GASSP Final Report

Adam Fowler (ajf2177), Patrycja Przewoznik (pap2154),
Samuel Weissmann (spw2136), Swan Htet (sh3969), Yuanxin Yang (yy3036)

April 26, 2021

Contents

1. Introduction
GASSP is a statically typed object-oriented general purpose programming language with its roots in C++ and Java. As an object-oriented language, GASSP does not attempt to needlessly reinvent the wheel, but rather it adopts an Umbridgean worldview: to preserve what must be preserved, perfect what can be perfected, and prune practices that ought to be prohibited. Through this philosophy we hope to embrace the strengths of the object-oriented paradigm without falling into the common trap of trying to turn a tool into a toolbox and while striving to shield the user from unintended consequences.

With GASSP, we aim to provide a simple and intuitive language free of syntactic ambiguity that does its best to limit the users' ability to create unpredictable or unintended behaviors. It features strong static typing, replaces null with the option type, does not perform type coercion, does not permit function overloading, and avoids undefined behaviors wherever possible.

2. Language Tutorial
Using Gassp is a straightforward process, the syntax will be very familiar to anyone who has used C, C++, or Java. In many ways, Gassp represents a stripped down version of these languages. Any syntactic differences can be resolved by consulting the LRM, which is contained in sections 3-6.

2.1 Environment
We have included a Dockerfile that provides the appropriate environment for Gassp, and build, start, and stop shell scripts to help users get their Docker image up and running. Once Docker is running, users should be able to use the different included Makefiles and their various commands to build the language and compile and run test programs. Further information on these steps can be found by consulting the relevant readme files included with the language.

3. Lexical Conventions

3.1 Comments
Multi-line comments are enclosed inside `/* */`. GASSP does not support single line comments.

```
/* This is a multi-
line comment. It's readable;
but not poetry */
```

3.2 Identifiers
GASSP recognizes ASCII symbols, and identifiers may be constructed from a combination ASCII letters, digits, and underscores. Identifiers must begin with a letter or underscore and cannot override keywords.

3.3 Keywords
Keywords are reserved in GASSP and they cannot be used as identifiers.
GASSP keywords include:

```
if          else       false      true
string      pass       int        class
for         while      float      main
public      break      bool       void
private     continue   char       return
```

3.4 Operators

GASSP features a standard set of built-in operators that can be broadly classified into four categories.

Arithmetic	`+  -  *  /  %  **  ++  --`
Comparison	`==  !=  <  <=  >  >=`
Logical	`&&  ||  !`
Bitwise	`&	^ << >> ~`

3.4.1 Arithmetic Operators:

Arithmetic operators can be used on integers and floats (with one exception, see 2.4.4). Operands must be of the same type.

3.4.2 Comparison Operators:

Comparison operators can be used on integers, floats, chars, and strings. The operands must be of the same type. Strings and chars are compared on their ASCII lexicographic order.

3.4.3 Logical Operators:

Logical operators can be used on booleans or expressions that evaluate to booleans.

3.4.4 Operator Precedence

From highest to lowest precedence.

Operator	Description	Associativity
`++` `--`	Increment Decrement	Right to left

`!` `~`	Logical not Bitwise not	Right to left
`**`	Exponentiation	Right to left
`* / %`	Multiplicative	Left to right
`+ -` `+`	Additive String concatenation	Left to right
`>> <<`	Shifts	Left to right
`&`	Bitwise and	Left to right
`^`	Bitwise xor	Left to right
`|`	Bitwise or	Left to right
`== !=`	Equality	Left to right
`< > <= >=`	Comparison	Left to right
`&&`	Logical and	Left to right
`||`	Logical or	Left to right
`=`	Assignment	Right to left

3.5 Literals

Literals are constant values for one of GASSP's built-in types (i.e. a primitive or string).

3.5.1 Integer Literals

An integer literal is any sequence of one or more decimal digits. Integer literals may not begin with a 0.

3.5.2 Float Literals

A float literal is a sequence of one or more decimal digits followed by a decimal point (`.`) and another sequence of one more decimal digits.

3.5.3 Boolean Literals

Boolean literals are represented with the keywords `true` and `false`.

3.5.4 Char Literals

A char literal is any of the ASCII values enclosed in single quotation marks.

3.5.5 String Literals

A string literal is any sequence of chars enclosed in double quotation marks.

4. Data Types

All data in GASSP falls into one of two categories: primitives or objects. Data types are not inferred at runtime and need to be explicitly declared.

4.1 Primitives

GASSP has four primitive data types. It should be noted that unlike in some other common languages, the char data type can not be used as an integer value, and attempting to do so will result in an error.

Name	Size	Operators	Syntax
`int`	4 bytes	All	`int x = 3;`
`float`	8 bytes	All	`float x = 3.14;`
`boolean`	1 byte	Logical, Comparison	`bool x = true;`
`char`	1 byte	Comparison	`char x = 'x';`

4.2 Objects

All objects have an associated type and value with it. The value may be either a primitive or a reference to another object. GASSP includes strings and arrays as built-in objects. Objects are created by calling the "new" command. They are either public or private. Methods are accessed using familiar dot (.) notation as follows:

```
bankAcc acount1 = new bankAcc(); /*creates new bankAcc object called account*/
bankAcc.balance(); /* prints "0" */
bankAcc.deposit(500);
bankAcc.balance(); /* prints "500" */
```

4.2.1 Strings

Strings in GASSP are immutable. GASSP supports string concatenation using `strcat()`.

```
string str1 = "hello";
string str2 = str_concat(str1," world"); /* "hello world" */
```

Additionally, strings support `str.upper()` and `str.lower()` functions, where `str` is a string object and they return the same string in all uppercase or all lowercase letters, respectively.

GASSP also supports returning string lengths (`strlen()`), string copy (`strcpy()`), string comparison (`strcmp()`), and checking whether a substring exists in the given string (`(strstr()`).

4.2.2 User Defined Objects

GASSP allows users to define their own objects by means of a class system. The basis class syntax is as follows:

```
public void Haiku() {

     /* constructor, and other class
     methods belong here */
}
```

A Class must have an access modifier, either public or private, a return type, a name that must start with a capital letter, a pair of parentheses with optional arguments, and a pair of curly braces that contain the class body. When the class has type void, there is no return, otherwise a return statement is required.

4.2.3 Main

The main function is the entry point of any GASSP program. It is required and the main keyword is reserved. The function returns void.

```
public int main() {

     /* statements */
}
```

5. Statements

Statements are executed in the sequence they are written, except when otherwise stated. Statement scope is demarcated with curly braces. The different types of statements are defined in the following sections.

5.1 Expression Statements

Expression statements evaluate to a particular value. Expressions are terminated with a semicolon and take the form:

expression **;**

5.1.1 Operators

5.1.2 Operator Precedence

5.2 if and if-else Statements

The `if` statement is composed of a boolean expression, a statement that is executed if the boolean expression evaluates to true, and an optional `else` statement that is executed if the boolean expression evaluates to false. Mandatory curly braces define the scope of the statements, and else ambiguity is resolved by attaching an else statement to the most recently encountered elseless `if` of the same scope level. The statements take the following forms:

`if` *boolean expression* { *statement* }
`if` *boolean expression* { *statement* } `else` { *statement* }

5.3 While Statements

The `while` statement is composed of an expression and statement in the following form:

`while` (*expression*) { *statement* }

The statement will be repeatedly executed as long as the expression evaluates to true.

5.4 For Statements

The `for` statement is composed of three expressions and a statement in the following form:

`for` (*expression-1* **;** *expression-2* **;** *expression-3*) { *statement* }

Expression-1 initializes the loop with a value; expression-2 specifies a condition that is checked before every iteration and that must evaluate to true for the loop to continue running; and expression-3 is evaluated at the end of each iteration, and is typically used to update a loop control variable.

5.5 Return Statements

The `return` statement returns control to the caller of a function, and optionally passes back the value of an expression. It has the following form:

`return` (*expression*) **;**

6. Functions

6.1 User Defined Functions

Functions in GASSP are a type of statement that may take in zero or more arguments and executes. Functions may have a type or be of type "void". Typed functions must return an expression that evaluates to their type, while void functions return nothing. Function type must be specified in function's definition. Function arguments are specified as a list of typed parameters in the function definition. Functions are defined and called as follows:

```
int sumOfInt(int x, int y) {
     return (x+y);
}
```

6.2 Standard Library

GASSP standard library consists of a small number of built-in functions.

6.2.1 Mathematical Functions

GASSP has the following mathematical functions

Min and Max

The min and max functions take in two arguments of the same type and return the smaller value or larger value respectively. Arguments may be ints or doubles. If the arguments have the same value, the result is that same value.

```
int x = min(5, 9); /* x = 5 */
int y = max(5, 9); /* y = 9 */
```

sqrt(a)

Returns the positive square root of a float value. The argument can be integer (7 significant digits) or float (16 significant digits) type. For results that exceed the precision, the number is truncated.

```
float x = sqrt(10); /* x = 3.162277660168379*/
```

rand()

Returns a float with a value in the range [0, rand_max)

```
int x = rand(); /* x = 2 */
```

6.2.2 Print Functions

GASSP includes multiple print functions. Each print function will output a specific type of data.

Print function	Printing type	Syntax

`print`	int	`print(3);`
`printf`	floating numbers	`printf(3.14);`
`printb`	boolean	`printb(true);`
`printc`	char	`printc('x');`
`prints`	string	`prints("hello");`

7. Example

GCD implemented in GASSP

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

8. Project Plan

Language Guru: Adam Fowler
- Looks at all files, mainly works on parser, semant, codegen

Tester: Patrycja Przewoznik
- Looks at all files, mainly works on scanner, semant, codegen, test cases

Manager: Samuel Weissmann
- Looks at all files, mainly works on semant, codegen

System Architect: Swan Htet, Yuanxin Yang
- Look at all files, mainly work on ast, sast, semant, codegen, test cases

All of us contributed to writing the LRM, the final report, and the presentation.

When developing GASSP, we started by consulting MicroC and Java+- (from 2016 Spring) to gain some understanding of the functionality of each file. We used docker to control the environment, and we used github for team collaboration and version control. Ocaml was used to build GASSP and llvm was used to compile GASSP. The process of development is scrum-like.

We did regular checkups, we planned the features that we will build for the next due day, we then built the features, tested them, and submitted them. The process was repeated for each milestone. Tests did not happen only at the end. We tested our code throughout the development process.

8.1 Style guide
We followed the MicroC naming conventions to compound variable names, data types, and object names. We also used descriptive names. For example, function formal declaration was named fformal.

8.2 Timeline
Our team meets weekly. Whenever the manager or any of the team members feels like we need more time to discuss the project, we meet twice a week, usually Monday 8:15pm and Thursday 8:15pm.

01/11 - 01/20: team formation, getting to know each other

01/21 - 02/03: discussed about project scopes, language features, formulated a proposal

02/04 - 02/12: discussed about language features, decided language style

02/13 - 02/24: drafting the LRM, worked on completing the parser

02/25 - 03/08: wrote scanner.ml, modified parser.ml, learned about code in ast.ml and sast.ml

03/09 - 03/17: finished ast.ml and sast.ml, learned about semant.ml and codegen.ml

03/17 - 03/24: worked on semant.ml and codegen.ml, modified parser.ml, ast.ml, and sast.ml. Wrote tests for printing numbers and strings. Completed hello world milestone

03/25 - 04/07: thought about how objects should be represented in GASSP. Consulted projects from previous semester

04/08 - 04/26: object is a new feature to add into GASSP, so did not split the tasks into files but work on them all

