sat Dec 19 04:52:16 2015 -0500

    More tests, assignment equality working
```

```
commit 4e5aa8e7d38b4c0a25e6cceb30e3758b71408731
Author: Kunal Kamath
Date:   Sat Dec 19 02:45:36 2015 -0500

    Added tests, working on var_decls

commit b92e7183ac0a346b77fad53f2cb3ab0aeeea1d71
Author: Annie Zhang
Date:   Fri Dec 18 15:19:10 2015 -0500

    Slight code cleanup

commit dfec4202da4448b5f11865517508aaad95b3ce06
Author: Annie Zhang
Date:   Fri Dec 18 15:12:35 2015 -0500

    Working on semantic checking for grams

commit 8607cc00ce8a72969b8b3bef56928209e60a7230
Author: Annie Zhang
Date:   Fri Dec 18 14:04:30 2015 -0500

    Parsing and scanning works for grams

commit 997f3e89f6c403f2ddaa32b1e0737d8ad3e488c3
Author: Annie Zhang
Date:   Fri Dec 18 10:48:15 2015 -0500

    Saving changes before checking out

commit 6c53df94230e18de1f025d2c03223a644538dcb0
Author: Annie Zhang
Date:   Fri Dec 18 01:31:39 2015 -0500

    Slowly but surely, semantic for grammars

commit c42a20e50e1d7454865376d0deb6c0d9211bea00
Author: Annie Zhang
Date:   Thu Dec 17 20:23:49 2015 -0500

    Added grams and rules to scanner, parser, and AST

commit 51a80145bafd857301119adb9bdc4b7231a279bf
Author: Calvin Li
Date:   Thu Dec 17 18:05:27 2015 -0500
```

```
    improved c graphics code, adding comments to
    indicate where a FRAC gram maps in

commit 7ef9fa41ac82ee87828238741b9571faf0b6b215
Merge: 03d0def 23e0298
Author: Annie Zhang
Date:   Thu Dec 17 15:16:17 2015 -0500

    Fixed actual parameters scoping error

commit 23e0298394e6ef93660e1104c62072d6bcf65da7
Author: Kunal Kamath
Date:   Thu Dec 17 02:32:19 2015 -0500

    Code gen flow complete

commit 03d0def9fe62a7ae1098b73b5aae3b4b5a89dfce
Author: Annie Zhang
Date:   Wed Dec 16 23:23:24 2015 -0500

    Added return types to function code gen

commit 24390afcdffed8882f12965bc87c8340c26579b3
Author: Annie Zhang
Date:   Wed Dec 16 19:42:10 2015 -0500

    Main function included in checked_fdecls

commit 8117ee11c7452a0b486da3c2ef372f34a7a552f7
Merge: 84d7eca b0fafc9
Author: Kunal Kamath
Date:   Wed Dec 16 19:25:46 2015 -0500

    Merge branch 'master' of https://github.com/kunalkamath/FRAC

commit 84d7eca576ab70219da21a4e5fd0df2e28a838af
Author: Kunal Kamath
Date:   Wed Dec 16 19:25:08 2015 -0500

    Codegen flow almost working

commit b0fafc9afded36e97aa423824a863aaf055fe2b5
Merge: 78abde8 834bd60
Author: Calvin Li
Date:   Wed Dec 16 16:51:16 2015 -0500
```

```
    Merge branch 'master' of https://github.com/kunalkamath/FRAC

commit 78abde8922e4f225a1217c30e9f246f3c2723ae6
Author: Calvin Li
Date:   Wed Dec 16 16:50:50 2015 -0500

    added c graphics stuff

commit 834bd6001302b71f5e0aa8fa347cc0b2ae27f542
Merge: 806c27b fc12231
Author: Kunal Kamath
Date:   Wed Dec 16 05:26:56 2015 -0500

    Merged with semantic branch

commit fc122315a8de19c22018c3c825e83b04a915c4b4
Merge: 70079e1 586928f
Author: Kunal Kamath
Date:   Wed Dec 16 05:21:30 2015 -0500

    Resolving conflicts in semantic branch

commit 806c27b7a838ea0a47be40b614ff8185c48190aa
Author: Kunal Kamath
Date:   Wed Dec 16 05:16:37 2015 -0500

    Semantics mostly in place, now working on code generation

commit 586928f81a33dd6ba336ee96b5b2ac85585b6452
Author: Annie Zhang
Date:   Wed Dec 16 01:13:51 2015 -0500

    Print function call checked

commit 8dd000f6a36e80934ea95fe2a173ad5e88b733e2
Author: Kunal Kamath
Date:   Wed Dec 16 01:02:31 2015 -0500

    For loops implemented

commit c80bfed07f9da170c30472a5e721fd936d151c87
Author: Annie Zhang
Date:   Wed Dec 16 00:39:28 2015 -0500

    Function call checking works
```

```
commit e7aa3fe2d99b85f3cc4533df8e7f469de28b5a11
Author: Kunal Kamath
Date:   Wed Dec 16 00:32:42 2015 -0500

    If, while, and more operators added

commit 02024da48f5ce71a47a29fa3c3d4779bc080a7e5
Author: Kunal Kamath
Date:   Tue Dec 15 23:07:58 2015 -0500

    Fixed small merge error

commit 70079e117a08019bb081b8b0accf8a4318ada445
Merge: a546a09 aad94cd
Author: Kunal Kamath
Date:   Tue Dec 15 23:04:39 2015 -0500

    Merged Annie's semantic work

commit a546a098db311d1674a7cca8603020d10c4127e1
Merge: e90cfe9 fb3a301
Author: Kunal Kamath
Date:   Tue Dec 15 22:53:08 2015 -0500

    Merge branch 'semantic_kunal'

commit fb3a301b74c3f945c6e5f6a29280cedb7bac8d79
Merge: 57f68b8 e90cfe9
Author: Kunal Kamath
Date:   Tue Dec 15 22:52:56 2015 -0500

    Merging semantic with master

commit aad94cd6ff491d72a0434e595ec8f992408230c3
Author: Annie Zhang
Date:   Tue Dec 15 22:43:23 2015 -0500

    Updated vdecl

commit f3ce8ad9e6c6c79c4f476ebeb7053014535afc50
Author: Annie Zhang
Date:   Tue Dec 15 22:26:57 2015 -0500

    Scope kind of working

commit 57f68b836d66c57fb606933f65879ff80b4cbaff
```

```
Author: Kunal Kamath
Date:   Tue Dec 15 21:31:23 2015 -0500

    Variable environment shit is impossible

commit 6c57c6f57598f5ef460416799e81cd374e8f89ab
Author: Kunal Kamath
Date:   Tue Dec 15 17:59:30 2015 -0500

    Problems with vdecl stuff

commit 622c148f7ad8a2f367dd9c5c9372f9091aa4d0c2
Author: Kunal Kamath
Date:   Tue Dec 15 15:13:23 2015 -0500

    More semantic progress

commit bc91535c5009650d1e58f2e4c4175d1ad8033e4d
Author: Annie Zhang
Date:   Tue Dec 15 15:05:36 2015 -0500

    Return type checking

commit d8707bfcf1a404e11562b63e71d4c1b1ec2c2c7a
Author: Annie Zhang
Date:   Tue Dec 15 14:13:41 2015 -0500

    A commit

commit 08e212b94c97c2c11fbc9ca8f0f54ce3bf3cc5b4
Author: Annie Zhang
Date:   Sat Dec 5 16:58:29 2015 -0500

    func decl checking works woo

commit f192a9cedfb4734c0d203e440e46129d45165e07
Author: Annie Zhang
Date:   Sat Dec 5 16:57:20 2015 -0500

    func decl checking works woo

commit 3f2dbecffe71e594507a8a0e6aa028bf7198b288
Author: Kunal Kamath
Date:   Sat Dec 5 16:55:36 2015 -0500

    Assignment checking in progress
```

```
commit 1f53b1b873c7360b0dc850132b35acac30e633eb
Author: Kunal Kamath
Date:   Mon Nov 30 17:09:42 2015 -0500

    Binop semantic analysis underway

commit 252b8c797f32c3f74e92cc589606d627453f816a
Author: Annie Zhang
Date:   Sun Nov 29 19:34:27 2015 -0500

    Updated repeat function testing

commit c56c95035860df124d5363c2f6b6f976493e0a15
Author: Annie Zhang
Date:   Sat Nov 28 15:06:21 2015 -0500

    Mutually recursive functions not compiling for some reason

commit e9d07d2e9d891cfa164ca482a9b63c1e3ae51611
Author: Annie Zhang
Date:   Sat Nov 28 14:45:44 2015 -0500

    Starting function body/statement checking

commit b5badc9f2fa94dfe162ad8176799abf0735676c2
Author: Annie Zhang
Date:   Sat Nov 28 13:46:26 2015 -0500