8.3 Project Logs
Below is a chronological git history

```
commit 353dad0dbfc2bbba59795b6fca3be600651c6aae
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 20:31:40 2021 -0400

    Removed microc ref from Dockerfile

commit 38089316b6f7d9c36caafcc8e2bb891d24402b15
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 20:20:28 2021 -0400
```

```
    add object tests

commit c51e44025eb4d85fceba8a13b91182ed1fd987b0
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 20:09:49 2021 -0400

    Fix tests

commit fd6cbb4bd58b845262a554bbd44b2a852173097e
Merge: 1b27280 e705f4e
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 20:07:21 2021 -0400

    Clean up

commit 1b2728099924fe5222e86441df504892b889caf0
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 20:06:24 2021 -0400

    Clean up, cite java+-

commit e705f4e8be419f18ee929c2c23ffa4fdc94c08c2
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Apr 26 20:00:22 2021 -0400

    create_module context javapm->gassp

commit fd4f493a03f42127bf9b4883896cb639048fc747
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 19:54:27 2021 -0400

    codegen building

commit e35cf68edf77a0ed8c38e6f88321d77a6b8fc2b1
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 19:38:39 2021 -0400

    deleted microc references

commit b96262726af40643940694cef99f632d8a90ecd6
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 09:01:36 2021 -0400

    Added env

commit e26e1be454d63a3e73d9a6e323e62050fd6fe60b
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 00:25:54 2021 -0400

    Where is printf

commit f00f1434ce5f7e8576c4a142914f018dc527593c
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 00:24:04 2021 -0400

    Types for builtins
```

```
commit 81d6b23a98bd2d788b13e2d6dbb2e3b8f3e7faf3
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 00:23:04 2021 -0400

    adding types to builtins

commit 25c4a95dd8a415c4d6c0ca3f07c8f6d74742633f
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 00:04:54 2021 -0400

    getting there

commit 08cfc3ca15166c5b49a4b92c6e8bffb515519fd2
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 25 23:36:30 2021 -0400

    sort of building

commit db598c043a8f6a1558354ce2f6009b99819914bc
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 25 23:07:05 2021 -0400

    cleaning up parser

commit e3a4a242b060ddbc2eeb0fd5fd7e33e0b310c355
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 25 21:26:05 2021 -0400

    builtin functions

commit 7fa3bc0f01eb3118f571b77ccb0d83408e51ec22
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 25 21:13:51 2021 -0400

    clean up

commit c69417fd80df263276b396c229a51400302411bc
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 25 21:08:29 2021 -0400

    codegen not working

commit 044bdaef6bb44d308a08b052bece21089e207bcc
Merge: 28b84c0 2204343
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 25 20:55:24 2021 -0400

    Merge branch 'ajf2177/objects-feat' of github.com:usddddd/PLT into
ajf2177/objects-feat

commit 28b84c017ca1cf9ae9256828b7f7bd7fd7c95bed
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 25 20:38:26 2021 -0400

    Updated parser to match latest
```

```
commit 2204343bbc8f18d06f141eef69f2c4c10a1a31d8
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 25 20:38:26 2021 -0400

    Updated parser to match latest

commit 3c8375bf1946e443aff1ff02e7b389905517a255
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 25 17:43:53 2021 -0400

    Variable declaration and definition

commit 39ff98ef9c03c02dae733fd884f91f6eff3b26e4
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 25 15:59:44 2021 -0400

    Pass builtins as function list in sast

commit 4ac8d9de8053f17541e7a9d56c32d461c5d87a18
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 25 15:58:05 2021 -0400

    Add local variables

commit 56bd71f0182ce28f8fc78bccb82a3c2f646d9c5b
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 25 13:32:54 2021 -0400

    Semant passing

commit ccb62651f57591cf5af2e381c8cbe64e7756fa48
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 25 12:33:25 2021 -0400

    Eliminate redundant methods

commit 15ae4868f67a7affd6caaf084d72eb8a39fcc91a
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 25 12:19:38 2021 -0400

    Be consistent in type naming

commit 2a93d7693914a1909a6c3fc8356ba25711e4049d
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sat Apr 24 22:33:29 2021 -0400

    Comment out pretty printing for now, semant seems to be working

commit bec6ee2aa9dc6d40c778dc798e2a2dd627e4c166
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sat Apr 24 22:09:46 2021 -0400

    No more binds

commit fb3411ba475f044a26238db6acc34ab72159bf49
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sat Apr 24 22:01:19 2021 -0400
```

```
    Convert class to sast

commit 93bba67251ac9b2fc890c7bff91dc8930baece43
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sat Apr 24 16:40:34 2021 -0400

    Use cbody record for class body instead of list of methods

commit ca43c9d32806754767df4d084be15a6cf6e88494
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sat Apr 24 13:57:11 2021 -0400

    Add explicit formals instead of bind list

commit b071f3004624769eedc2c94c064f72513a15c6de
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Thu Apr 22 22:05:32 2021 -0400

    Add main entrypoint to program definition

commit 9a5ff327bd8218050461aeb6d505a7e9b8bd50d1
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Thu Apr 22 21:10:31 2021 -0400

    Remove array from parser and use record for var definition

commit 0dd72573e726ddca70253e876f27bedd7e86f0ae
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Wed Apr 21 23:25:27 2021 -0400

    Fix parser errors

commit 9a6652e7e20bdbc421a35458aa79bb4c2c89402c
Merge: a990d05 62c6437
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Wed Apr 21 23:15:06 2021 -0400

    Merge branch 'ajf2177/objects-feat' of github.com:usdddddd/PLT into
ajf2177/objects-feat

commit a990d05544f764768bbce71a9ea873ea3d736e7a
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Wed Apr 21 23:03:49 2021 -0400

    Added types to Sast

commit 62c6437b5a4cb86b0627fd1d1fc3ec1d3e74324e
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Wed Apr 21 23:03:49 2021 -0400

    Added types to Sast

commit 27589f9372b56a0c9da7f9b39965254919c27871
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Wed Apr 21 22:50:15 2021 -0400
```

```
    Removed commented out program definition

commit b79f972dfec998377c303b8f8aba516e7eaf7e6a
Merge: 677a136 0bc46ff
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Wed Apr 21 22:49:19 2021 -0400

    Merge branch 'ajf2177/objects-feat' of github.com:usdddddd/PLT into
ajf2177/objects-feat

commit 677a1361281244eda3afe151702e409b7e80f569
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Apr 20 21:21:44 2021 -0400

    updating parser

commit 0bc46ff8e6755cbc110e4b46d19510cba62f44bf
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Apr 20 21:21:44 2021 -0400

    updating parser

commit 0f42111349fdc1e43cd785e443fb99b31c4b3c7c
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 19 22:55:51 2021 -0400

    Add class basics to ast and parser

commit 96cd6eed0aa327d06942c778723606c08c351290
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 18 20:31:50 2021 -0400

    Re-added functions to semant

commit 5ce5e8c84987f60b49b748f53ee4e5fd1b25c1e3
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Thu Apr 15 20:04:36 2021 -0400

    Changed test file ext

commit 748ff0c0e15382c3d4301d2d940ed67a17f67f02
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Thu Apr 15 19:46:56 2021 -0400

    Use .g extension for test source files

commit e74d44d8aef8401988356b4942f2c4e4f4e6236c
Merge: df0a739 49361b3
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Thu Apr 15 19:30:32 2021 -0400

    Merged swan's function call work into break/continue branch

commit 49361b336d3e51283da04f64ba5a930a29c2e8b7
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Thu Apr 15 19:02:41 2021 -0400
```

```
    Renamed to match develop

commit 8cd71445d2cbc40d2462f59a5691eeb22f742fff
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Thu Apr 15 18:56:09 2021 -0400

    added gitignore and removed unused check function

commit 2fd90285f7e1b4e5e9e3a99771378628c9e41bf6
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Wed Apr 14 23:32:07 2021 -0400

    Removed non-working grammar constructs for now

commit 23802ba60c39e98f8c88eb003e0ad1c41acb69b6
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Wed Apr 14 22:09:28 2021 -0400

    Check types for bind with value

commit df0a739a54641d117f44904b6eed44e1b7bb06e4
Author: swanhtethtetswan <ucancallmex@gmail.com>
Date:   Wed Apr 14 14:44:57 2021 -0400

    makefile withc PHONY and math and string funs

commit b89ff7544b53f18308c0b3173ef1c35c080928a7
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 12 20:37:37 2021 -0400

    Removed switch and introduced bind with value

commit 6672d6a33e149114b7e792eb0c1d2badf4850136
Author: swanhtethtetswan <ucancallmex@gmail.com>
Date:   Mon Apr 12 19:52:12 2021 -0400

    min and max together

commit bc43eb89fd5862ee8e4f519c09167cebce95ce88
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 11 12:46:04 2021 -0400

    integrating pati's changes

commit 70401653156258987be1405e5c4792a1726f5b06
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Wed Apr 7 22:48:37 2021 -0400

    break and continue loop

commit 2e5c0a10f32d160cddbabea5b66aef32919bee81
Author: swanhtethtetswan <ucancallmex@gmail.com>
Date:   Wed Apr 7 21:19:16 2021 -0400

    min and max

commit 8f0c21c541ff849bea84571788ab33b29b08a1a3
```

```
Author: swanhtethtetswan <ucancallmex@gmail.com>
Date:   Wed Apr 7 18:26:15 2021 -0400

    changed gasp to gassp and added min function

commit b7994f46b47b0c31d14d1919cecdc44cf23c90dd
Author: swanhtethtetswan <ucancallmex@gmail.com>
Date:   Wed Apr 7 18:23:47 2021 -0400

    new directory gassp

commit 638f398ef43e7b45495b038a2fa9112dc09c4c74
Author: swanhtethtetswan <ucancallmex@gmail.com>
Date:   Wed Apr 7 18:22:50 2021 -0400

    removed old stuff

commit ed8b01f2bdb0e37485d6342109c32e424f84bdf1
Merge: 75abbe5 580d094
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Mar 29 22:48:46 2021 -0400

    Merge branch 'develop' of github.com:usddddd/PLT into develop

commit 75abbe517506fae4fcf31047f18c1e6bebb50804
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Mar 29 22:27:18 2021 -0400

    removed built compiler from tracked files

commit 580d094d4d71588c9dcd4b4147d1e25160b0f695
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Mar 29 22:27:18 2021 -0400

    removed built compiler

commit 4143528f2ea2adbc51fbfe321870a7a25f17bde5
Merge: f5020aa 14f3339
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Mar 29 22:22:14 2021 -0400

    Merge branch 'feat/helloworld-building' into develop

commit 14f33390fb9b4389b2232b970df8ce04c22bba2d
Merge: 63fb08c fc59665
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Mar 29 22:03:45 2021 -0400

    Merge branch 'feat/helloworld-building' of github.com:usddddd/PLT into
feat/helloworld-building

commit 63fb08c9f14e3f60c14aa77e9c509f8f8fbd906d
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Mar 29 22:03:39 2021 -0400

    expanded Makefile's clean command
```

```
commit f5020aa75219beb1d68d121977736586b953ed1a
Author: Yuanxin-Yang <55468947+Yuanxin-Yang@users.noreply.github.com>
Date:   Mon Mar 29 21:16:02 2021 -0400

    updated array dec

commit fc5966555a45e053982fff1e507b8081df93c6d3
Author: Yuanxin-Yang <55468947+Yuanxin-Yang@users.noreply.github.com>
Date:   Mon Mar 29 21:10:41 2021 -0400

    updated array dec

commit 9922040ab46bdda892583a9b4baa901a5e25ea1f
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Mar 28 15:57:00 2021 -0400

    Switch should be a statement not expression

commit 098fc1c88672dbfb74c1a0c2e21fa61f6de64b6b
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Wed Mar 24 17:28:14 2021 -0400

    Added cases for pattern matching

commit 5e1b060a17c367feb9a144cef8572b49f7fe5e4c
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Wed Mar 24 16:53:42 2021 -0400

    Added character printing and made minor changes in scanner

commit 1c41e5b5208f94c26224c82f0ae18c57d533532b
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Wed Mar 24 02:46:22 2021 -0400

    updated Makefile2 to help build hello world file

commit d30ec87bac68999edea183b66660ff01ec28dcc8
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Wed Mar 24 02:40:04 2021 -0400

    basic helloworld file

commit 34139de73d3057084b4d9a991f55f40b618f99e5
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Wed Mar 24 02:17:09 2021 -0400

    swan's modifications to add string printing

commit 02adeee3a68c913b26feadaacc57bb867ab788fb
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 23 23:35:00 2021 -0400

    added microc

commit c45b4247c68c095cc9ba59706f9d30923c436c0a
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Mar 23 22:10:50 2021 -0400
```

```
    Builds with warnings now

commit 921626e574439d164dcf9696829f4567479f11d8
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Mar 23 20:33:18 2021 -0400

    Add string to class constructor

commit e3a27a02c6539803e8575baad892edcaa7df2022
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Mar 23 20:07:59 2021 -0400

    added missing stuff

commit 2f000085d8ef6ece207dff63b020e961d1f02f92
Merge: fecd0a4 2fb9846
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Mar 23 20:01:42 2021 -0400

    Merge branch 'develop' of github.com:usddddd/PLT into develop

commit fecd0a4208dad04782c782c8d4c027f5886dfb72
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Mar 23 19:38:45 2021 -0400

    Fixing make errors

commit 2fb9846a719aa0db9f7cb9aa77c64468b3c844fe
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Mar 23 19:38:45 2021 -0400

    Fixing make errors

commit bb2b40264b1420c7e8bd05d22e224a076eb0e493
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 23 18:18:56 2021 -0400

    added char, string, and object to string_of_typ function

commit 7c5b3b64d74711ff7a1f90fc90279a2fc703841c
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 23 18:16:04 2021 -0400

    basic Makefile

commit cc11c7c0eb2ae7a57d45083cfa3dfb6019ddd544
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 23 18:06:34 2021 -0400

    moved _tags to gasp directory

commit cfe996697eb3e5bf4b71401b7349e5c3bd4dc0d9
Merge: 8308a28 0b4f543
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 23 18:00:55 2021 -0400
```

```
    Merge branch 'develop' of github.com:usddddd/PLT into develop

commit 8308a28d23e1ce8ced68573902819f53efba4523
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 23 18:00:32 2021 -0400

    fixed ast filename

commit 9a84ac3e03b37f4e91f12682ef565da2b849b906
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 23 17:59:19 2021 -0400

    added a gassp.ml adapted from microc.ml

commit 0b4f543311386d057edb243db7e4b0a73d907b58
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Tue Mar 23 17:57:15 2021 -0400

    Added strings and chars to codegen.ml and semant.ml

commit 37aea8c782043f141f5e23fc2c00fc153d966387
Merge: d434e2b ed96a14
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Mar 22 20:10:24 2021 -0400

    Merge branch 'ajf2177/parser' into develop

commit ed96a14c43e87fefa8f1de97ce2cf6f08e7ef5ce
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Mar 22 20:05:34 2021 -0400

    Re-structured project to match microc for building with Makefile
(eventually)

commit 97af73769d8b4491c57754ec3539ab04f337cf9c
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Sun Mar 21 20:11:11 2021 -0400

    Pushing corrected sast.ml and scanner.mll

commit 6e6194cc2e0fb6e930c4640544a05b4186dcddc4
Merge: 87fce8a 278f2c6
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Sun Mar 21 18:43:58 2021 -0400

    Merge branch 'ajf2177/parser' of github.com:usddddd/PLT into ajf2177/parser

commit 87fce8a7977f37d5e6f0cb1f7b0dce501a189769
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Sun Mar 21 18:43:03 2021 -0400

    Created sast file

commit d434e2b40dd8d4eeb4b31bb37f029d15473ffc69
Merge: 0d68216 f647553
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sat Mar 20 23:32:02 2021 -0400
```

```
    Added docker build/start/stop scripts

commit 0d68216acf995fa570ea93f95a6090b4386b72bc
Author: Adam Fowler <afowler@peerstream.com>
Date:   Sat Mar 20 23:23:36 2021 -0400

    Fixed Dockerfile

commit 278f2c6728cdcd40c8b9266f880708b36de16678
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Sat Mar 20 16:33:00 2021 -0400

    finished remaining lrm sections

commit dceeeab7d785048fb85ab39c4b686d9fb3207de0
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Sat Mar 20 14:29:18 2021 -0400

    added titles to links

commit 5b53de2e21b93f640c4c4ee3621da077ba8470fd
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Sat Mar 20 14:14:17 2021 -0400

    finished writing all subsection headings, confirmed all toc linkings works

commit 37bd4cc44de0fc9656920c67d743e22698c202fd
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Sat Mar 20 14:05:39 2021 -0400

    section 3 completed

commit f3c2fe2b85a6452254dba5f0533c0a77826535c4
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Sat Mar 20 13:53:33 2021 -0400

    completed section 2

commit 3f6c1904789359572e827c5e7880c4f3e4dfc580
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Sat Mar 20 13:27:29 2021 -0400

    built and linked toc

commit ce760bc01d55d1245824311906722bd0921b81c9
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Thu Mar 18 21:01:08 2021 -0400

    did lrm skeleton and sections 1 to 2.4.1

commit c453ea3634d0d4bdc3686a94340b9285be7cf444
Author: Adam Fowler <afowler@peerstream.com>
Date:   Thu Mar 18 19:46:10 2021 -0400

    Added Dockerfile and moved project files
```

```
commit 6ff5b8c125136964f1e010ed6d63b7be24d4a45a
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Thu Mar 18 15:59:51 2021 -0400

    added bitwise xor, shifts, and exponent tokens

commit f647553acf08e5141c4702537d3b3377de0ec78b
Author: swanhtethtetswan <23371162+swanhtethtetswan@users.noreply.github.com>
Date:   Wed Mar 17 23:37:28 2021 -0400

    Create gast.ml

commit 0deb45e17be96c8f3e734c992cf4db219bb66075
Author: swanhtethtetswan <23371162+swanhtethtetswan@users.noreply.github.com>
Date:   Wed Mar 17 23:12:57 2021 -0400

    Delete ast.ml

commit 249143fc98ac993543edf41785dca8075c4ce78b
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 16 20:47:50 2021 -0400

    added bitwise and, or, negation to scanner

commit eb3384e367999348767df81003cd5bf872ce9229
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 16 20:30:03 2021 -0400

    members and methods tokens

commit e98933745fa7cb62999c324ced2b998ae704066c
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Tue Mar 16 19:59:49 2021 -0400

    Removed test file

commit 65ca9c04374e02d6df74e5ea939eaf5eb5dd4fc2
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Tue Mar 16 19:56:41 2021 -0400

    Added string, float and character literals to the scanner.mll file

commit f0d2c7edbe736b714645bee4e853cb45ab08da25
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sat Mar 13 15:34:09 2021 -0500

    working on bringing scanner up to parser

commit 69f8e58f5ae05a553e528be484cf1289dcedbe4a
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sat Mar 13 15:26:26 2021 -0500

    Fixed default issue in switch statement

commit acd0797a3c116ca57d846d69ae1678c1df7af31a
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Feb 23 22:02:31 2021 -0500
```

```
    added switch and case

commit e699217f22c4ee53021cf9268a21f1bea86ee6a6
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Feb 23 21:03:15 2021 -0500

    Cleaned up language grammar

commit 142c2c61b80422a4a0a358ecddc3d75993039d14
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Feb 23 00:02:01 2021 -0500

    Cleaned up structure

commit 56d6ddd0f200555de790f1959fc5b6eceded528b
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Feb 22 23:24:53 2021 -0500

    Parser and Grammar almost done

commit a763036730302fd231dcb54f86e6a96d446f6bf5
Author: Adam Fowler <680993+usddddd@users.noreply.github.com>
Date:   Sun Jan 24 16:41:01 2021 -0500

    Initial commit

commit 6a15ed53317d723f16f870a782bd36231b6976a0
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Apr 26 21:38:26 2021 -0400

    added compilation and testing instructions to Readme

commit 98a9407d24c13af2f9160e243c8c8c6a7f1f37f6
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Apr 26 21:37:26 2021 -0400

    updated docker start, stop, build scripts

commit 16166547e3f9389909de1c8a2d5f3359d3a6f315
Merge: 101654e d8cd9f9
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Mon Apr 26 21:30:14 2021 -0400

    Remove microc

commit 101654e855a2d3947e0f9beb564a2b6c8b632d55
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Mon Apr 26 21:27:35 2021 -0400

    Finishing up

commit d8cd9f9dcc59f4cf9691eb6ff518a12122b0d1b2
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 20:38:00 2021 -0400

    Removed microc
```

```
commit e04b6ca1c9babad864e68c5f5857aeeea762fb1e
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 09:41:24 2021 -0400

    Use .files for test

commit 801c6fca3a15235cf1a9b188304f047becc79b34
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 26 09:36:35 2021 -0400

    Renamed microc to gassp

commit 96cd6eed0aa327d06942c778723606c08c351290
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 18 20:31:50 2021 -0400

    Re-added functions to semant

commit 5ce5e8c84987f60b49b748f53ee4e5fd1b25c1e3
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Thu Apr 15 20:04:36 2021 -0400

    Changed test file ext

commit 748ff0c0e15382c3d4301d2d940ed67a17f67f02
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Thu Apr 15 19:46:56 2021 -0400

    Use .g extension for test source files

commit e74d44d8aef8401988356b4942f2c4e4f4e6236c
Merge: df0a739 49361b3
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Thu Apr 15 19:30:32 2021 -0400

    Merged swan's function call work into break/continue branch

commit 49361b336d3e51283da04f64ba5a930a29c2e8b7
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Thu Apr 15 19:02:41 2021 -0400

    Renamed to match develop

commit 8cd71445d2cbc40d2462f59a5691eeb22f742fff
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Thu Apr 15 18:56:09 2021 -0400

    added gitignore and removed unused check function

commit 2fd90285f7e1b4e5e9e3a99771378628c9e41bf6
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Wed Apr 14 23:32:07 2021 -0400

    Removed non-working grammar constructs for now

commit 23802ba60c39e98f8c88eb003e0ad1c41acb69b6
```

```
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Wed Apr 14 22:09:28 2021 -0400

    Check types for bind with value

commit df0a739a54641d117f44904b6eed44e1b7bb06e4
Author: swanhtethtetswan <ucancallmex@gmail.com>
Date:   Wed Apr 14 14:44:57 2021 -0400

    makefile withc PHONY and math and string funs

commit b89ff7544b53f18308c0b3173ef1c35c080928a7
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Apr 12 20:37:37 2021 -0400

    Removed switch and introduced bind with value

commit 6672d6a33e149114b7e792eb0c1d2badf4850136
Author: swanhtethtetswan <ucancallmex@gmail.com>
Date:   Mon Apr 12 19:52:12 2021 -0400

    min and max together

commit bc43eb89fd5862ee8e4f519c09167cebce95ce88
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Apr 11 12:46:04 2021 -0400

    integrating pati's changes

commit 70401653156258987be1405e5c4792a1726f5b06
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Wed Apr 7 22:48:37 2021 -0400

    break and continue loop

commit 2e5c0a10f32d160cddbabea5b66aef32919bee81
Author: swanhtethtetswan <ucancallmex@gmail.com>
Date:   Wed Apr 7 21:19:16 2021 -0400

    min and max

commit 8f0c21c541ff849bea84571788ab33b29b08a1a3
Author: swanhtethtetswan <ucancallmex@gmail.com>
Date:   Wed Apr 7 18:26:15 2021 -0400

    changed gasp to gassp and added min function

commit b7994f46b47b0c31d14d1919cecdc44cf23c90dd
Author: swanhtethtetswan <ucancallmex@gmail.com>
Date:   Wed Apr 7 18:23:47 2021 -0400

    new directory gassp

commit 638f398ef43e7b45495b038a2fa9112dc09c4c74
Author: swanhtethtetswan <ucancallmex@gmail.com>
Date:   Wed Apr 7 18:22:50 2021 -0400
```

```
    removed old stuff

commit ed8b01f2bdb0e37485d6342109c32e424f84bdf1
Merge: 75abbe5 580d094
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Mar 29 22:48:46 2021 -0400

    Merge branch 'develop' of github.com:usddddd/PLT into develop

commit 75abbe517506fae4fcf31047f18c1e6bebb50804
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Mar 29 22:27:18 2021 -0400

    removed built compiler from tracked files

commit 580d094d4d71588c9dcd4b4147d1e25160b0f695
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Mar 29 22:27:18 2021 -0400

    removed built compiler

commit 4143528f2ea2adbc51fbfe321870a7a25f17bde5
Merge: f5020aa 14f3339
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Mar 29 22:22:14 2021 -0400

    Merge branch 'feat/helloworld-building' into develop

commit 14f33390fb9b4389b2232b970df8ce04c22bba2d
Merge: 63fb08c fc59665
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Mar 29 22:03:45 2021 -0400

    Merge branch 'feat/helloworld-building' of github.com:usddddd/PLT into
feat/helloworld-building

commit 63fb08c9f14e3f60c14aa77e9c509f8f8fbd906d
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Mon Mar 29 22:03:39 2021 -0400

    expanded Makefile's clean command

commit f5020aa75219beb1d68d121977736586b953ed1a
Author: Yuanxin-Yang <55468947+Yuanxin-Yang@users.noreply.github.com>
Date:   Mon Mar 29 21:16:02 2021 -0400

    updated array dec

commit fc5966555a45e053982fff1e507b8081df93c6d3
Author: Yuanxin-Yang <55468947+Yuanxin-Yang@users.noreply.github.com>
Date:   Mon Mar 29 21:10:41 2021 -0400

    updated array dec

commit 9922040ab46bdda892583a9b4baa901a5e25ea1f
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sun Mar 28 15:57:00 2021 -0400
```

```
    Switch should be a statement not expression

commit 098fc1c88672dbfb74c1a0c2e21fa61f6de64b6b
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Wed Mar 24 17:28:14 2021 -0400

    Added cases for pattern matching

commit 5e1b060a17c367feb9a144cef8572b49f7fe5e4c
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Wed Mar 24 16:53:42 2021 -0400

    Added character printing and made minor changes in scanner

commit 1c41e5b5208f94c26224c82f0ae18c57d533532b
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Wed Mar 24 02:46:22 2021 -0400

    updated Makefile2 to help build hello world file

commit d30ec87bac68999edea183b66660ff01ec28dcc8
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Wed Mar 24 02:40:04 2021 -0400

    basic helloworld file

commit 34139de73d3057084b4d9a991f55f40b618f99e5
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Wed Mar 24 02:17:09 2021 -0400

    swan's modifications to add string printing

commit 02adeee3a68c913b26feadaacc57bb867ab788fb
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 23 23:35:00 2021 -0400

    added microc

commit c45b4247c68c095cc9ba59706f9d30923c436c0a
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Mar 23 22:10:50 2021 -0400

    Builds with warnings now

commit 921626e574439d164dcf9696829f4567479f11d8
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Mar 23 20:33:18 2021 -0400

    Add string to class constructor

commit e3a27a02c6539803e8575baad892edcaa7df2022
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Mar 23 20:07:59 2021 -0400

    added missing stuff
```

```
commit 2f000085d8ef6ece207dff63b020e961d1f02f92
Merge: fecd0a4 2fb9846
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Mar 23 20:01:42 2021 -0400

    Merge branch 'develop' of github.com:usddddd/PLT into develop

commit fecd0a4208dad04782c782c8d4c027f5886dfb72
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Mar 23 19:38:45 2021 -0400

    Fixing make errors

commit 2fb9846a719aa0db9f7cb9aa77c64468b3c844fe
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Mar 23 19:38:45 2021 -0400

    Fixing make errors

commit bb2b40264b1420c7e8bd05d22e224a076eb0e493
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 23 18:18:56 2021 -0400

    added char, string, and object to string_of_typ function

commit 7c5b3b64d74711ff7a1f90fc90279a2fc703841c
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 23 18:16:04 2021 -0400

    basic Makefile

commit cc11c7c0eb2ae7a57d45083cfa3dfb6019ddd544
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 23 18:06:34 2021 -0400

    moved _tags to gasp directory

commit cfe996697eb3e5bf4b71401b7349e5c3bd4dc0d9
Merge: 8308a28 0b4f543
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 23 18:00:55 2021 -0400

    Merge branch 'develop' of github.com:usddddd/PLT into develop

commit 8308a28d23e1ce8ced68573902819f53efba4523
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 23 18:00:32 2021 -0400

    fixed ast filename

commit 9a84ac3e03b37f4e91f12682ef565da2b849b906
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 23 17:59:19 2021 -0400

    added a gassp.ml adapted from microc.ml

commit 0b4f543311386d057edb243db7e4b0a73d907b58
```

```
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Tue Mar 23 17:57:15 2021 -0400

    Added strings and chars to codegen.ml and semant.ml

commit 37aea8c782043f141f5e23fc2c00fc153d966387
Merge: d434e2b ed96a14
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Mar 22 20:10:24 2021 -0400

    Merge branch 'ajf2177/parser' into develop

commit ed96a14c43e87fefa8f1de97ce2cf6f08e7ef5ce
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Mar 22 20:05:34 2021 -0400

    Re-structured project to match microc for building with Makefile
(eventually)

commit 97af73769d8b4491c57754ec3539ab04f337cf9c
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Sun Mar 21 20:11:11 2021 -0400

    Pushing corrected sast.ml and scanner.mll

commit 6e6194cc2e0fb6e930c4640544a05b4186dcddc4
Merge: 87fce8a 278f2c6
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Sun Mar 21 18:43:58 2021 -0400

    Merge branch 'ajf2177/parser' of github.com:usddddd/PLT into ajf2177/parser

commit 87fce8a7977f37d5e6f0cb1f7b0dce501a189769
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Sun Mar 21 18:43:03 2021 -0400

    Created sast file

commit d434e2b40dd8d4eeb4b31bb37f029d15473ffc69
Merge: 0d68216 f647553
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sat Mar 20 23:32:02 2021 -0400

    Added docker build/start/stop scripts

commit 0d68216acf995fa570ea93f95a6090b4386b72bc
Author: Adam Fowler <afowler@peerstream.com>
Date:   Sat Mar 20 23:23:36 2021 -0400

    Fixed Dockerfile

commit 278f2c6728cdcd40c8b9266f880708b36de16678
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Sat Mar 20 16:33:00 2021 -0400

    finished remaining lrm sections
```

```
commit dceeeab7d785048fb85ab39c4b686d9fb3207de0
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Sat Mar 20 14:29:18 2021 -0400

    added titles to links

commit 5b53de2e21b93f640c4c4ee3621da077ba8470fd
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Sat Mar 20 14:14:17 2021 -0400

    finished writing all subsection headings, confirmed all toc linkings works

commit 37bd4cc44de0fc9656920c67d743e22698c202fd
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Sat Mar 20 14:05:39 2021 -0400

    section 3 completed

commit f3c2fe2b85a6452254dba5f0533c0a77826535c4
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Sat Mar 20 13:53:33 2021 -0400

    completed section 2

commit 3f6c1904789359572e827c5e7880c4f3e4dfc580
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Sat Mar 20 13:27:29 2021 -0400

    built and linked toc

commit ce760bc01d55d1245824311906722bd0921b81c9
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Thu Mar 18 21:01:08 2021 -0400

    did lrm skeleton and sections 1 to 2.4.1

commit c453ea3634d0d4bdc3686a94340b9285be7cf444
Author: Adam Fowler <afowler@peerstream.com>
Date:   Thu Mar 18 19:46:10 2021 -0400

    Added Dockerfile and moved project files

commit 6ff5b8c125136964f1e010ed6d63b7be24d4a45a
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Thu Mar 18 15:59:51 2021 -0400

    added bitwise xor, shifts, and exponent tokens

commit f647553acf08e5141c4702537d3b3377de0ec78b
Author: swanhtethtetswan <23371162+swanhtethtetswan@users.noreply.github.com>
Date:   Wed Mar 17 23:37:28 2021 -0400

    Create gast.ml

commit 0deb45e17be96c8f3e734c992cf4db219bb66075
Author: swanhtethtetswan <23371162+swanhtethtetswan@users.noreply.github.com>
Date:   Wed Mar 17 23:12:57 2021 -0400
```

```
    Delete ast.ml

commit 249143fc98ac993543edf41785dca8075c4ce78b
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 16 20:47:50 2021 -0400

    added bitwise and, or, negation to scanner

commit eb3384e367999348767df81003cd5bf872ce9229
Author: Samuel Weissmann <38514250+SPWeissmann@users.noreply.github.com>
Date:   Tue Mar 16 20:30:03 2021 -0400

    members and methods tokens

commit e98933745fa7cb62999c324ced2b998ae704066c
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Tue Mar 16 19:59:49 2021 -0400

    Removed test file

commit 65ca9c04374e02d6df74e5ea939eaf5eb5dd4fc2
Author: Patrycja Przewoznik <patrycja.przewoznik@columbia.edu>
Date:   Tue Mar 16 19:56:41 2021 -0400

    Added string, float and character literals to the scanner.mll file

commit f0d2c7edbe736b714645bee4e853cb45ab08da25
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sat Mar 13 15:34:09 2021 -0500

    working on bringing scanner up to parser

commit 69f8e58f5ae05a553e528be484cf1289dcedbe4a
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Sat Mar 13 15:26:26 2021 -0500

    Fixed default issue in switch statement

commit acd0797a3c116ca57d846d69ae1678c1df7af31a
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Feb 23 22:02:31 2021 -0500

    added switch and case

commit e699217f22c4ee53021cf9268a21f1bea86ee6a6
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Feb 23 21:03:15 2021 -0500

    Cleaned up language grammar

commit 142c2c61b80422a4a0a358ecddc3d75993039d14
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Tue Feb 23 00:02:01 2021 -0500

    Cleaned up structure
```