    Function declaration checking WORKS

commit a8bffddb8731db83969c712bfb820742fc53fbf4
Author: Annie Zhang
Date:   Fri Nov 27 23:53:14 2015 -0500

    Function declaration checking, failing tests tho

commit 832d228fa32044c7a9c64a53ae66decae01156b1
Author: Annie Zhang
Date:   Tue Nov 24 13:43:02 2015 -0500

    Updated Makefile to include semantic stuff

commit 8384235ac4b58c4f1ef5e030ff271d00be235ea7
Author: Annie Zhang
Date:   Tue Nov 24 02:01:11 2015 -0500
```

```
    Started semantic checking

commit e90cfe936f6f09705ac9c8deb4ffb551d0602685
Author: Kunal Kamath
Date:   Tue Nov 17 16:48:11 2015 -0500

    Fixed escape sequence error in strings

commit 9fc71731ade6993c4d721e0d58ecceffd8554347
Author: Annie Zhang
Date:   Tue Nov 17 13:38:52 2015 -0500

    Working demo

commit 321c6e6ad3702dc94e2e1b53cc44e009f75de247
Author: Kunal Kamath
Date:   Tue Nov 17 00:43:57 2015 -0500

    Finishing touches before hello_world deliverable

commit 25630b78d5cc4d29a9db884faddd76392118db7b
Merge: dff3882 8c66cb0
Author: kunalkamath
Date:   Tue Nov 17 00:37:40 2015 -0500

    Merge pull request #2 from kunalkamath/symbol-table

    Symbol table merged

commit 8c66cb084b71062e911ff1e53e45fc0ce19fcf4e
Author: Annie Zhang
Date:   Tue Nov 17 00:35:59 2015 -0500

    Non-main function declarations working

commit dff3882d4c8095c4eb476e05788c97d24b608987
Author: Kunal Kamath
Date:   Tue Nov 17 00:22:53 2015 -0500

    Merged testing branch and cleaned up Makefile

commit 4eeabb8d739e140ab3c966371b06183170baa0b1
Merge: c2337c7 7a589e5
Author: kunalkamath
Date:   Tue Nov 17 00:18:48 2015 -0500
```

```
    Merge pull request #1 from kunalkamath/testing

    Testing

commit bf298bb78df7be6a5d07629959285c10ab83e691
Author: Annie Zhang
Date:   Tue Nov 17 00:11:50 2015 -0500

    Function calls working

commit 8ed2368c64ab90316e839e6fe6fe1de60fbc5bd3
Author: Annie Zhang
Date:   Tue Nov 17 00:01:39 2015 -0500

    Compiles most expressions and statements

commit 7a589e588c20d829dba0cd6327c84a9ffbbc824b
Author: Calvin Li
Date:   Mon Nov 16 22:32:50 2015 -0500

    took out redundant copying of compiled file through mktemp

commit 418c845dfa50754dbf4eee3f6814f4ad8f63d9c2
Author: Kunal Kamath
Date:   Mon Nov 16 19:44:45 2015 -0500

    Testing fully functional

commit e9186dc93ad49f590712dad1c465ccfc8ea9df2a
Author: Kunal Kamath
Date:   Mon Nov 16 01:04:50 2015 -0500

    Testing script in early stages

commit 9e48bf2b8b25ced92d7131eb7301fddbc733b743
Author: Kunal Kamath
Date:   Sun Nov 15 21:14:54 2015 -0500

    Hello world workingclear

commit 6a5d6cfd32baafeaf924011084bbc5df69596265
Merge: aac8615 dd5d636
Author: Kunal Kamath
Date:   Sun Nov 15 18:49:01 2015 -0500
```

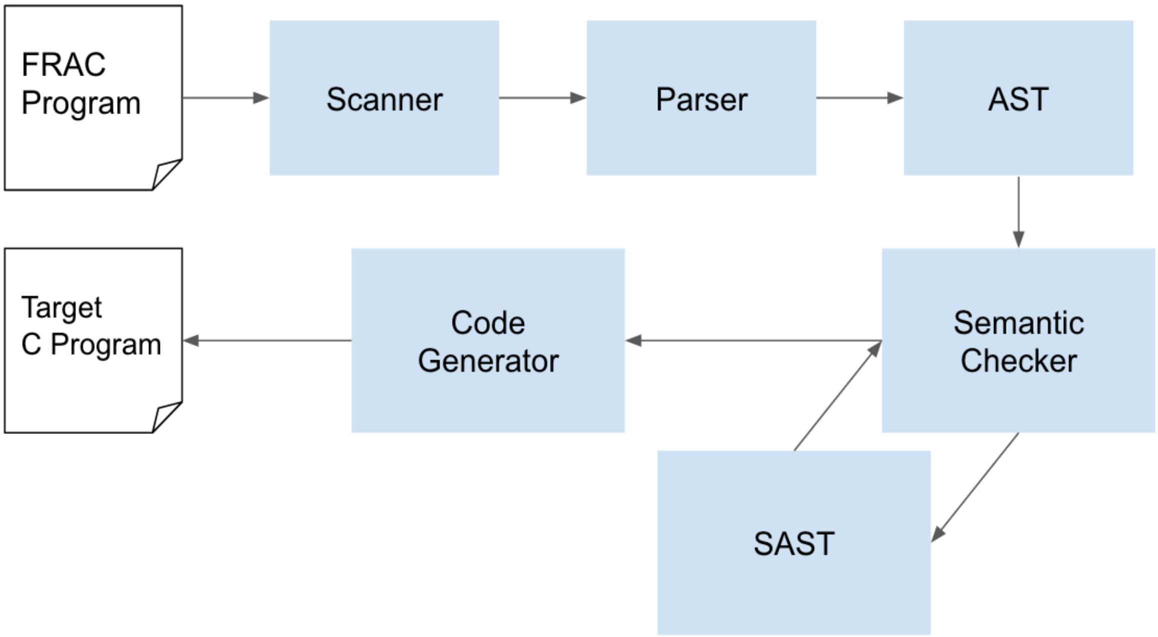
```
    Merge branch 'hello-world' of https://github.com/kunalkamath/FRAC into hello-wo

commit dd5d63621e3aea390d4e1fca46bb75f6d9651b1f
Author: Annie Zhang
Date:   Sun Nov 15 18:48:38 2015 -0500

    Compiling but not working

commit aac8615abd002d157b81b4fa1e5d3f3aa977cd3c
Merge: 339b1b1 140936e
Author: Kunal Kamath
Date:   Sun Nov 15 17:16:07 2015 -0500

    Merge branch 'hello-world' of https://github.com/kunalkamath/FRAC into hello-wo

commit 140936ef5090731f2d613024138c87460b7baae1
Author: Annie Zhang
Date:   Sun Nov 15 04:10:30 2015 -0500

    Basic untested code generator

commit 0dea5b880828aad40de28b0193bcaa33ccac4447
Author: Annie Zhang
Date:   Sun Nov 15 02:42:13 2015 -0500

    A lotta shit

commit 339b1b15a7a7ca8d155c03592f3e77a8b96da994
Merge: c2337c7 b44adb0
Author: Kunal Kamath
Date:   Sat Nov 14 11:17:11 2015 -0500

    Merge branch 'hello-world' of https://github.com/kunalkamath/FRAC into hello-wo

commit b44adb089003e740535f8e9c993706cfa80537ff
Author: Annie Zhang
Date:   Sat Nov 14 11:12:22 2015 -0500

    Modified AST

commit 57988bccbb0017fd20a3ac6c901fa1f49b589ae6
Author: Annie Zhang
Date:   Fri Nov 13 14:07:21 2015 -0500

    Initial commit
```

```
commit c2337c73d61ef82f704497e7bcd554f1d57399c2
Author: Kunal Kamath
Date:   Thu Nov 5 20:58:38 2015 -0500

    Added microc files with hello-world functionality

commit 54d34e8665f091149b72742da271eb3ef1a2a1f3
Author: Kunal Kamath
Date:   Tue Nov 3 16:41:22 2015 -0500

    Added to AST

commit cc66415d5878e2e5eda9cac2dca0eb085b99ed69
Author: Annie Zhang
Date:   Mon Nov 2 19:13:58 2015 -0500

    Finished scanner

commit 220e425d54a3cd24fb5cac5e3ca69eaf947b44bc
Author: Kunal Kamath
Date:   Mon Nov 2 18:06:43 2015 -0500

    Parser tokens added

commit 457636453bb594ddc05b4d8532753d3a412b3bfa
Author: Calvin Li
Date:   Mon Nov 2 15:00:12 2015 -0500

    fixed ' characters

commit 1fe7f7a51b8d29bac3ea75b31558d4c77a6032d9
Author: Calvin Li
Date:   Sun Nov 1 16:02:39 2015 -0500

    copied in scanner and parser for MicroC

commit 5332f24964f5b77a677b15736353a124e343b6da
Author: kunalkamath
Date:   Tue Oct 6 16:35:19 2015 -0400