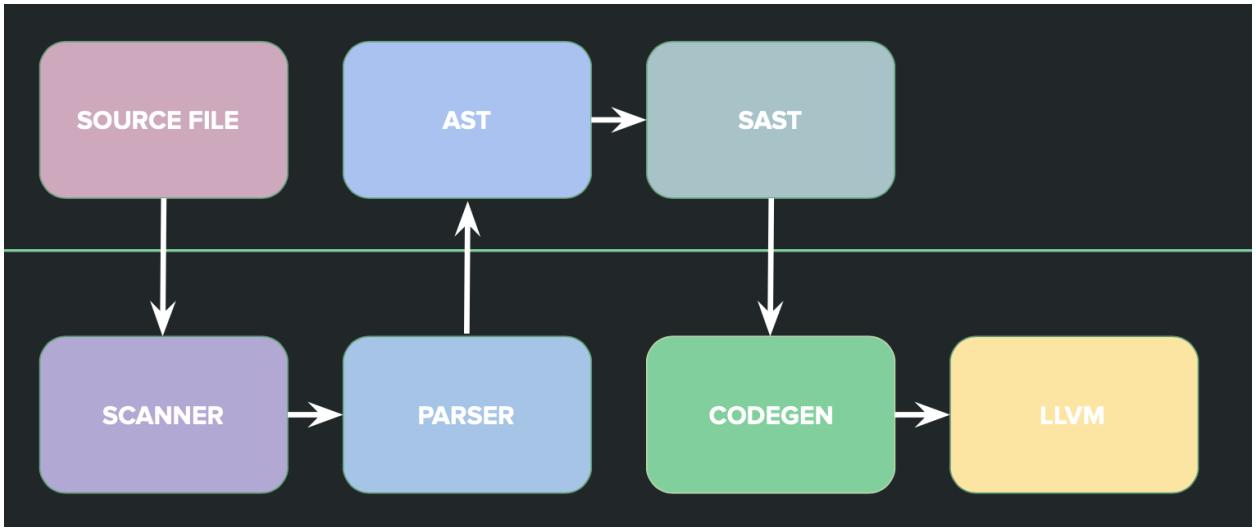
```
commit 56d6ddd0f200555de790f1959fc5b6eceded528b
Author: Adam Fowler <ajf2177@columbia.edu>
Date:   Mon Feb 22 23:24:53 2021 -0500

    Parser and Grammar almost done

commit a763036730302fd231dcb54f86e6a96d446f6bf5
Author: Adam Fowler <680993+usdddddd@users.noreply.github.com>
Date:   Sun Jan 24 16:41:01 2021 -0500

    Initial commit
```

9. Architectural Design



We borrowed the architecture mainly from MicroC.
There should be a source file being fed into the scanner to scan the token. The parser then parses the tokens, and an AST (abstract syntax tree) is built. The semant.ml file takes the ast, checks the semantic and generates a SAST (semantically checked abstract syntax tree), which basically binds the expressions and their types. It is then passed into codegen.ml and it will spit out llvm IR. The llvm IR then goes into llvm and generates an executable.

10. Test Plan

```
int main () {
        if (true) {}
        If (false) {} else {}
        if  (42) {}  /* error: non-bool predicate */
}
```

This test case was chosen to show that similar to MicroC that GASSP supports only boolean predicates.

```
int main () {
	print(max(9,2));
	print(min(2,9));
	print(strlen(“gassp”));
	prints(str_concat(“GASSP is awesome. ”, “Please give us an A :)”));
	i	}f (strcmp(“GASSP”, “Pati”)) {
		prints(“1”);

	else {
		prints(“0”);
	}
	printf(sqrt(4.0));
	printf(rand());
}
```

This test case was chosen to show: 1) GASSP is able to scan, parse, and interpret a source file. 2) The source file shows different built-in functions of GASSP.

We also borrowed some test cases from MicroC. We used testall.sh to run all test cases, which will generate results of comparing generated results with the expected ones. If they match, no error messages will be shown. If they do not match, there would be an error message.

Our group worked on writing some test cases together. To be more specific, Pati and Adam took care of adopting the MicroC test cases. Pati, Swan, and I worked on other ones.

11. Lessons Learned

Swan: I learned how the parser and scanner work together to compile a program. The coolest thing I learned in this class was that we can actually create an oracle that tells the compiler whether to shift or reduce when compiling. I may not remember the details of the intricate steps in a few years but I will be comfortable knowing that I know how the compiler works and I can always brush up my knowledge if I need to know how a compiler for a specific language works. I also learned about the grammars and syntax trees which I thought will be only for the theoretical part of my CS journey but I actually had to use them to generate an actual programming language. I also learned about lambda calculus which I wished was introduced before OCaml. Understanding lambda calculus makes everything in OCaml clicks. And of course, I learned OCaml, my very first functional programming language which certainly will not be my last.

Sam: I learned a lot about Ocaml, functional programming, and what goes into creating a programming language. I feel like I have a much deeper understanding of what is going on “under the hood”, so to say, and feel comfortable with how source code goes from being text to an executable. However, I think the most important lessons were the importance of breaking larger tasks into smaller tasks, assigning tasks/roles, and proactively checking in with everyone to make sure things are being accomplished and people aren’t getting left behind. For projects

and teams of this size, communication and project management is paramount and needs to be continuously enforced.

Adam: I learned the importance of breaking a large project into smaller tasks to distribute amongst team members and also how important it is to coordinate amongst the group who is doing what so as to not duplicate work. I also learned that the best way to approach a project of this size is to work iteratively and get something working/building before starting work on larger features. I also learned it is imperative to check in regularly and make sure everyone is comfortable with their tasks and not lost and feeling hopeless. I also learned the hardest part of a project like this is coordinating effort amongst team members.