    Initial commit
```

4.3 Development Tools

We used `OCaml` to write the entirety of our compiler, specifically using `ocamllex` and `ocamllyacc` for the front-end, and regular `OCaml` for the semantic checker and the code generator. Our team used the Bash shell to run testing scripts, as well as Sublime Text and Atom as text editors for writing code. Finally, we used Github extensively for version control throughout the building of our project.

5. Architectural Design



5.1. Compiler Structure

The scanner `scanner.mll` parses a FRAC program into a list of recognizable tokens. The parser `parser.mly` makes sure that there are no syntax errors, and uses this list of tokens to generate an abstract syntax tree (AST). Then, the semantic checker `semantic.ml` walks through the AST, making sure that there are no semantic errors, and generates an SAST. Finally, the code generator `compile.ml` walks through the SAST and generates the C target code.

5.2. Turtle Graphics in C

Our compiled C code uses a turtle graphics library to generate fractal images. The library that we use, which can be found in our source code, uses simple commands like `turtle_forward()`, `turtle_turn_right()`, and `turtle_turn_left()` to generate graphics. The terminal functions used in FRAC grammar declarations map directly to these functions.

When compiled into C, each grammar declaration in a FRAC program is transformed into two functions. The first function, `[gram_name]()`, represents the rules of the grammar. The second function, `[gram_name]_start()`, represents the init string of the grammar. Each rule symbol becomes a call to the `[gram_name]()` function.

If any `draw()` or `grow()` functions are called in the main function, the compiler generates the C code necessary for creating, saving, and cleaning up and image. The following is an example of the C program generated from the Koch snowflake FRAC program included in Section 2.2.3.

```
#include "turtle.h"
#include
#include

void koch(char var, int iter) {
    if (iter < 0) {
        if (var == 'F') {
            turtle_forward(1);
        }
    } else {
        if(var == 'F') {
            koch('F', iter - 1);
            koch('m', iter - 1);
            koch('F', iter - 1);
            koch('p', iter - 1);
            koch('p', iter - 1);
            koch('F', iter - 1);
            koch('m', iter - 1);
            koch('F', iter - 1);
        }
        if (var == 'm') {
            turtle_turn_right(60);
        }
        if (var == 'p') {
            turtle_turn_left(60);
        }
    }
}
```

```
void koch_start(int iter) {
    koch('F', iter);
    koch('p', iter);
    koch('p', iter);
    koch('F', iter);
    koch('p', iter);
    koch('p', iter);
    koch('F', iter);
}

int main(){
    turtle_init(2000, 2000);
    koch_start(6);
    turtle_save_bmp("koch.bmp");
    turtle_cleanup();
    return 0;
}
```

5.3. GIF Generation

Our compiler uses two libraries, `GraphicsMagick` and `gifsicle`, which can be found in our source code directory. These are used to create animated GIFs when a FRAC program uses the `grow()` system function. When the `run.sh` shell script is run on a FRAC program that uses `grow()`, a series of BMP images showing the growth of the fractal is generated. Then, we use the `GraphicsMagick` library to link those images together into a single GIF image, and the `gifsicle` library to animate that GIF.

6. Testing

We primarily conducted full stack integration tests during the development of our compiler, with a focus on testing semantic checking and C-code generating. We stuck to writing a new test as we implemented a new feature of the language, ensuring that the new feature would work and compile as intended in our LRM. Our testing boiled down to 3 main areas of focus: correctly catching errors in the semantic checking, testing generated syntax based on our LRM, and testing cases of ambiguity in frac programs. Towards the end, we tried to cover all of our bases by testing as many features as written in our LRM. We also separated out our tests into two folders: a folder of frac programs that should compile and run as intended (pass) and a folder of frac programs that we intentionally wrote to throw a compile error (fail).

The following are examples of some of the tests in our test suite. The rest can be found in our source code directory.

A passing test:

`test-fdecl_return.frac`

```
foo(string x, bool b) {
    print(x);
    return b;
}

bar(double d) {
  return d * 2.0;
}

main() {
  print(bar(11.11));
  if(foo("hello",100000 > -1) == true) {
    print("sweet!");
  }
}
```

Expected output:

`test-fdecl_return.txt`

```
foo(string x, bool b) {
    print(x);
    return b;
}

bar(double d) {
  return d * 2.0;
}

main() {
  print(bar(11.11));
  if(foo("hello",100000 > -1) == true) {
    print("sweet!");
  }
}
```

And a failing test:

`gram_no_alph.frac`

```
gram koch = {
    alphabet: [],
    init: 'F p p F p p F',
    rules: {
        'F' -> 'F m F p p F m F',
        'F' -> move(1),
        'm' -> rturn(60),
        'p' -> lturn(60)
    }
}

main() {
    draw(koch, 6);
}
```

Expected output:

`gram_no_alph.txt`

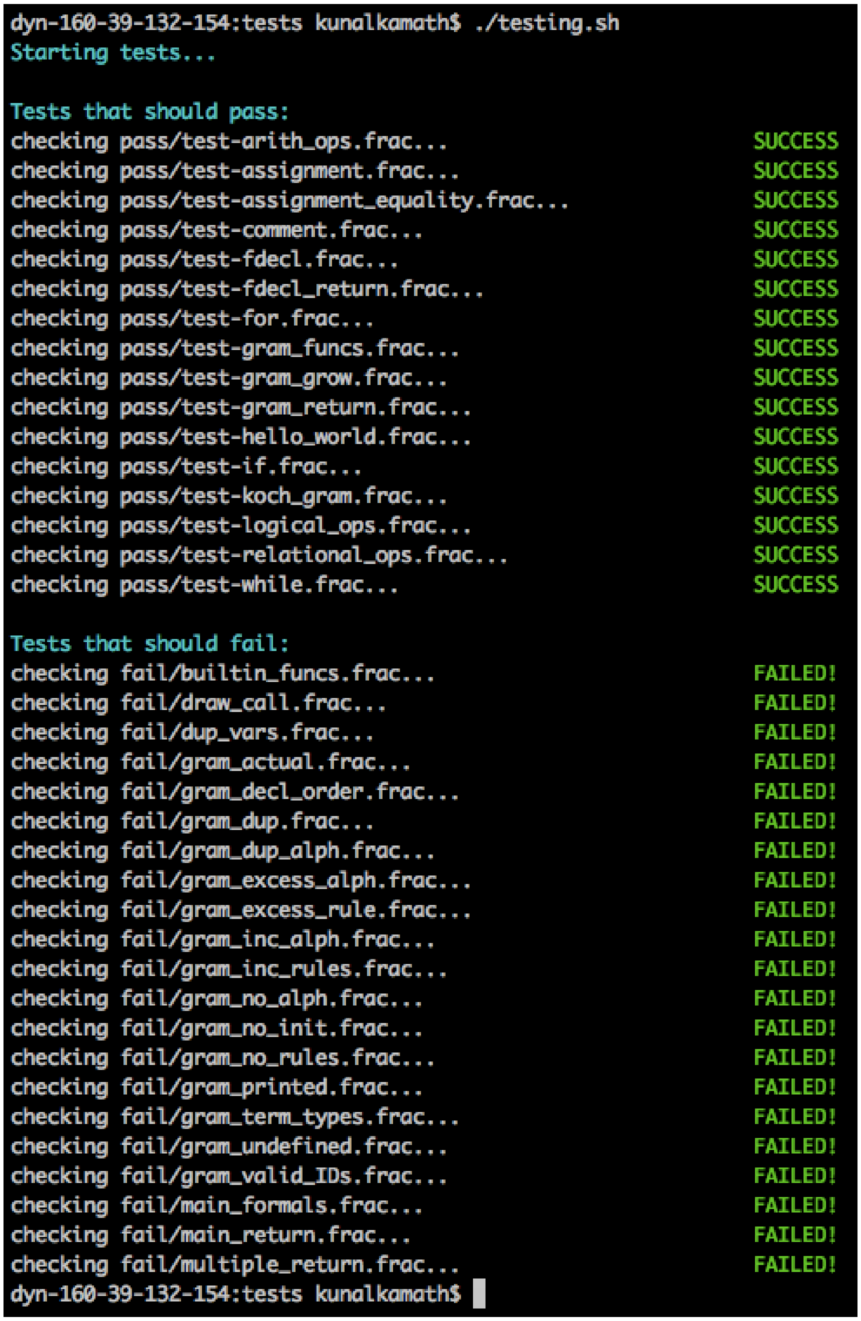
```
Fatal error: exception Parsing.Parse_error
```

We built a regression test suite by automating the testing process with a shell script, testing.sh. This script went through the list of tests we had amassed and compared the compiled output of each frac program to the intended output. Note that there are some .c files in these folders as well because we began the testing process by comparing the c program generated by the compiled frac program with the intended c file before realizing that this would be too laborious. The regression test suite was an effective strategy because, since we wrote a new test each time we implemented a new feature, checking all of our old tests continuously helped ensure that our new features didn't break any old ones. Below is the intended output of our test script, if all of the tests in the pass/ directory pass as expected, and all of the tests in the fail/ directory fail as expected:

`testing.sh`