Pati: I learned that the communication within the team is staggeringly important to be able to complete a project of this size. In case of poor communication within the team, the project halts and it sets back other members of the team as well. I also learned that it is really important to go to office hours and ask questions if lost since it saves some time. The best approach to the project in the beginning and throughout the project in general is to break up tasks in terms of features rather than specific files so that everybody gets a good understanding of how they are related from the very start. I think overall, I got a good grasp of the structure of the compiler and features of the language while working on the project.

Yuanxin: First and most importantly, I learned a lot about programming languages and compilers. I have little knowledge about compilers before taking the class, and it amazes me how precise, detailed, and clearly-crafted a compiler should be. When working on the project, I learned that communication and constant checkups are extremely important and essential to have a high performing team. There were times that during implementation, more than one group members worked on the same materials, which turned out to create code that both overlapped and contradicted at the same time. I also found that most of our team members had a tendency of only working on their own branch. I understand that work on one's own code is more consistent, but it also makes it hard as a group to collaborate. This is something I found myself having a tendency doing, and I should learn to work collaboratively.

12. Appendix

We have two primary versions of our code. The first is a working version that builds, can compile programs, and supports many of our langauge's features, but does not support classes and objects. The second is code that supports classes and objects and can build, but isn't fully functional and doesn't reliably compile programs. We've included it to demonstrate the continued work that we put into our project towards the goals outlined in our proposal and LRM.

12.1 Gassp without objects

12.1.1 Scanner

```
{ open Gasspparser }

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

```
let letter = [ '_' 'a' - 'z' 'A' - 'Z']
let variable = letter['a'-'z' 'A'-'Z' '0'-'9' '_']*
let whitespace = [' ' '\t' '\r' '\n']+
let float_literal = digit+ '.' digit*
         | digit* '.' digit+
         | digit+ ('.')? digit* ['e' 'E'] ['+' '-']? digit+
         | digit* ('.')? digit+ ['e' 'E'] ['+' '-']? digit+
let ascii = [' '-'!' '#'-'~']
let char_literal = ''' (ascii as chlit) '''
let string_literal = '"' (ascii* as slit) '"'


rule token = parse
 [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LBRACKET }
| ']'      { RBRACKET }
| ';'      { SEMI }
| ':'      { COLON }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '%'      { MOD }
| '/'      { DIVIDE }
| "**"     { EXP }
| '.'      { DOT }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "~"      { BNOT }
```

```
| "|"      { BOR }
| "^"      { BXOR }
| "&"      { BAND }
|"<<"      { LSHIFT }
|">>"      { RSHIFT }
| "++"     { INCR}
| "--"     { DECR }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "char"   { CHAR }
| "string" { STRING }
| "bool"   { BOOL }
| "float"  { FLOAT }
| "void"   { VOID }
| "true"   { BLIT(true)  }
| "false"  { BLIT(false) }
| "switch" {SWITCH}
| "case" {CASE}
| "default" {DEFAULT}
| "continue" {CONTINUE}
| "break" {BREAK}
| digits as lxm { LITERAL(int_of_string lxm) }
| "class" { CLASS}
| char_literal         { CHLIT(chlit) }
| float_literal as lxm { FLIT(lxm) }
| string_literal       { SLIT(slit) }
| variable     as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

12.1.2 Parser

```
/* Ocamlyacc parser for GA$$P*/
```

```
%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA PLUS MINUS TIMES DIVIDE ASSIGN DOT
LBRACKET RBRACKET MOD INCR DECR BREAK CONTINUE SWITCH CASE DEFAULT COLON EXP
%token NOT EQ NEQ LT LEQ GT GEQ AND OR BNOT BOR BXOR BAND LSHIFT RSHIFT
%token RETURN IF ELSE FOR WHILE INT BOOL CHAR FLOAT VOID STRING
%token <int> LITERAL
%token <bool> BLIT
%token <string> SLIT
%token <char> CHLIT
%token <string> ID FLIT
%token CLASS
%token EOF

%start program
%type <Ast.program> program

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

%%

program:
 decls EOF { $1 }

decls:
  /* nothing */ { ([], [])               }
| decls vdecl { (($2 :: fst $1), snd $1) }
| decls fdecl { (fst $1, ($2 :: snd $1)) }
/* | decls cdecl { (fst $1, ($2 :: snd $1)) } */

fdecl:
```

```
  typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
    { { ftyp = $1;
  fname = $2;
  fformals = List.rev $4;
  flocals = List.rev $7;
  fbody = List.rev $8 } }

/* class body */
cbody:
  mdecl_list  { (fst $1 , snd $1)} /* fst is variable list snd is method list*/

/* member declaration list for class members and methods*/
mdecl_list:
 /* nothing */ {([], [])}
 | mdecl_list local_decl{(($2 :: fst $1), snd $1)}
 | mdecl_list method_decl {(fst $1, ($2 :: snd $1))}

local_decl:
 | typ ID SEMI{ ($1, $2) } /* member variable with value*/
 /* | typ ID ASSIGN expr SEMI{ ($1, $2, $4) } member variable with value */
 /* | arr_decl ID SEMI  {($1, $2, Noexpr)}member arr variable without value */


/* member declaration members and methods*/
method_decl:
 | typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE { { mtyp = $1;
                                                                          mname = $2;
                                                                          mformals =
List.rev $4;
                                                                          mlocals =
List.rev $7;
                                                                          mbody =
List.rev $8 }
                                                                          }


cdecl:
   CLASS ID LBRACE cbody RBRACE {{cname = $2; clocals = fst $4; cbody = snd $4}}
   /* CLASS ID LBRACE cbody RBRACE {()} */

formals_opt:
   /* nothing */ { [] }
```

```
 | formal_list   { $1 }

formal_list:
   typ ID                   { [($1,$2)]     }
 | formal_list COMMA typ ID { ($3,$4) :: $1 }
arr_decl:
   typ LBRACKET LITERAL RBRACKET {$1}

typ:
   INT   { Int   }
 | CHAR  { Char  }
 | STRING{ String}
 | BOOL  { Bool  }
 | FLOAT { Float }
 | VOID  { Void  }


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

vdecl:
  typ ID SEMI { ($1, $2) }
  | typ ID ASSIGN expr SEMI {($1, $2)}
  | arr_decl ID SEMI  {($1, $2)}

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

stmt:
   expr SEMI                               { Expr $1               }
 | RETURN expr_opt 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_opt SEMI expr SEMI expr_opt RPAREN stmt
                                           { For($3, $5, $7, $9)   }
 | WHILE LPAREN expr RPAREN stmt           { While($3, $5)         }
 | BREAK SEMI                                 {Break}
 | CONTINUE SEMI                              {Continue}
```

```
expr_opt:
   /* nothing */ { Noexpr }
 | expr          { $1 }

expr:
   LITERAL          { Literal($1)            }
 | CHLIT            { Chliteral($1)              }
 | SLIT             { Sliteral($1)               }
 | FLIT             { Fliteral($1)           }
 | BLIT             { BoolLit($1)            }
 | ID               { Id($1)                 }
 | expr PLUS   expr { Binop($1, Add,   $3)   }
 | expr MOD    expr { Binop($1, Mod,   $3)   }
 | expr MINUS  expr { Binop($1, Sub,   $3)   }
 | expr TIMES  expr { Binop($1, Mult,  $3)   }
 | expr DIVIDE expr { Binop($1, Div,   $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 GEQ    expr { Binop($1, Geq,   $3)   }
 | expr BAND    expr { Binop($1, Band,   $3)   }
 | expr BOR     expr { Binop($1, Bor,    $3)   }
 | expr BXOR     expr { Binop($1, Bxor,    $3)   }
 | expr AND    expr { Binop($1, And,   $3)   }
 | expr OR     expr { Binop($1, Or,    $3)   }
 | expr LSHIFT expr { Binop($1, Lshift,   $3)   }
 | expr RSHIFT expr { Binop($1, Rshift,    $3)   }
 | ID DOT ID  LPAREN args_list RPAREN { Mcall($1, $3, $5)}
 | MINUS expr %prec NOT { Unop(Neg, $2)      }
 | expr INCR     { Unop(Incr, $1)      }
 | expr DECR     { Unop(Decr, $1)      }
 | NOT expr         { Unop(Not, $2)          }
 | BNOT expr         { Unop(Bnot, $2)          }
 | ID ASSIGN expr   { Assign($1, $3)         }
 | ID LPAREN args_opt RPAREN { Call($1, $3)  }
 | LPAREN expr RPAREN { $2                   }
lit_expr:
   LITERAL          { Literal($1)            }
 | CHLIT            { Chliteral($1)              }
 | SLIT             { Sliteral($1)               }
```

```
 | FLIT             { Fliteral($1)           }
 | BLIT             { BoolLit($1)            }

args_opt:
   /* nothing */ { [] }
 | args_list  { List.rev $1 }

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

12.1.3 Ast

```
(* Abstract Syntax Tree and functions for printing it *)

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq |
          And | Or | Mod | Member | Exp | Lshift | Rshift | Band | Bxor | Bor

type uop = Neg | Not | Incr | Decr | Bnot

type typ = Int | Bool | Float | Void | Char | String | Object

type bind = typ * string

type expr =
   Literal of int
 | Sliteral of string
 | Chliteral of char
 | Fliteral of string
 | BoolLit of bool
 | Id of string
 | Binop of expr * op * expr
 | Unop of uop * expr
 | Assign of string * expr
 | Call of string * expr list
 | Mcall of string * string * expr list (* method call needs class name, method name,
and args*)
 | Noexpr

type ebind = typ * string * expr

type stmt =
```

```
   Block of stmt list
 | Expr of expr
 | Return of expr
 | If of expr * stmt * stmt
 | For of expr * expr * expr * stmt
 | While of expr * stmt
 | Break
 | Continue



type func_decl = {
 ftyp : typ;
 fname : string;
 fformals : bind list;
 flocals : bind list;
 fbody : stmt list;
}

type method_decl = {
 mtyp : typ;
 mname : string;
 mformals : bind list;
 mlocals : bind list;
 mbody : stmt list;
}

type class_decl = {
 cname : string;
 clocals : bind list;
 cbody : method_decl list;
}

type arr_decl = {
 arrtyp : typ;
 size: int;
}

type program = bind list * func_decl list
(*type program = bind list * func_decl list * class_decl list *)
```

```
(* Pretty-printing functions *)

let string_of_op = function
   Add -> "+"
 | Sub -> "-"
 | Mult -> "*"
 | Div -> "/"
 | Equal -> "=="
 | Neq -> "!="
 | Less -> "<"
 | Leq -> "<="
 | Greater -> ">"
 | Geq -> ">="
 | And -> "&&"
 | Or -> "||"
 | Mod -> "%"
 | Member -> "."
 | Exp -> "**"
 | Band -> "&"
 | Bor -> "|"
 | Bxor -> "^"
 | Lshift -> "<<"
 | Rshift -> ">>"

let string_of_uop = function
   Neg -> "-"
 | Decr -> "--"
 | Incr -> "++"
 | Not -> "!"
 | Bnot -> "!"

let string_of_char c = String.make 1 c

let rec string_of_expr = function
   Literal(l) -> string_of_int l
 | Fliteral(l) -> l
 | BoolLit(true) -> "true"
 | BoolLit(false) -> "false"
 | Sliteral(s) -> s
 | Chliteral(c) -> string_of_char c
 | Id(s) -> s
 | Binop(e1, o, e2) ->
```

```
    string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Mcall(c, m, el) ->
    c ^ "." ^ m ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Call(f, el) ->
    f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) ->
    "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
                      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
    "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
    string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | Break -> "break;"
  | Continue -> "continue;"

let string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Float -> "float"
  | Void -> "void"
  | Char -> "char"
  | String -> "string"
  | Object -> "object"

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

let string_of_edecl (t, id, e) = string_of_typ t ^ " " ^ id ^ " = " ^ string_of_expr e
^ ";\n"

let string_of_fdecl fdecl =
 string_of_typ fdecl.ftyp ^ " " ^
 fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.fformals) ^
 ")\n{\n" ^
```

```
 String.concat "" (List.map string_of_vdecl fdecl.flocals) ^
 String.concat "" (List.map string_of_stmt fdecl.fbody) ^
 "}\n"

let string_of_program (vars, funcs) =
 String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
 String.concat "\n" (List.map string_of_fdecl funcs)
```

12.1.4 Sast

```
(* Semantically-checked Abstract Syntax Tree and functions for printing it *)

open Ast

type sexpr = typ * sx
and sx =
   SLiteral of int
 | SFliteral of string
 | SSliteral of string
 | SChliteral of char
 | SBoolLit of bool
 | SId of string
 | SBinop of sexpr * op * sexpr
 | SUnop of uop * sexpr
 | SAssign of string * sexpr
 | SCall of string * sexpr list
 | SNoexpr

type sstmt =
   SBlock of sstmt list
 | SExpr of sexpr
 | SReturn of sexpr
 | SIf of sexpr * sstmt * sstmt
 | SFor of sexpr * sexpr * sexpr * sstmt
 | SWhile of sexpr * sstmt
 | SBreak
 | SContinue

type sfunc_decl = {
   styp : typ;
   sfname : string;
   sformals : bind list;
```

```
   slocals : bind list;
   sbody : sstmt list;
 }

type smethod_decl = {
   smtyp : typ;
   smname : string;
   smformals : bind list;
   smlocals : bind list;
   smbody : sstmt list;
}

type sclass_decl = {
   scname : string;
   sclocals : bind list;
   scbody : smethod_decl list;
}

type sarr_decl = {
 sarrtyp : typ;
 ssize: int;
}

type sprogram = bind list * sfunc_decl list

(* Pretty-printing functions *)

let rec string_of_sexpr (t, e) =
 "(" ^ string_of_typ t ^ " : " ^ (match e with
   SLiteral(l) -> string_of_int l
 | SBoolLit(true) -> "true"
 | SBoolLit(false) -> "false"
 | SSliteral(s) -> s
 | SChliteral(c) -> string_of_char c
 | SFliteral(l) -> l
 | SId(s) -> s
 | SBinop(e1, o, e2) ->
     string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
 | SUnop(o, e) -> string_of_uop o ^ string_of_sexpr e
 | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
 | SCall(f, el) ->
     f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
```

```
  | SNoexpr -> ""
        ) ^ ")"

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

let string_of_sfdecl fdecl =
  string_of_typ fdecl.styp ^ " " ^
  fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd fdecl.sformals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.slocals) ^
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
  "}\n"

let string_of_sprogram (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_sfdecl funcs)
```

12.1.5 Semant

```
open Ast
open Sast

module StringMap = Map.Make(String)

(* Semantic checking of the AST. Returns an SAST if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)
```

```
let check (globals, functions) =

 (* Verify a list of bindings has no void types or duplicate names *)
 let check_binds (kind : string) (binds : bind list) =
   List.iter (function
         (Void, b) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
       | _ -> ()) binds;
   let rec dups = function
       [] -> ()
     | ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
       raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
     | _ :: t -> dups t
   in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
 in

   (**** Check global variables ****)
   check_binds "global" globals;

 (**** Check functions ****)
 (* Collect function declarations for built-in functions: no bodies *)
 let built_in_decls =
   let add_bind map (name, ty) = StringMap.add name {
       ftyp = Void;
       fname = name;
       fformals = [(ty, "x")];
       flocals = []; fbody = [] } map
   in List.fold_left add_bind StringMap.empty [ ("print", Int);
                                                ("printb", Bool);
                                                ("printf", Float);
                                                ("printbig", Int);
                                                ("prints", String);
                                                ("printc", Char) ]
 in
 let built_in_decls =
   let add_bind map (name, ty1, ty2, fty) = StringMap.add name {
       ftyp = fty;
       fname = name;
       fformals = [(ty1, "x"); (ty2, "x")];
       flocals = []; fbody = [] } map
   in List.fold_left add_bind built_in_decls [ ("min", Int, Int, Int); ("max", Int,
Int, Int);
```

```
                                                ("strcpy", String, String, String);
("strcat", String, String, String);
                                                ("strcmp", String, String, Bool);
("strstr", String, String, String);
                                                ("str_concat", String, String, String)]
 in
  let built_in_decls =
   let add_bind map (name, ty, fty) = StringMap.add name {
       ftyp = fty;
       fname = name;
       fformals = [(ty, "x")];
       flocals = []; fbody = [] } map
   in List.fold_left add_bind built_in_decls [ ("sin", Float, Float); ("cos", Float,
Float); ("tan", Float, Float);
                                                ("sinh", Float, Float); ("cosh", Float,
Float); ("tanh", Float, Float);
                                                ("exp", Float, Float); ("log", Float,
Float); ("sqrt", Float, Float);
                                                ("strlen", String, Int) ]
 in


 let built_in_decls =
        let add_bind map (name, fty) = StringMap.add name {
        ftyp = fty;
        fname = name;
        fformals = [];
        flocals = []; fbody = [] } map
   in List.fold_left add_bind built_in_decls [ ("rand", Float)]
  in
 (* Add function name to symbol table *)
 let add_func map fd =
   let built_in_err = "function " ^ fd.fname ^ " may not be defined"
   and dup_err = "duplicate function " ^ fd.fname
   and make_err er = raise (Failure er)
   and n = fd.fname (* Name of the function *)
   in match fd with (* No duplicate functions or redefinitions of built-ins *)
     _ when StringMap.mem n built_in_decls -> make_err built_in_err
   | _ when StringMap.mem n map -> make_err dup_err
   | _ ->  StringMap.add n fd map
 in
```

```
(* Collect all function names into one symbol table *)
let function_decls = List.fold_left add_func built_in_decls functions
in

(* Return a function from our symbol table *)
let find_func s =
  try StringMap.find s function_decls
  with Not_found -> raise (Failure ("unrecognized function " ^ s))
in

let _ = find_func "main" in (* Ensure "main" is defined *)

let check_function func =
  (* Make sure no formals or locals are void or duplicates *)
  check_binds "formal" func.fformals;
  check_binds "local" func.flocals;

  (* Raise an exception if the given rvalue type cannot be assigned to
     the given lvalue type *)
  let check_assign lvaluet rvaluet err =
    if lvaluet = rvaluet then lvaluet else raise (Failure err)
  in

  (* Build local symbol table of variables for this function *)
  let symbols = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
      StringMap.empty (globals @ func.fformals @ func.flocals )
  in

  (* Return a variable from our local symbol table *)
  let type_of_identifier s =
    try StringMap.find s symbols
    with Not_found -> raise (Failure ("undeclared identifier " ^ s))
  in

  (* Return a semantically-checked expression, i.e., with a type *)
  let rec expr = function
      Literal  l -> (Int, SLiteral l)
    | Fliteral l -> (Float, SFliteral l)
    | BoolLit l  -> (Bool, SBoolLit l)
    | Chliteral l -> (Char, SChliteral l)
    | Sliteral l -> (String, SSliteral l)
    | Mcall(c, f, args) -> ignore(c); ignore(f); ignore(args); (Void, SNoexpr)
```

```
| Noexpr     -> (Void, SNoexpr)
| Id s       -> (type_of_identifier s, SId s)
| Assign(var, e) as ex ->
  let lt = type_of_identifier var
  and (rt, e') = expr e in
  let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
            string_of_typ rt ^ " in " ^ string_of_expr ex
  in (check_assign lt rt err, SAssign(var, (rt, e')))
| Unop(op, e) as ex ->
  let (t, e') = expr e in
  let ty = match op with
      Neg when t = Int || t = Float -> t
    | Not when t = Bool -> Bool
    | _ -> raise (Failure ("illegal unary operator " ^
                           string_of_uop op ^ string_of_typ t ^
                           " in " ^ string_of_expr ex))
  in (ty, SUnop(op, (t, e')))
| Binop(e1, op, e2) as e ->
  let (t1, e1') = expr e1
  and (t2, e2') = expr e2 in
  (* All binary operators require operands of the same type *)
  let same = t1 = t2 in
  (* Determine expression type based on operator and operand types *)
  let ty = match op with
      Add | Sub | Mult | Div when same && t1 = Int   -> Int
    | Add | Sub | Mult | Div when same && t1 = Float -> Float
    | Equal | Neq            when same               -> Bool
    | Less | Leq | Greater | Geq
      when same && (t1 = Int || t1 = Float) -> Bool
    | And | Or when same && t1 = Bool -> Bool
    | _ -> raise (
        Failure ("illegal binary operator " ^
                 string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
                 string_of_typ t2 ^ " in " ^ string_of_expr e))
  in (ty, SBinop((t1, e1'), op, (t2, e2')))
| Call(fname, args) as call ->
  let fd = find_func fname in
  let param_length = List.length fd.fformals in
  if List.length args != param_length then
    raise (Failure ("expecting " ^ string_of_int param_length ^
                    " arguments in " ^ string_of_expr call))
  else let check_call (ft, _) e =
```

```
            let (et, e') = expr e in
            let err = "illegal argument found " ^ string_of_typ et ^
                      " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e
            in (check_assign ft et err, e')
      in
      let args' = List.map2 check_call fd.fformals args
      in (fd.ftyp, SCall(fname, args'))
in

let check_bool_expr e =
  let (t', e') = expr e
  and err = "expected Boolean expression in " ^ string_of_expr e
  in if t' != Bool then raise (Failure err) else (t', e')
in

(* Return a semantically-checked statement i.e. containing sexprs *)
let rec check_stmt = function
    Expr e -> SExpr (expr e)
  | Continue -> SContinue
  | Break -> SBreak
  | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt b2)
  | For(e1, e2, e3, st) ->
    SFor(expr e1, check_bool_expr e2, expr e3, check_stmt st)
  | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
  | Return e -> let (t, e') = expr e in
    if t = func.ftyp then SReturn (t, e')
    else raise (
        Failure ("return gives " ^ string_of_typ t ^ " expected " ^
                 string_of_typ func.ftyp ^ " in " ^ string_of_expr e))

  (* A block is correct if each statement is correct and nothing
             follows any Return statement.  Nested blocks are flattened. *)
  | Block sl ->
    let rec check_stmt_list = function
        [Return _ as s] -> [check_stmt s]
      | Return _ :: _   -> raise (Failure "nothing may follow a return")
      | Block sl :: ss  -> check_stmt_list (sl @ ss) (* Flatten blocks *)
      | s :: ss         -> check_stmt s :: check_stmt_list ss
      | []              -> []
    in SBlock(check_stmt_list sl)

in (* body of check_function *)
```

```
   { styp = func.ftyp;
     sfname = func.fname;
     sformals = func.fformals;
     slocals  = func.flocals;
     sbody = match check_stmt (Block func.fbody) with
         SBlock(sl) -> sl
       | _ -> raise (Failure ("internal error: block didn't become a block?"))
   }
 in (globals, List.map check_function functions)
```

12.1.6 Codegen

```
module L = Llvm
module A = Ast
open Sast

module StringMap = Map.Make(String)

(* translate : Sast.program -> Llvm.module *)
let translate (globals, functions) =
 let context    = L.global_context () in
  (* Create the LLVM compilation module into which
    we will generate code *)
 let the_module = L.create_module context "Gassp" in

 (* Get types from the context *)
 let i32_t      = L.i32_type    context
 and i8_t       = L.i8_type     context
 and i1_t       = L.i1_type     context
 and float_t    = L.double_type context
 and str_t      = L.pointer_type (L.i8_type context)
 and void_t     = L.void_type   context in

 (* Return the LLVM type for a MicroC type *)
 let ltype_of_typ = function
     A.Int   -> i32_t
   | A.Char  -> i8_t
   | A.Bool  -> i1_t
   | A.Float -> float_t
   | A.String-> str_t
   | A.Void  -> void_t
```

```
    | A.Object -> i32_t (* Fix Later *)
  in

  (* Create a map of global variables after creating each *)
  let global_vars : L.llvalue StringMap.t =
    let global_var m (t, n) =
      let init = match t with
          A.Float -> L.const_float (ltype_of_typ t) 0.0
        | _ -> L.const_int (ltype_of_typ t) 0
      in StringMap.add n (L.define_global n init the_module) m in
    List.fold_left global_var StringMap.empty globals in

  let printf_t : L.lltype =
      L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
  let printf_func : L.llvalue =
      L.declare_function "printf" printf_t the_module in

  let printbig_t : L.lltype =
      L.function_type i32_t [| i32_t |] in
  let printbig_func : L.llvalue =
      L.declare_function "printbig" printbig_t the_module in
  let min_t : L.lltype = L.function_type i32_t [| i32_t; i32_t |] in
  let min_func : L.llvalue = L.declare_function "min" min_t the_module in
  let max_t : L.lltype = L.function_type i32_t [| i32_t; i32_t |] in
  let max_func : L.llvalue = L.declare_function "max" max_t the_module in
  (* basic string.h functions *)
  let strlen_t : L.lltype = L.function_type i32_t [| str_t |] in
  let strlen_func : L.llvalue = L.declare_function "strlen" strlen_t the_module in
  let strcpy_t : L.lltype = L.function_type str_t [| str_t; str_t |] in
  let strcpy_func : L.llvalue = L.declare_function "strcpy" strcpy_t the_module in
  let strcat_t : L.lltype = L.function_type str_t [| str_t; str_t |] in
  let strcat_func : L.llvalue = L.declare_function "strcat" strcat_t the_module in
  let strcmp_t : L.lltype = L.function_type i1_t [| str_t; str_t |] in
  let strcmp_func : L.llvalue = L.declare_function "strcmp" strcmp_t the_module in
  let strstr_t : L.lltype = L.function_type str_t [| str_t; str_t |] in
  let strstr_func : L.llvalue = L.declare_function "strstr" strstr_t the_module in
  let str_concat_t : L.lltype = L.function_type str_t [| str_t ; str_t|] in
  let str_concat_f : L.llvalue = L.declare_function "str_concat" str_concat_t
the_module in
  (* basic math.h functions *)
  let sin_t : L.lltype = L.function_type float_t [| float_t |] in
  let sin_func : L.llvalue = L.declare_function "sin" sin_t the_module in
```