```
#!/bin/bash

NC='\033[0m'
CYAN='\033[0;36m'
RED='\033[0;31m'
GREEN='\033[0;32m'

PASS_FILES="pass/*.frac"
FAIL_FILES="fail/*.frac"
EXEC="../../frac"
C_EXEC="./a.out"

printf "${CYAN}Starting tests...\n\n"

printf "${CYAN}Tests that should pass:\n${NC}"

for input in $PASS_FILES; do

    c_file=${input/.frac/.c}
    output=${input/.frac/.txt}
    name=${input:5}
    tmp=${name/.frac/.c}
    $EXEC $input
    if [ -e "$c_file" ]; then
        diff -wB $c_file $tmp
        if [ "$?" -ne 0 ]; then
            printf "%-60s ${RED}ERROR\n${NC}" "checking contents of $c_file..." 1>&
            exit 1
        fi
    fi

    if [ -e "$output" ]; then
        gcc -g -Wall $tmp
        $C_EXEC > $tmp
        diff -wB $output $tmp
        if [ "$?" -ne 0 ]; then
            printf "%-60s ${RED}ERROR\n${NC}" "checking output of $output..." 1>&2
            rm -rf a.out.dSYM a.out
            exit 1
        fi
    fi

    rm -f $tmp
    printf "%-60s ${GREEN}SUCCESS\n${NC}" "checking $input..."

done

printf "\n${CYAN}Tests that should fail:\n${NC}"


for input in $FAIL_FILES; do

    output=${input/.frac/.txt}
    error="$($EXEC $input 2>&1)"

    if [ -e "$output" ]; then
        diff -u <(cat "$output") <(echo "$error")
        if [ "$?" -ne 0 ]; then
            printf "%-60s ${RED}DIDN'T FAIL\n${NC}" "checking output of $output..."
            exit 1
        fi
    fi

    rm -f $tmp
    printf "%-60s ${GREEN}FAILED!\n${NC}" "checking $input..."

done

rm -rf a.out.dSYM a.out .DS_Store $tmp error
exit 0
```

Testing output



```
dyn-160-39-132-154:tests kunalkamath$ ./testing.sh
Starting tests...

Tests that should pass:
checking pass/test-arith_ops.frac...                         SUCCESS
checking pass/test-assignment.frac...                        SUCCESS
checking pass/test-assignment_equality.frac...               SUCCESS
checking pass/test-comment.frac...                           SUCCESS
checking pass/test-fdecl.frac...                             SUCCESS
checking pass/test-fdecl_return.frac...                      SUCCESS
checking pass/test-for.frac...                               SUCCESS
checking pass/test-gram_funcs.frac...                        SUCCESS
checking pass/test-gram_grow.frac...                         SUCCESS
checking pass/test-gram_return.frac...                       SUCCESS
checking pass/test-hello_world.frac...                       SUCCESS
checking pass/test-if.frac...                                SUCCESS
checking pass/test-koch_gram.frac...                         SUCCESS
checking pass/test-logical_ops.frac...                       SUCCESS
checking pass/test-relational_ops.frac...                    SUCCESS
checking pass/test-while.frac...                             SUCCESS

Tests that should fail:
checking fail/builtin_funcs.frac...                          FAILED!
checking fail/draw_call.frac...                              FAILED!
checking fail/dup_vars.frac...                               FAILED!
checking fail/gram_actual.frac...                            FAILED!
checking fail/gram_decl_order.frac...                        FAILED!
checking fail/gram_dup.frac...                               FAILED!
checking fail/gram_dup_alph.frac...                          FAILED!
checking fail/gram_excess_alph.frac...                       FAILED!
checking fail/gram_excess_rule.frac...                       FAILED!
checking fail/gram_inc_alph.frac...                          FAILED!
checking fail/gram_inc_rules.frac...                         FAILED!
checking fail/gram_no_alph.frac...                           FAILED!
checking fail/gram_no_init.frac...                           FAILED!
checking fail/gram_no_rules.frac...                          FAILED!
checking fail/gram_printed.frac...                           FAILED!
checking fail/gram_term_types.frac...                        FAILED!
checking fail/gram_undefined.frac...                         FAILED!
checking fail/gram_valid_IDs.frac...                         FAILED!
checking fail/main_formals.frac...                           FAILED!
checking fail/main_return.frac...                            FAILED!
checking fail/multiple_return.frac...                        FAILED!
dyn-160-39-132-154:tests kunalkamath$
```

7. Lessons Learned

Calvin Li

Like everyone says, don't wait until the last minute to do the work, and instead come up with good concrete goals that your group can deliver incrementally. Also, periodically giving each team member a clear idea of his/her task is a good way to ensure that everyone is at least doing something, even if it's not just code. I had a tough semester, so I often found myself falling way behind my team, and sometimes I was afraid to admit it. However, my teammates were willing to help me catch up once I asked. Even so, I really wish I could have contributed more. So, if you want to feel more useful, don't be afraid to ask your team to fill you in on what's going on if you feel behind.

Anne Zhang

Perhaps the most surprising thing that I learned, which I suppose is a big part of the material in this course, is just how much goes into semantic checking in a compiler. I foolishly assumed in the beginning that implementing basic language features would be easy, and we ended up struggling with that quite a bit. However, that was also the most interesting part of writing the compiler for me, and I now have a much better understanding of, and appreciation for, everything that compilers do. In terms of team roles, I feel that I could have done a better job as manager in bringing our team together. It was difficult to get our team members motivated to work on the project when there weren't any impending hard deadlines, but I should have created and enforced additional deadlines in order to keep our team on track. I also feel that I could have pushed some of our team members to contribute more to the project.

Kunal Kamath

I learned that pair programming is imperative in a project of this magnitude. I found myself staring at my OCaml code trying to debug far too many times, and would’ve significantly benefitted from a fresh pair of eyes. Whenever I was working with Annie, even though we were usually tackling separate problems, having a partner to bounce ideas off of and talk through your code is extremely helpful. This expands to a larger lesson learned: figuring out how to best work within your team is crucial to a good experience. Communicate with your teammates every day, figure out a good workflow, and meet regularly (like, actually) if you want to do well.

8. Appendix

8.1 Scanner

`scanner.mll`

```
{ open Parser }
```

```
let num = ('-')?['0'-'9']+
let dbl = ('-')?(['0'-'9']+'.'['0'-'9']+ | '.'['0'-'9']+)
let boolean = "true" | "false"

rule token = parse
(* Whitespace *)
  [' ' '\t' '\r' '\n'] { token lexbuf }
(* Comments *)
| "/*"     { multi_comment lexbuf }
| "//"     { single_comment lexbuf }
(* Punctuation *)
| '('      { LPAREN } | ')'      { RPAREN }
| '{'      { LBRACE } | '}'      { RBRACE }
| ';'      { SEMI }   | ','      { COMMA }
(* Arithmetic Operators *)
| '+'      { PLUS }   | '-'      { MINUS }
| '*'      { TIMES }  | '/'      { DIVIDE }
| '%'      { MOD }    | '='      { ASSIGN }
(* Logical Operators *)
| "=="     { EQ }     | "!="     { NEQ }
| '<'      { LT }     | "<="     { LEQ }
| ">"      { GT }     | ">="     { GEQ }
| "||"     { OR }     | "&&"     { AND }
| '!'      { NOT }

(* Grammar Syntax *)
| "gram"   { GRAM }    | "rules"    { RULES }
| "init"   { INIT }    | "alphabet" { ALPHABET }
| ':'      { COLON }   | '''        { QUOTE }
| '['      { LSQUARE } | ']'        { RSQUARE }
| "->"     { ARROW }
| "rturn"  { RTURN }  | "lturn"    { LTURN }
| "move"     { MOVE }

(* Statements *)
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }

(* Type Names *)
| "int"    { INT }
| "double" { DOUBLE }
| "string" { STRING }
| "bool"   { BOOL }
| '"'      { read_string (Buffer.create 17) lexbuf }
| num as lxm { INT_LIT (int_of_string lxm) }
| dbl as lxm { DOUBLE_LIT (float_of_string lxm) }
| boolean as lxm  { BOOL_LIT (bool_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID (lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and read_string buf =
  parse
  | '"'       { STRING_LIT (Buffer.contents buf) } (*
  | '\\' '/'  { Buffer.add_char buf '/'; read_string buf lexbuf }
  | '\\' '\\' { Buffer.add_char buf '\\'; read_string buf lexbuf }
  | '\\' 'b'  { Buffer.add_char buf '\b'; read_string buf lexbuf }
  | '\\' 'f'  { Buffer.add_char buf '\012'; read_string buf lexbuf }
  | '\\' 'n'  { Buffer.add_char buf '\n'; read_string buf lexbuf }
  | '\\' 'r'  { Buffer.add_char buf '\r'; read_string buf lexbuf }
  | '\\' 't'  { Buffer.add_char buf '\t'; read_string buf lexbuf }
  | [^ '"' '\\']+
    { Buffer.add_string buf (Lexing.lexeme lexbuf);
      read_string buf lexbuf
    }
  | _ { raise (Failure ("Illegal string character: " ^ Lexing.lexeme lexbuf)) } *)
  | _ { Buffer.add_string buf (Lexing.lexeme lexbuf); read_string buf lexbuf}
  | eof { raise (Failure ("String is not terminated")) }

and multi_comment = parse
  "*/" { token lexbuf }
| _    { multi_comment lexbuf }

and single_comment = parse
  '\n' { token lexbuf }
| _    { single_comment lexbuf }
```

8.2 Parser

`parser.mly`