```
let cos_t : L.lltype = L.function_type float_t [| float_t |] in
let cos_func : L.llvalue = L.declare_function "cos" cos_t the_module in
let tan_t : L.lltype = L.function_type float_t [| float_t |] in
let tan_func : L.llvalue = L.declare_function "tan" tan_t the_module in
let sinh_t : L.lltype = L.function_type float_t [| float_t |] in
let sinh_func : L.llvalue = L.declare_function "sinh" sinh_t the_module in
let cosh_t : L.lltype = L.function_type float_t [| float_t |] in
let cosh_func : L.llvalue = L.declare_function "cosh" cosh_t the_module in
let tanh_t : L.lltype = L.function_type float_t [| float_t |] in
let tanh_func : L.llvalue = L.declare_function "tanh" tanh_t the_module in
let exp_t : L.lltype = L.function_type float_t [| float_t |] in
let exp_func : L.llvalue = L.declare_function "exp" exp_t the_module in
let log_t : L.lltype = L.function_type float_t [| float_t |] in
let log_func : L.llvalue = L.declare_function "log" log_t the_module in
let sqrt_t : L.lltype = L.function_type float_t [| float_t |] in
let sqrt_func : L.llvalue = L.declare_function "sqrt" sqrt_t the_module in
let rand_t : L.lltype = L.function_type i32_t [|  |] in
let rand_func : L.llvalue = L.declare_function "rand" rand_t the_module in
let printbig_t : L.lltype =
    L.function_type i32_t [| i32_t |] in
let printbig_func : L.llvalue =
    L.declare_function "printbig" printbig_t the_module in


(* Define each function (arguments and return type) so we can
   call it even before we've created its body *)
let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
  let function_decl m fdecl =
    let name = fdecl.sfname
    and formal_types =
Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.sformals)
    in let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types in
    StringMap.add name (L.define_function name ftype the_module, fdecl) m in
  List.fold_left function_decl StringMap.empty functions in
 (* Fill in the body of the given function *)
let build_function_body fdecl =
  let (the_function, _) = StringMap.find fdecl.sfname function_decls in
  let builder = L.builder_at_end context (L.entry_block the_function) in

  let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
  and float_format_str = L.build_global_stringptr "%g\n" "fmt" builder
  and string_format_str = L.build_global_stringptr "%s\n" "fmt" builder
```

```
  and char_format_str = L.build_global_stringptr "%c\n" "fmt" builder
  and bool_format_str = L.build_global_stringptr "%d\n" "fmt" builder in

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

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

    let formals = List.fold_left2 add_formal StringMap.empty fdecl.sformals
        (Array.to_list (L.params the_function)) in
    List.fold_left add_local formals fdecl.slocals
  in

  (* Return the value for a variable or formal argument.
     Check local names first, then global names *)
  let lookup n = try StringMap.find n local_vars
                 with Not_found -> StringMap.find n global_vars
  in

  (* Construct code for an expression; return its value *)
  let rec expr builder ((_, e) : sexpr) = match e with
SLiteral i   -> L.const_int i32_t i
    | SBoolLit b   -> L.const_int i1_t (if b then 1 else 0)
    | SFliteral l  -> L.const_float_of_string float_t l
    | SChliteral c -> L.const_int i8_t (Char.code c)
    | SSliteral s  -> L.build_global_stringptr s "str" builder
    | SNoexpr      -> L.const_int i32_t 0
    | SId s        -> L.build_load (lookup s) s builder
    | SAssign (s, e) -> let e' = expr builder e in
                        ignore(L.build_store e' (lookup s) builder); e'
```

```
  | SBinop ((A.Float,_ ) as e1, op, e2) ->
let e1' = expr builder e1
and e2' = expr builder e2 in
(match op with
  A.Add     -> L.build_fadd
| A.Sub     -> L.build_fsub
| A.Mult    -> L.build_fmul
| A.Div     -> L.build_fdiv
| A.Equal   -> L.build_fcmp L.Fcmp.Oeq
| A.Neq     -> L.build_fcmp L.Fcmp.One
| A.Less    -> L.build_fcmp L.Fcmp.Olt
| A.Leq     -> L.build_fcmp L.Fcmp.Ole
| A.Greater -> L.build_fcmp L.Fcmp.Ogt
| A.Geq     -> L.build_fcmp L.Fcmp.Oge
| A.And | A.Or | A.Band | A.Bor | A.Bxor | A.Lshift | A.Rshift ->
    raise (Failure "internal error: semant should have rejected and/or on float")
| A.Mod -> L.build_fadd
| A.Member -> L.build_fadd
| A.Exp -> L.build_fadd
) e1' e2' "tmp" builder
  | SBinop (e1, op, e2) ->
let e1' = expr builder e1
and e2' = expr builder e2 in
(match op with
  A.Add     -> L.build_add
| A.Sub     -> L.build_sub
| A.Mult    -> L.build_mul
      | A.Div     -> L.build_sdiv
| A.And     -> L.build_and
| A.Or      -> L.build_or
| A.Band     -> L.build_and
| A.Bor      -> L.build_or
| A.Bxor      -> L.build_xor
| A.Lshift -> L.build_shl
| A.Rshift -> L.build_ashr
| A.Equal   -> L.build_icmp L.Icmp.Eq
| A.Neq     -> L.build_icmp L.Icmp.Ne
| A.Less    -> L.build_icmp L.Icmp.Slt
| A.Leq     -> L.build_icmp L.Icmp.Sle
| A.Greater -> L.build_icmp L.Icmp.Sgt
| A.Geq     -> L.build_icmp L.Icmp.Sge
| A.Mod -> L.build_add
```

```
    | A.Member -> L.build_add
    | A.Exp -> L.build_add
    ) e1' e2' "tmp" builder
      | SUnop(op, ((t, _) as e)) ->
          let e' = expr builder e in
    (match op with
      A.Neg when t = A.Float -> L.build_fneg
    | A.Neg                  -> L.build_neg
    | A.Bnot -> L.build_neg
    | A.Incr -> L.build_neg
    | A.Decr -> L.build_neg
          | A.Not                  -> L.build_not) e' "tmp" builder
      (* Print ints *)
      | SCall ("print", [e]) -> L.build_call printf_func [| int_format_str ; (expr
builder e) |] "print" builder

      (* Print bools*)
      | SCall ("printb", [e]) -> L.build_call printf_func [| bool_format_str ; (expr
builder e) |] "printb" builder

      (* print floats *)
      | SCall ("printf", [e]) -> L.build_call printf_func [| float_format_str ; (expr
builder e) |] "printf" builder

      (* print chars *)
      | SCall ("printc", [e]) -> L.build_call printf_func [| char_format_str ; (expr
builder e) |] "prints" builder

      (* print strings *)
      | SCall ("prints", [e]) -> L.build_call printf_func [| string_format_str ; (expr
builder e) |] "prints" builder

      (* printbig *)
      | SCall ("printbig", [e]) -> L.build_call printbig_func [| (expr builder e) |]
"printbig" builder

      (* min *)
      | SCall ("min",[e1;e2]) -> L.build_call min_func [| (expr builder e1) ; (expr
builder e2) |] "min" builder

      (* max *)
```

```
    | SCall ("max",[e1;e2]) -> L.build_call max_func [| (expr builder e1) ; (expr
builder e2) |] "max" builder

    (* strlen *)
    | SCall ("strlen", [e1])     -> L.build_call strlen_func  [| (expr builder e1) |]
"strlen" builder

    (* strcpy *)
    | SCall ("strcpy", [e1; e2]) -> L.build_call strcpy_func  [| (expr builder e1);
(expr builder e2) |] "strcpy" builder

    (* strcat *)
    | SCall ("strcat", [e1; e2]) -> L.build_call strcat_func  [| (expr builder e1);
(expr builder e2) |] "strcat" builder

    (* strcmp *)
    | SCall ("strcmp", [e1; e2]) -> L.build_call strcmp_func  [| (expr builder e1);
(expr builder e2) |] "strcmp" builder

    (* strstr *)
    | SCall ("strstr", [e1; e2]) -> L.build_call strstr_func  [| (expr builder e1);
(expr builder e2) |] "strstr" builder

    (* str_concat  *)
    | SCall ("str_concat", [e1 ; e2]) -> L.build_call str_concat_f [| (expr builder
e1) ; (expr builder e2)|] "str_concat" builder

    (* sin *)
    | SCall ("sin", [e]) -> L.build_call sin_func   [| (expr builder e) |] "sin"
builder

    (* cos *)
    | SCall ("cos",   [e]) -> L.build_call cos_func   [| (expr builder e) |] "cos"
builder

    (* tan *)
    | SCall ("tan",   [e]) -> L.build_call tan_func   [| (expr builder e) |] "tan"
builder

    (* sinh *)
    | SCall ("sinh",  [e]) -> L.build_call sinh_func  [| (expr builder e) |] "sinh"
builder
```

```
    (* cosh *)
    | SCall ("cosh",  [e]) -> L.build_call cosh_func  [| (expr builder e) |] "cosh"
builder

    (* tanh *)
    | SCall ("tanh",  [e]) -> L.build_call tanh_func  [| (expr builder e) |] "tanh"
builder

    (* exp *)
    | SCall ("exp",   [e]) -> L.build_call exp_func   [| (expr builder e) |] "exp"
builder

    (* log *)
    | SCall ("log",   [e]) -> L.build_call log_func   [| (expr builder e) |] "log"
builder

    (* sqrt *)
    | SCall ("sqrt",  [e]) -> L.build_call sqrt_func  [| (expr builder e) |] "sqrt"
builder

    (* rand *)
    | SCall ("rand",  []) -> L.build_call rand_func   [|  |] "rand"  builder
    | SCall ("print", [e]) ->
  L.build_call printf_func [| int_format_str ; (expr builder e) |]
  "print" builder

    (* Print bools*)
    | SCall ("printb", [e]) ->
  L.build_call printf_func [| bool_format_str ; (expr builder e) |]
  "printb" builder

    (* print floats *)
    | SCall ("printf", [e]) ->
  L.build_call printf_func [| float_format_str ; (expr builder e) |]
  "printf" builder

    (* print chars *)
    | SCall ("printc", [e]) ->
    L.build_call printf_func [| char_format_str ; (expr builder e) |]
    "prints" builder
```

```
    (* print strings *)
    | SCall ("prints", [e]) ->
    L.build_call printf_func [| string_format_str ; (expr builder e) |]
    "prints" builder

    (* printbig *)
    | SCall ("printbig", [e]) ->
  L.build_call printbig_func [| (expr builder e) |]
  "printbig" builder
    | SCall (f, args) ->
       let (fdef, fdecl) = StringMap.find f function_decls in
 let llargs = List.rev (List.map (expr builder) (List.rev args)) in
 let result = (match fdecl.styp with
                     A.Void -> ""
                   | _ -> f ^ "_result") in
       L.build_call fdef (Array.of_list llargs) result builder
  in

  (* LLVM insists each basic block end with exactly one "terminator"
     instruction that transfers control.  This function runs "instr builder"
     if the current block does not already have a terminator.  Used,
     e.g., to handle the "fall off the end of the function" case. *)
  let add_terminal builder instr =
    match L.block_terminator (L.insertion_block builder) with
Some _ -> ()
    | None -> ignore (instr builder) in
   (* Build the code for the given statement; return the builder for
     the statement's successor (i.e., the next instruction will be built
     after the one generated by this call) *)

  let rec stmt break_bb continue_bb builder = function
    SBlock sl -> List.fold_left (stmt break_bb continue_bb) builder sl
    | SContinue -> ignore (L.build_br continue_bb builder);builder
    | SBreak -> ignore (L.build_br break_bb builder);builder
    | SExpr e -> ignore(expr builder e); builder
    | SReturn e -> ignore(match fdecl.styp with
                            (* Special "return nothing" instr *)
                           A.Void -> L.build_ret_void builder
                            (* Build return statement *)
                         | _ -> L.build_ret (expr builder e) builder );
                  builder
    | SIf (predicate, then_stmt, else_stmt) ->
```

```
        let bool_val = expr builder predicate in
  let merge_bb = L.append_block context "merge" the_function in
        let build_br_merge = L.build_br merge_bb in (* partial function *)

  let then_bb = L.append_block context "then" the_function in
  add_terminal (stmt break_bb continue_bb (L.builder_at_end context then_bb)
then_stmt)
    build_br_merge;

  let else_bb = L.append_block context "else" the_function in
  add_terminal (stmt break_bb continue_bb (L.builder_at_end context else_bb)
else_stmt)
    build_br_merge;

  ignore(L.build_cond_br bool_val then_bb else_bb builder);
  L.builder_at_end context merge_bb

    | SWhile (predicate, body) ->
   let pred_bb = L.append_block context "while" the_function in
   ignore(L.build_br pred_bb builder);

   let body_bb = L.append_block context "while_body" the_function in

   let pred_builder = L.builder_at_end context pred_bb in
   let bool_val = expr pred_builder predicate in

   let merge_bb = L.append_block context "merge" the_function in
   add_terminal (stmt merge_bb pred_bb (L.builder_at_end context body_bb) body)
     (L.build_br pred_bb);
   ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
   L.builder_at_end context merge_bb