```
%{ open Ast %}

%token SEMI COMMA COLON
%token LPAREN RPAREN LBRACE RBRACE
%token PLUS MINUS TIMES DIVIDE MOD ASSIGN
```

```
%token EQ NEQ LT LEQ GT GEQ
%token OR AND NOT
%token RETURN IF ELSE FOR WHILE
%token INT DOUBLE STRING BOOL
%token GRAM ALPHABET INIT RULES
%token LSQUARE RSQUARE ARROW QUOTE HYPHEN
%token RTURN LTURN MOVE
%token  ID
%token  INT_LIT
%token  DOUBLE_LIT
%token  STRING_LIT
%token  BOOL_LIT
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT

%start program
%type  program

%%

program:
    /* nothing */      { [], [] }
  | program gdecl      { let (grams, funcs) = $1 in $2::grams, funcs }
  | program fdecl      { let (grams, funcs) = $1 in grams, $2::funcs }

/* VARIABLES */

var_type:
    INT    { Int }
  | DOUBLE { Double }
  | STRING { String }
  | BOOL   { Bool }
  | GRAM   { Gram }

vdecl:
    var_type ID SEMI              { Var($1, $2)}
  | var_type ID                   { Var($1, $2)}
  | var_type ID ASSIGN expr SEMI  { Var_Init($1, $2, $4)}

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

/* RULES */

rule_id_list:
    ID                     { [$1] }
  | rule_id_list ID  { $2 :: $1 }

comma_list:
    ID                     { [$1] }
  | comma_list COMMA ID  { $3 :: $1 }

rule:
    QUOTE ID QUOTE ARROW RTURN LPAREN expr RPAREN   { Term($2, Rturn($7)) }
  | QUOTE ID QUOTE ARROW LTURN LPAREN expr RPAREN   { Term($2, Lturn($7)) }
  | QUOTE ID QUOTE ARROW MOVE LPAREN expr RPAREN    { Term($2, Move($7)) }
  | QUOTE ID QUOTE ARROW QUOTE rule_id_list QUOTE   { Rec($2, List.rev $6) }

rule_list:
    rule                  { [$1] }
  | rule_list COMMA rule  { $3 :: $1 }

/* GRAMS */

gdecl:
    GRAM ID ASSIGN LBRACE
      ALPHABET COLON LSQUARE comma_list RSQUARE COMMA
      INIT COLON QUOTE rule_id_list QUOTE COMMA
      RULES COLON LBRACE rule_list RBRACE
    RBRACE
    { { gname = $2;
        alphabet = $8;
        init = $14;
        rules = List.rev $20 } }

 /* FUNCTIONS */

fdecl:
   ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { fname = $1;
     formals = $3;
     locals = List.rev $6;
     body = List.rev $7 } }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
    vdecl                  { [$1] }
  | formal_list COMMA vdecl { $3 :: $1 }

/* STATEMENTS */

stmt:
    expr SEMI                                            { Expr($1) }
  | RETURN expr SEMI                                     { Return($2) }
  | LBRACE stmt_list RBRACE                              { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE              { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt                  { If($3, $5, $7) }
  | FOR LPAREN expr SEMI expr SEMI expr RPAREN stmt      { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt                        { While($3, $5) }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

/* EXPRESSIONS */

expr:
    INT_LIT          { Int_lit($1) }
  | DOUBLE_LIT       { Double_lit($1) }
  | ID               { Id($1) }
  | STRING_LIT       { String_lit($1) }
  | BOOL_LIT         { Bool_lit($1) }
  | LPAREN expr RPAREN { ParenExpr($2) }
  | NOT expr         { Unop(Not, $2) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr MOD    expr { Binop($1, Mod,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr OR     expr { Binop($1, Or,    $3) }
  | expr AND    expr { Binop($1, And,   $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }
```

8.3 AST

`ast.ml`

```
(* Operators *)
type op = Add | Sub | Mult | Div | Mod | Equal | Neq | Less | Leq
| Greater | Geq | Or | And | Not

(* Variable types *)
type var_type =
    Void
  | Int
  | Double
  | String
  | Bool
  | Gram

(* Expressions *)
type expr =
    Int_lit of int
  | Double_lit of float
  | Id of string
  | String_lit of string
  | Bool_lit of bool
  | ParenExpr of expr
  | Unop of op * expr
  | Binop of expr * op * expr
  | Assign of string * expr
  | Call of string * expr list
  | Noexpr
```

```
(* Statements *)
type stmt =
    Expr of expr
  | Block of stmt list
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt

(* Variable Declarations *)
type var_decl =
    Var of var_type * string
  | Var_Init of var_type * string * expr

type term =
    Rturn of expr
  | Lturn of expr
  | Move of expr

(* Rule Definitions *)
type rule =
    Rec of string * string list
  | Term of string * term

(* Grammar Declarations *)
type gram_decl = {
  gname : string;
  alphabet : string list;
  init : string list;
  rules : rule list;
}

(* Function Declarations *)
type func_decl = {
  fname : string;
  formals : var_decl list;
  locals : var_decl list;
  body : stmt list;
}

(* Program entry point *)
type program = gram_decl list * func_decl list
```

8.4 Semantic Checker

`semantic.ml`

```
open Ast
open Sast

type symbol_table = {
  mutable vars: var_decl list;
  mutable funcs: func_decl list;
  mutable grams: gram_decl list;
}

(**************
 * Exceptions *
**************)

exception Failure of string

let op_error t = match t with
    Ast.Not -> raise (Failure("Invalid use of unop: '!'"))
  | Ast.Add -> raise (Failure("Invalid types for binop: '+'"))
  | Ast.Sub -> raise (Failure("Invalid types for binop: '-'"))
  | Ast.Mult -> raise (Failure("Invalid types for binop: '*'"))
  | Ast.Div -> raise (Failure("Invalid types for binop: '/'"))
  | Ast.Mod -> raise (Failure("Invalid types for binop: '%'"))
  | Ast.Or -> raise (Failure("Invalid types for binop: '||'"))
  | Ast.And -> raise (Failure("Invalid types for binop: '&&'"))
  | Ast.Equal -> raise (Failure("Invalid types for binop: '=='"))
  | Ast.Neq -> raise (Failure("Invalid types for binop: '!='"))
  | Ast.Less -> raise (Failure("Invalid types for binop: '<'"))
  | Ast.Greater -> raise (Failure("Invalid types for binop: '>'"))
  | Ast.Leq -> raise (Failure("Invalid types for binop: '<='"))
  | Ast.Geq -> raise (Failure("Invalid types for binop: '>='"))

(**************
 * Checking *
**************)

let rec check_expr (env : symbol_table) (expr : Ast.expr) = match expr with
    Noexpr -> Sast.Noexpr, Void
  | Id(str) -> (match (find_vname str env.vars) with
                  Var(vt, s) -> Sast.Id(s), vt
                | Var_Init(vt, s, e) -> Sast.Id(s), vt)
  | Int_lit(i) -> Sast.Int_lit(i), Sast.Int
  | Double_lit(d) -> Sast.Double_lit(d), Sast.Double
  | String_lit(s) -> Sast.String_lit(s), Sast.String
  | Bool_lit(b) -> Sast.Bool_lit(b), Sast.Boolean
  | ParenExpr(e) -> check_paren_expr env e
  | Unop(_, _) as u -> check_unop env u
  | Binop(_, _, _) as b -> check_binop env b
  | Assign(_, _) as a -> check_assign env a
  | Call(_, _) as c -> check_call env c

and check_paren_expr (env : symbol_table) pe =
  let e = check_expr env pe in
  let (_, t) = e in
  Sast.ParenExpr(e), t

and find_vname (vname : string) (vars : Sast.var_decl list) = match vars with
    [] -> raise(Failure "variable not defined")
  | hd :: tl -> let name = (match hd with
                              Var(vt, s) -> s
                            | Var_Init(vt, s, e) -> s) in
                if(vname = name) then hd
                else find_vname vname tl

and check_unop (env : symbol_table) unop = match unop with
  Ast.Unop(op, e) ->
    (match op with
      Not ->
        let expr = check_expr env e in
        let (_, t) = expr in
        if (t <> Boolean)
          then op_error op
        else Sast.Unop(op, expr), t
      | _ -> raise (Failure "Invalid unary operator"))
  | _ -> raise (Failure "Invalid unary operator")

and check_binop (env : symbol_table) binop = match binop with
  Ast.Binop(ex1, op, ex2) ->
    let e1 = check_expr env ex1 and e2 = check_expr env ex2 in
    let (_, t1) = e1 and (_, t2) = e2 in
    let t = match op with
        Mod ->
          if (t1 <> Int || t2 <> Int)
                then op_error op
          else Sast.Int
      | Add | Sub | Mult | Div ->
          if (t1 <> Int || t2 <> Int) then
            if (t1 <> Double || t2 <> Double)
              then op_error op
            else Sast.Double
          else Sast.Int
      | Greater | Less | Leq | Geq ->
          if (t1 <> Int || t2 <> Int) then
            if (t1 <> Double || t2 <> Double)
              then op_error op
            else Sast.Boolean
          else Sast.Boolean
      | And | Or ->
          if (t1 <> Boolean || t2 <> Boolean)
            then op_error op
          else Sast.Boolean
      | Equal | Neq ->
          if (t1 <> Int || t2 <> Int) then
            if (t1 <> Double || t2 <> Double) then
              if (t1 <> Boolean || t2 <> Boolean)
                then op_error op
              else Sast.Boolean
            else Sast.Boolean
          else Sast.Boolean
      | _ -> raise (Failure "Invalid binary operator")
    in Sast.Binop(e1, op, e2), t
  | _ -> raise (Failure "Not a binary operator")

and check_assign (env : symbol_table) a = match a with
  Ast.Assign(id, expr) ->
```

```
    let vdecl = find_vname id env.vars in
    let (t,n) = (match vdecl with
              Var(vt, s) -> (vt,s)
            | Var_Init(vt, s, e) -> (vt,s)) in
    let e = check_expr env expr in
    let (_, t2) = e in
    if t <> t2 then raise (Failure "Incorrect type for assignment") else
    Sast.Assign(n, e), t
  | _ -> raise (Failure "Not a valid assignment")

and check_call (env : symbol_table) c = match c with
    Ast.Call(f, actuals) -> (match f with
        "print" -> (match actuals with
            []        -> raise(Failure "print() requires an argument")
          | hd :: []     -> let (id, t) = check_expr env hd in (match t with
                                Sast.Void -> raise(Failure "cannot print an
                                expression of type void")
                              | _ -> Sast.Call(f, [(id, t)]), Sast.Void)
          | hd :: tl -> raise(Failure "print() only takes one argument"))
      | "draw" -> (match actuals with
            [g; i] -> (match (g, i) with
                          (Id(s), Int_lit(n)) -> ignore(try
                                                  List.find(fun gram ->
                                                  gram.gname = s) env.grams
                                                  with Not_found ->
                                                  raise(Failure ("gram " ^ s ^ " not defined")));
                          Sast.Call(f, [Sast.Id(s), Sast.Gram; Sast.Int_lit(n), Sast.Int]), Sast.Void
                        | _ -> raise(Failure "draw takes a gram g and int n as arguments"))
          | _      -> raise(Failure "draw() requires two arguments"))
      | "grow" -> (match actuals with
            [g; i] -> (match (g, i) with
                          (Id(s), Int_lit(n)) -> ignore(try
                               List.find(fun gram -> gram.gname = s) env.grams
                               with Not_found -> raise(Failure ("gram " ^ s ^ " not defined")));
                          Sast.Call(f, [Sast.Id(s), Sast.Gram; Sast.Int_lit(n), Sast.Int]), Sast.Void
                        | _ -> raise(Failure "grow takes a gram g and int n as arguments"))
          | _      -> raise(Failure "draw() requires two arguments"))
      | _ -> let called_func = (try
                List.find(fun func -> func.fname = f) env.funcs
                with Not_found -> raise(Failure ("function " ^ f ^ " not defined"))) in
             Sast.Call(f, (check_args env (called_func.formals, actuals))), called_func.rtype)
  | _ -> raise (Failure "Not a valid function call")

and check_args (env : symbol_table) ((formals : var_decl list), (actuals : Ast.expr list)) =
  match (formals, actuals) with
    ([], []) -> []
  | (f_hd :: f_tl, a_hd :: a_tl) ->
      let f_type = (match f_hd with
                Var(t, _) -> t
              | Var_Init(t, _, _) -> t) in
      let (a_expr, a_type) = check_expr env a_hd in
                             if (f_type <> a_type) then raise (Failure "wrong argument type")
                             else (a_expr, a_type) :: check_args env (f_tl, a_tl)
  | (_, _) -> raise (Failure "wrong number of arguments")

let check_vtype (t : Ast.var_type) = match t with
    Int    -> Sast.Int
  | Double -> Sast.Double
  | String -> Sast.String
  | Bool   -> Sast.Boolean
  | Gram   -> Sast.Gram
  | _      -> raise (Failure "Variables cannot be of this type.")

let rec check_dup_vdecl (vname : string) (vars : Sast.var_decl list) = match vars with
    [] -> vname
  | hd :: tl -> (match hd with
                    Var(_, name) -> if(name = vname) then raise(Failure ("variable " ^
                    vname ^ " already declared"))
                                    else check_dup_vdecl vname tl
                  | Var_Init(_, name, _) -> if(name = vname) then raise(Failure ("variable " ^
                  vname ^ " already declared"))
                                            else check_dup_vdecl vname tl
                )

let check_vdecl (env : symbol_table) (v : Ast.var_decl) =
  (match v with
    Var(t, name) ->
      ignore(check_dup_vdecl name env.vars);
      let t = check_vtype t in Sast.Var(t, name)
  | Var_Init(t, name, expr) ->
      ignore(check_dup_vdecl name env.vars);
      let t = check_vtype t in
      let expr = check_expr env expr in
      let (_, t2 ) = expr in
      if t <> t2 then raise (Failure "Incorrect type for variable initialization")
      else Sast.Var_Init(t, name, expr))

let rec check_vdecl_list (env : symbol_table) (vl : Ast.var_decl list) = match vl with
    [] -> []
  | hd :: tl -> let checked_vdecl = check_vdecl env hd in
                checked_vdecl :: (check_vdecl_list { vars = (checked_vdecl :: env.vars);
                funcs = env.funcs; grams = env.grams } tl)

let rec check_stmt (env : symbol_table) (s : Ast.stmt) = match s with
    Block(sl) -> Sast.Block(check_stmt_list env sl)
  | Expr(e) -> Sast.Expr(check_expr env e)
  | Return(e) -> Sast.Return(check_expr env e)
  | If(e, s1, s2) ->
    let expr = check_expr env e in
    let (_, t) = expr in
    if t <> Sast.Boolean then
      raise (Failure "If statement uses a boolean expression")
    else
      let stmt1 = check_stmt env s1 in
      let stmt2 = check_stmt env s2 in
      Sast.If(expr, stmt1, stmt2)
  | For(e1, e2, e3, s) ->
    let ex1 = check_expr env e1 in
    let ex2 = check_expr env e2 in
    let (_, t) = ex2 in
    if t <> Sast.Boolean then
      raise (Failure "For statement uses a boolean expression")
    else
      let ex3 = check_expr env e3 in
      let stmt = check_stmt env s in
      Sast.For(ex1, ex2, ex3, stmt)
  | While(e, s) ->
    let expr = check_expr env e in
    let (_, t) = expr in
    if t <> Sast.Boolean then
      raise (Failure "While statement uses a boolean expression")
    else
```

```
      let stmt = check_stmt env s in
      Sast.While(expr, stmt)

and check_stmt_list (env : symbol_table) (sl : Ast.stmt list) = match sl with
    [] -> []
  | hd :: tl -> (check_stmt env hd) :: (check_stmt_list env tl)

let rec find_rtype (env : symbol_table) (body : Ast.stmt list) (rtype : Sast.var_type) =
  match body with
    [] -> rtype
  | hd :: tl -> (match hd with
      Return(e) -> if (rtype <> Sast.Void)
        then raise(Failure "function cannot have multiple return statements")
                   else let (_, t) = (check_expr env e) in find_rtype env tl t
    | _ -> find_rtype env tl rtype)

let sast_fdecl (env : symbol_table) (f : Ast.func_decl) =
  let checked_formals = check_vdecl_list env f.formals in
  let formals_env = { vars = env.vars @ checked_formals; funcs = env.funcs; grams = env.grams } in
  let checked_locals = check_vdecl_list formals_env f.locals in
  let new_env = { vars = formals_env.vars @ checked_locals; funcs = env.funcs; grams = env.grams } in
  { fname = f.fname; rtype = (find_rtype new_env f.body Sast.Void);
  formals = checked_formals; locals = checked_locals; body = (check_stmt_list new_env f.body) }

(* returns an updated func_decl with return type *)
let check_fdecl (env : symbol_table) (f : Ast.func_decl) = match f.fname with
    "main" -> (match f.formals with
        [] -> let sast_main = sast_fdecl env f in if (sast_main.rtype <> Sast.Void)
        then raise(Failure "main function should not return anything")
              else sast_main
      | _  -> raise(Failure "main function cannot have formal parameters"))
  | _ -> sast_fdecl env f

(* checks the list of function declarations in the program *)
let rec check_fdecl_list (env : symbol_table ) (fdecls : Ast.func_decl list) = match fdecls with
    []       -> raise(Failure "Valid FRAC program must have at least a main function")
  | hd :: [] -> if hd.fname <> "main" then raise(Failure "main function must be defined last")
                else (check_fdecl env hd) :: env.funcs
  | hd :: tl -> if (List.exists (fun func -> func.fname = hd.fname) env.funcs)
  then raise(Failure("function " ^ hd.fname ^ "() defined twice"))
                else match hd.fname with
                    "print" -> raise(Failure "reserved function name 'print'")
                  | "draw"  -> raise(Failure "reserved function name 'draw'")
                  | "grow"  -> raise(Failure "reserved function name 'grow'")
                  | "main"  -> raise(Failure "main function can only be defined once")
                  | _ -> check_fdecl_list { vars = env.vars;
                  funcs = (check_fdecl env hd) :: env.funcs; grams = env.grams } tl

let rec find_rule (id : string) (rules : Ast.rule list) = match rules with
    [] -> raise(Failure "all elements of the alphabet must have corresponding rules")
  | hd :: tl -> (match hd with
                    Rec(c, rl) -> if(c = id) then c
                                  else find_rule id tl
                  | Term(c, t) -> if(c = id) then c
                                  else find_rule id tl)

let rec check_alphabet (checked : string list) (rules : Ast.rule list) (a : string list) =
  match a with
    [] -> []
  | hd :: tl -> if(List.mem hd checked) then raise(Failure "cannot have duplicates in alphabet")
                else let checked_c = find_rule hd rules in
                checked_c :: (check_alphabet (checked_c :: checked) rules tl)

let rec check_rule (a : string list) (i : string list) = match i with
    [] -> []
  | hd :: tl -> ignore(try List.find (fun id -> id = hd) a with Not_found ->
  raise(Failure "contains a rule not found in alphabet"));
                hd :: (check_rule a tl)

let check_turn_expr (e : Ast.expr) = match e with
    Int_lit(i) -> Sast.Int_lit(i)
  | Double_lit(d) -> Sast.Double_lit(d)
  | _ -> raise(Failure "turn functions must have argument of type int or double")

let check_move_expr (e : Ast.expr) = match e with
    Int_lit(i) -> Sast.Int_lit(i)
  | _ -> raise(Failure "move functions must have argument of type int")

let rec check_rules (recs : Sast.rule list) (terms : Sast.rule list) (a : string list)
(rules : Ast.rule list) = match rules with
    []       -> recs, terms
  | hd :: tl -> (match hd with
                    Rec(c, rl) -> ignore(try List.find (fun id -> id = c) a
                    with Not_found -> raise(Failure "rule not found in alphabet"));
                      ignore(if(List.exists (fun (rl : Sast.rule) -> match rl with
                          Rec(id, _) -> if(id = c) then true else false
                        | Term(_, _) -> false) recs)
                        then raise(Failure "multiple recursive rules of the same name")
                      else check_rule a rl); let checked_rec = Sast.Rec(c, rl) in
                      check_rules (checked_rec :: recs) terms a tl
                  | Term(c, t) -> ignore(try List.find (fun id -> id = c) a
                  with Not_found -> raise(Failure "rule not found in alphabet"));
                      if(List.exists (fun (t : Sast.rule) -> match t with
                          Term(id, _) -> if(id = c) then true else false
                        | Rec(_, _) -> false) terms)
                        then raise(Failure "multiple terminal rules of the same name")
                      else let checked_t = (match t with
                          Rturn(e) -> Sast.Rturn(check_turn_expr e)
                        | Lturn(e) -> Sast.Lturn(check_turn_expr e)
                        | Move(e) -> Sast.Move(check_move_expr e)) in
                      let checked_term = Sast.Term(c, checked_t) in
                      check_rules recs (checked_term :: terms) a tl
                )

let check_gdecl (g : Ast.gram_decl) =
  let checked_alphabet = check_alphabet [] g.rules g.alphabet in
  let (checked_recs, checked_terms) = check_rules [] [] checked_alphabet g.rules in
  let checked_init = check_rule checked_alphabet g.init in
  { gname = g.gname; alphabet = checked_alphabet; init = checked_init;
  rec_rules = checked_recs; term_rules = checked_terms }

let rec check_gdecl_list (checked_gdecls : Sast.gram_decl list) (gdecls : Ast.gram_decl list) =
match gdecls with
    [] -> checked_gdecls
  | hd :: tl -> if (List.exists (fun gram -> gram.gname = hd.gname) checked_gdecls)
  then raise(Failure("gram " ^ hd.gname ^ " defined twice"))
                else check_gdecl_list ((check_gdecl hd) :: checked_gdecls) tl

(* entry point *)
let check_program (prog : Ast.program) =
  let (gdecls, fdecls) = prog in
  let env = { vars = []; funcs = []; grams = [] } in
  let checked_gdecls = check_gdecl_list [] (List.rev gdecls) in
  let grams_env = { vars = env.vars; funcs = env.funcs; grams = checked_gdecls } in
  let checked_fdecls = check_fdecl_list grams_env (List.rev fdecls) in
  checked_gdecls, checked_fdecls
```

8.5. SAST

`sast.ml`

```
open Ast

(* Variable types *)
type var_type =
    Void
  | Int
  | Double
  | String
  | Boolean
  | Gram

(* Variable Declarations*)
and var_decl =
    Var of var_type * string
  | Var_Init of var_type * string * expression

and term =
    Rturn of expr
  | Lturn of expr
  | Move of expr

(* Rule Definitions *)
and rule =
    Rec of string * string list
  | Term of string * term

(* Grammar Declarations *)
and gram_decl = {
  gname : string;
  alphabet : string list;
  init : string list;
  rec_rules : rule list;
  term_rules : rule list;
}
```

```
(* Function Declarations *)
and func_decl = {
  fname: string;
  rtype: var_type;
  formals: var_decl list;
  locals: var_decl list;
  body: stmt list;
}

(* Expressions *)
and expr =
    Noexpr
  | Int_lit of int
  | Double_lit of float
  | Id of string
  | String_lit of string
  | Bool_lit of bool
  | ParenExpr of expression
  | Unop of op * expression
  | Binop of expression * op * expression
  | Assign of string * expression
  | Call of string * expression list

and expression = expr * var_type

(* Statements *)
and stmt =
    Expr of expression
  | Block of stmt list
  | Return of expression
  | If of expression * stmt * stmt
  | For of expression * expression * expression * stmt
  | While of expression * stmt


type program = gram_decl list * func_decl list
```

8.6. Code Generator

`compile.ml`

```
open Ast
open Sast
```

```
let suffix_char s c = s ^ String.make 1 c

let c_print_types t = match t with
    Void    -> ""
  | Int     -> "\"%d\\n\""
  | Double  -> "\"%.2f\\n\""
  | String  -> "\"%s\\n\""
  | Boolean -> "\"%d\\n\""
  | Gram    -> ""

let rec expr = function
    Int_lit(i) -> string_of_int i
  | Bool_lit(b) -> if b == true then "1" else "0"
  | Double_lit(d) -> if String.get (string_of_float d) (String.length
    (string_of_float d) - 1) == '.'
                        then suffix_char (string_of_float d) '0'
                     else string_of_float d
  | Id(str) -> str
  | String_lit(s) -> "\"" ^ s ^ "\""
  | ParenExpr((e,_)) -> "(" ^ (expr e) ^ ")"
  | Unop(op, (e,_)) -> (match op with
        Not -> " ! "
      | _   -> ""
    ) ^ (expr e)
  | Binop ((e1,_), op, (e2,_)) -> (expr e1) ^ (match op with
        Add     -> " + "
      | Sub     -> " - "
      | Mult    -> " * "
      | Div     -> " / "
      | Mod     -> " % "
      | Equal   -> " == "
      | Neq     -> " != "
      | Less    -> " < "
      | Leq     -> " <= "
      | Greater -> " > "
      | Geq     -> " >= "
      | And     -> " && "
      | Or      -> " || "
      | _       -> ""
    ) ^ (expr e2)
  | Assign (str, (e,_)) -> str ^ " = " ^ (expr e)
  (* This DEFINITELY needs to be made more efficient *)
  | Call (fname, actuals) -> (match fname with
      "print" -> "printf(" ^
                 (let actuals_type = function
                   [] ->  ""
                   | (_,t)::[] -> c_print_types t
                   | _ -> ""
                   in actuals_type actuals)
                 ^ ", " ^
                 (let rec gen_actuals = function
                    [] ->  ""
                    | (e,_)::[] -> expr e
                    | _ -> ""
                    in gen_actuals actuals) ^ ")"
      | "draw" -> "turtle_init(2000, 2000);\n" ^
                (match actuals with
                    [Sast.Id(s), Sast.Gram; Sast.Int_lit(n), Sast.Int] ->
                      (s ^ "_start(" ^ (string_of_int n) ^ ");\nturtle_save_bmp(\"" ^
                      s ^ ".bmp\");\nturtle_cleanup()")
                  | _ -> raise(Failure "wrong argument types in draw()"))
      | "grow" -> (match actuals with
                    [Sast.Id(s), Sast.Gram; Sast.Int_lit(n), Sast.Int] ->
                      "char buf[1024];\nint i;\nfor(i = 0; i <" ^ (string_of_int n) ^
                      "; i++) {\nturtle_init(2000, 2000);\n" ^ s
                      ^"_start(i+1);\n" ^ "sprintf(buf, \"" ^ s ^
                      "%02d.bmp\", i);\nturtle_save_bmp(buf);\nturtle_cleanup();\n}\n"
                  | _ -> raise(Failure "wrong argument types in grow()"))
      | _       -> fname ^ "(" ^
                 (let rec gen_actuals = function
                    [] ->  ""
                    | (e,_)::[] -> expr e
                    | (e,_)::tl -> expr e ^ ", " ^ gen_actuals tl
                    in gen_actuals actuals) ^ ")")
  | Noexpr -> ""

let rec stmt = function
    Block sl -> String.concat "" (List.map stmt sl)
  | Expr (e,_) -> (match e with
                    Call(f, _) -> (match f with
                                    "grow" -> (expr e)
                                  | _      -> (expr e) ^ ";\n")
                  | _          -> (expr e) ^ ";\n")
  | Return (e,_) -> "return " ^ (expr e) ^ ";\n"
  | If ((e,_), st, Block[]) -> "if(" ^ (expr e) ^ ") {\n" ^ (stmt st) ^ "}\n"
  | If ((e,_), st1, st2) -> "if(" ^ (expr e) ^ ") {\n" ^ (stmt st1) ^ "}\n" ^
                            "else" ^ "{\n" ^ (stmt st2) ^ "}\n"
  | For ((e1,_), (e2,_), (e3,_), st) -> "for(" ^ (expr e1) ^ "; " ^ (expr e2) ^
    "; " ^ (expr e3) ^ ") {\n" ^ (stmt st) ^ "}\n"
  | While ((e,_), st) -> "while(" ^ (expr e) ^ ") {\n" ^ (stmt st) ^ "}\n"

let rec gen_var_types = function
      Void    -> "void "
    | Int     -> "int "
    | Double  -> "double "
    | String  -> "char *"
    | Boolean -> "int "
    | Gram -> ""

let gen_formals v =
    (match v with
       Var(var_type, str) -> gen_var_types var_type ^ str
     | Var_Init(var_type, str, _) -> gen_var_types var_type ^ str)

 let gen_locals v =
  (match v with
       Var(var_type, str) -> gen_var_types var_type ^ str
     | Var_Init(var_type, str, (e,_)) -> gen_var_types var_type ^ str ^ " = " ^ (expr e))

let rec gen_formals_list fl = match fl with
  [] -> ""
  | hd::[] -> gen_formals hd
  | hd::tl -> gen_formals hd ^ ", " ^ gen_formals_list tl

let rec gen_locals_list ll = match ll with
  [] -> ""
  | hd::[] -> gen_locals hd ^ ";\n"
  | hd::tl -> gen_locals hd ^ ";\n" ^ gen_locals_list tl

let gen_fdecl fdecl =
  (match fdecl.fname with
      "main" -> "int main()"
    | _      -> (match fdecl.rtype with
                    Sast.Void    -> "void "
                  | Sast.Int     -> "int "
                  | Sast.Double  -> "double "
                  | Sast.String  -> "char *"
                  | Sast.Boolean -> "int "
                  | Sast.Gram -> "")
  ^ fdecl.fname ^ "(" ^ (gen_formals_list fdecl.formals) ^ ")") ^ "{\n" ^
  (gen_locals_list fdecl.locals) ^ String.concat "" (List.map stmt fdecl.body) ^
  (match fdecl.fname with
      "main" -> "return 0;\n"
    | _      -> "" )
  ^ "}\n"

let rec divide_term_rules (tm, rtm) (recs : Sast.rule list) (terms : Sast.rule list) =
match terms with
    [] -> tm, rtm
  | hd :: tl -> let id = (match hd with
                    Term(name, _) -> name
                  | Rec(name, _) -> name) in
                if(List.exists (fun (rl : Sast.rule) -> match rl with
                      Rec(s, _) -> if(s = id) then true else false
                    | Term(_, _) -> false) recs) then divide_term_rules (tm, hd :: rtm) recs tl
                else divide_term_rules (hd :: tm, rtm) recs tl

let gen_term_arg (e : Sast.expr) = match e with
    Int_lit(i) -> string_of_int i
  | Double_lit(d) -> string_of_float d
  | _ -> ""

let rec gen_term_rules (terms : Sast.rule list) = match terms with
    [] -> ""
  | hd :: tl -> let (id, t) = (match hd with
                    Term(name, tp) -> name, tp
                  | Rec(_, _) -> raise(Failure "should be a terminal rule")) in
                    "if (var == '" ^ id ^ "') {\n" ^
                (match t with
                    Rturn(e) -> "turtle_turn_right(" ^ (gen_term_arg e) ^ ");\n"
                  | Lturn(e) -> "turtle_turn_left(" ^ (gen_term_arg e) ^ ");\n"
                  | Move(e) -> "turtle_forward(" ^ (gen_term_arg e) ^ ");\n"
                ) ^ "}\n" ^ gen_term_rules tl

let rec gen_init (gname : string) (rl : string list) = match rl with
    [] -> ""
  | hd :: tl -> gname ^ "('" ^ hd ^ "', iter);\n" ^ gen_init gname tl

let rec gen_rule (gname : string) (rl : string list) = match rl with
    [] -> ""
  | hd :: tl -> gname ^ "('" ^ hd ^ "', iter - 1);\n" ^ gen_rule gname tl

let rec gen_rec_rules (gname : string) (recs : Sast.rule list) = match recs with
    [] -> ""
  | hd :: tl -> let (id, rl) = (match hd with
                    Rec(name, rule) -> name, rule
                  | Term(_, _) -> raise(Failure "should be a recursive rule")) in
                "if(var == '" ^ id ^ "') {\n" ^ (gen_rule gname rl) ^ "}\n" ^
                  (gen_rec_rules gname tl)

let gen_gdecl (g : Sast.gram_decl) =
  let (terms, rterms) = divide_term_rules ([], []) g.rec_rules g.term_rules in
  "void " ^ g.gname ^ "(char var, int iter) {\n" ^ "if (iter < 0) {\n" ^
  (gen_term_rules rterms) ^ "} else {\n" ^ (gen_rec_rules g.gname g.rec_rules) ^
    (gen_term_rules terms) ^ "}\n}\n" ^
  "void " ^ g.gname ^ "_start(int iter) {\n" ^ (gen_init g.gname (List.rev g.init)) ^ "}\n"

let generate (grams : Sast.gram_decl list) (funcs : Sast.func_decl list) (name : string) =
  let outfile = open_out (name ^ ".c") in
  let translated_program =  (if List.length grams > 0
  then "#include \"turtle.h\"\n#include \n" else "") ^ "#include \n\n" ^
  String.concat "" (List.rev (List.map gen_gdecl grams)) ^
  String.concat "" (List.rev (List.map gen_fdecl funcs)) ^ "\n" in
  ignore(Printf.fprintf outfile "%s" translated_program);
  close_out outfile;
```

8.7. FRAC

`frac.ml`

```
type action = Semantic | Compile

(* Get the name of the program from the file name. *)
let get_prog_name source_file_path =
  let split_path = (Str.split (Str.regexp_string "/") source_file_path) in
  let file_name = List.nth split_path ((List.length split_path) - 1) in
  let split_name = (Str.split (Str.regexp_string ".") file_name) in
    List.nth split_name ((List.length split_name) - 2)

let _ =
  let name = get_prog_name Sys.argv.(1) in
  let path = Sys.getcwd() ^ "/" ^ name in
  let input = open_in Sys.argv.(1) in
  let lexbuf = Lexing.from_channel input in
  let program = Parser.program Scanner.token lexbuf in
  let (grams, funcs) = Semantic.check_program program in
  Compile.generate grams funcs path
```