     (* Implement for loops as while loops *)
     | SFor (e1, e2, e3, body) -> stmt break_bb continue_bb builder
     ( SBlock [SExpr e1 ; SWhile (e2, SBlock [body ; SExpr e3]) ] )
   in
   (* Build dummy block to use for initial break/continue basic blocks*)
   let dummy_bb = L.append_block context "dummy" the_function in
   ignore (L.build_unreachable (L.builder_at_end context dummy_bb));
   (* Build the code for each statement in the function *)
   let builder = stmt dummy_bb dummy_bb  builder (SBlock fdecl.sbody) in
```

```
    (* Add a return if the last block falls off the end *)
    add_terminal builder (match fdecl.styp with
        A.Void -> L.build_ret_void
      | A.Float -> L.build_ret (L.const_float float_t 0.0)
      | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in

  List.iter build_function_body functions;
  The_module
```

12.1.7 Gaspp.ml

```
(* Top-level gassp compiler, appropriated from microc *)

type action = Ast | Sast | LLVM_IR | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./gassp.native [-a|-s|-l|-c] [file.mc]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;
   let lexbuf = Lexing.from_channel !channel in
  let ast = Gasspparser.program Scanner.token lexbuf in
  match !action with
    Ast -> print_string (Ast.string_of_program ast)
  | _ -> let sast = Semant.check ast in
    match !action with
      Ast     -> ()
    | Sast    -> print_string (Sast.string_of_sprogram sast)
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
    | Compile -> let m = Codegen.translate sast in
  Llvm_analysis.assert_valid_module m;
  print_string (Llvm.string_of_llmodule m)
```

12.2 Gassp with objects

12.2.1 Scanner

```
{ open Gasspparser }

let digit = ['0' - '9']
let digits = digit+
let letter = [ '_' 'a' - 'z' 'A' - 'Z']
let variable = letter['a'-'z' 'A'-'Z' '0'-'9' '_']*
let whitespace = [' ' '\t' '\r' '\n']+
let float_literal = digit+ '.' digit*
         | digit* '.' digit+
         | digit+ ('.')? digit* ['e' 'E'] ['+' '-']? digit+
         | digit* ('.')? digit+ ['e' 'E'] ['+' '-']? digit+
let ascii = [' '-'!' '#'-'~']
let char_literal = ''' (ascii as chlit) '''
let string_literal = '"' (ascii* as slit) '"'


rule token = parse
 [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LBRACKET }
| ']'      { RBRACKET }
| ';'      { SEMI }
| ':'      { COLON }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '%'      { MOD }
| '/'      { DIVIDE }
| "**"     { EXP }
| '.'      { DOT }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
```

```
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "~"      { BNOT }
| "|"      { BOR }
| "^"      { BXOR }
| "&"      { BAND }
|"<<"      { LSHIFT }
|">>"      { RSHIFT }
| "++"     { INCR}
| "--"     { DECR }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "char"   { CHAR }
| "string" { STRING }
| "bool"   { BOOL }
| "float"  { FLOAT }
| "void"   { VOID }
| "true"   { BLIT(true)  }
| "false"  { BLIT(false) }
| "switch" {SWITCH}
| "case" {CASE}
| "default" {DEFAULT}
| "continue" {CONTINUE}
| "break" {BREAK}
| digits as lxm { LITERAL(int_of_string lxm) }
| "class" { CLASS}
| char_literal         { CHLIT(chlit) }
| float_literal as lxm { FLIT(lxm) }
| string_literal       { SLIT(slit) }
| variable     as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
```

```
 "*/" { token lexbuf }
| _    { comment lexbuf }
```

12.2.2 Parser

```
/* Ocamlyacc parser for GA$$P*/

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA PLUS MINUS TIMES DIVIDE ASSIGN DOT
LBRACKET RBRACKET MOD INCR DECR BREAK CONTINUE SWITCH CASE DEFAULT COLON EXP PUBLIC
PRIVATE NULL
%token NOT EQ NEQ LT LEQ GT GEQ AND OR BNOT BOR BXOR BAND LSHIFT RSHIFT NEW
%token RETURN IF ELSE FOR WHILE INT BOOL CHAR FLOAT VOID STRING
%token <int> LITERAL
%token <bool> BLIT
%token <string> SLIT
%token <char> CHLIT
%token <string> ID FLIT
%token CLASS
%token EOF

%start program
%type <Ast.program> program

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

%%

program:
 cdecls fdecl EOF { Program($1, $2) }
```

```
acc_mod:
| PUBLIC {Public}
| PRIVATE {Private}
decls:
  /* nothing */ { ([], [])               }
| decls vdecl { (($2 :: fst $1), snd $1) }
| decls fdecl { (fst $1, ($2 :: snd $1)) }
/* | decls cdecl { (fst $1, ($2 :: snd $1)) } */

fdecl:
  acc_mod typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
    { {
      faccess = $1;
      ftyp = $2;
  fname = $3;
  fformals = List.rev $5;
  fbody = List.rev $8 } }

/* class body */
cbody:
{{fields = []; constructors =[]; methods =[];}}
| cbody vdecl {{fields = $2::$1.fields; constructors = $1.constructors; methods =
$1.methods}}
| cbody constructor {{fields = $1.fields; constructors = $2::$1.constructors; methods
= $1.methods}}
| cbody fdecl {{fields = $1.fields; constructors = $1.constructors; methods =
$2::$1.methods}}



constructor:
| ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE {{
  faccess = Public;
  fname = $1;
  ftyp = Object($1);
  fformals = $3;
  fbody = List.rev $6;
}}

cdecls:
```

```
| cdecl_list {List.rev $1}

cdecl_list:
| cdecl {[$1]}
| cdecl_list cdecl {$2::$1}

cdecl:
   acc_mod CLASS ID LBRACE cbody RBRACE {{caccess = $1; cname = $3; cbody = $5}}
   /* CLASS ID LBRACE cbody RBRACE {()} */

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

formal_list:
   typ ID                   { [{fvtyp = $1; fvname = $2;}]     }
 | formal_list COMMA typ ID { {fvtyp = $3; fvname = $4;} :: $1 }

typ:
   INT    { Int    }
 | CHAR   { Char   }
 | STRING{ String}
 | BOOL   { Bool   }
 | FLOAT  { Float  }
 | VOID   { Void   }


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

vdecl:
  acc_mod typ ID SEMI { {vaccess = $1; vtyp = $2; vname = $3; vexpr = Noexpr;} }
  | acc_mod typ ID ASSIGN expr SEMI {{vaccess = $1; vtyp = $2; vname = $3; vexpr =
$5;} }

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

stmt:
   expr SEMI                              { Expr $1               }
```

```
  | RETURN expr_opt 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_opt SEMI expr SEMI expr_opt RPAREN stmt
                                            { For($3, $5, $7, $9)   }
  | WHILE LPAREN expr RPAREN stmt           { While($3, $5)         }
  | BREAK SEMI                                {Break}
  | CONTINUE SEMI                             {Continue}
  | vdecl                                   {Variable($1)}
  | typ ID SEMI                             {LocalVariable($1, $2, Noexpr)}
  | typ ID SEMI ASSIGN expr SEMI            {LocalVariable($1, $2, Noexpr)}


expr_opt:
   /* nothing */ { Noexpr }
  | expr          { $1 }

expr:
   LITERAL          { Literal($1)            }
  | CHLIT           { ChLiteral($1)              }
  | SLIT            { StrLiteral($1)               }
  | FLIT            { FLiteral($1)           }
  | BLIT            { BoolLiteral($1)            }
  | ID              { Id($1)                 }
  | NULL            { Null}
  | expr PLUS   expr { Binop($1, Add,   $3)   }
  | expr MOD    expr { Binop($1, Mod,   $3)   }
  | expr MINUS  expr { Binop($1, Sub,   $3)   }
  | expr TIMES  expr { Binop($1, Mult,  $3)   }
  | expr DIVIDE expr { Binop($1, Div,   $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 GEQ    expr { Binop($1, Geq,   $3)   }
  | expr BAND    expr { Binop($1, Band,   $3)   }
  | expr BOR     expr { Binop($1, Bor,    $3)   }
  | expr BXOR     expr { Binop($1, Bxor,    $3)   }
  | expr AND    expr { Binop($1, And,   $3)   }
  | expr OR     expr { Binop($1, Or,    $3)   }
```

```
 | expr LSHIFT expr { Binop($1, Lshift,   $3)   }
 | expr RSHIFT expr { Binop($1, Rshift,    $3)   }
 | expr DOT expr { AccessMember($1, $3)}
 | MINUS expr %prec NOT { Unop(Neg, $2)      }
 | expr INCR     { Unop(Incr, $1)      }
 | expr DECR     { Unop(Decr, $1)      }
 | NOT expr         { Unop(Not, $2)          }
 | BNOT expr         { Unop(Bnot, $2)          }
 | expr ASSIGN expr   { Assign($1, $3)         }
 | ID LPAREN args_opt RPAREN { Call($1, $3)  }
 | LPAREN expr RPAREN { $2                    }
 | NEW ID LPAREN args_opt RPAREN {NewObject($2, $4)}

args_opt:
   /* nothing */ { [] }
 | args_list  { List.rev $1 }

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

12.2.3 Ast

```
(* Abstract Syntax Tree and functions for printing it *)

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq |
          And | Or | Mod | Member | Exp | Lshift | Rshift | Band | Bxor | Bor

type uop = Neg | Not | Incr | Decr | Bnot

type typ = Int | Bool | Float | Void | Char | String | Object of string

type access_modifier = Private | Public

type formal = {
 fvtyp: typ;
 fvname: string;
}


type expr =
   Literal of int
```

```
 | StrLiteral of string
 | ChLiteral of char
 | FLiteral of string
 | BoolLiteral of bool
 | Id of string
 | Binop of expr * op * expr
 | Unop of uop * expr
 | Assign of expr * expr
 | Call of string * expr list
 | Noexpr
 | NewObject of string * expr list
 | AccessMember of expr * expr
 | Null

type vdecl = {
 vaccess: access_modifier;
 vtyp: typ;
 vname: string;
 vexpr: expr;
}
type stmt =
   Block of stmt list
 | Expr of expr
 | Return of expr
 | If of expr * stmt * stmt
 | For of expr * expr * expr * stmt
 | While of expr * stmt
 | Break
 | Continue
 | Variable of vdecl
 | LocalVariable of typ * string * expr

type func_decl = {
 faccess: access_modifier;
 ftyp : typ;
 fname : string;
 fformals : formal list;
 fbody : stmt list;
}
```

```
type cbody = {
 fields: vdecl list;
 constructors : func_decl list;
 methods : func_decl list;
}

type class_decl = {
 caccess: access_modifier;
 cname : string;
 cbody : cbody;
}

type arr_decl = {
 arrtyp : typ;
 size: int;
}

type program = Program of class_decl list * func_decl


let rec getType expr env = match expr with
   Id(s)      -> Int
 | Literal(s)     -> Int
 | FLiteral(f)    -> Float
 | ChLiteral(c)     -> Char
 | StrLiteral(s)    -> String
 | BoolLiteral(b)     -> Bool
 | Noexpr     -> Void
 | Null       -> Void
 | Binop(e1, op, e2)    -> (match op with
       Equal|Neq|Less|Leq|Greater|Geq|Or|And -> Bool
     | Add|Sub|Mult|Div -> if ((getType e1 env)=Float) || ((getType e2 env)=Float)
       then Float
       else Int
     | _ -> Int)
 | Unop(op, e)    -> getType e env
 | Assign(s, e)     -> getType e env
 | AccessMember(e1, e2)   -> Int
 | NewObject(s, el)   -> Object(s)
 | Call(s, el) -> Int
 | _ -> Void
```

```
(* Pretty-printing functions *)
let string_of_typ = function
   Char     -> "char"
 | Void     -> "void"
 | Bool  -> "boolean"
 | Float    -> "float"
 | Int     -> "int"
 | Object(_) -> "object"
 | String -> "string"
let string_of_op = function
   Add -> "+"
 | Sub -> "-"
 | Mult -> "*"
 | Div -> "/"
 | Equal -> "=="
 | Neq -> "!="
 | Less -> "<"
 | Leq -> "<="
 | Greater -> ">"
 | Geq -> ">="
 | And -> "&&"
 | Or -> "||"
 | Mod -> "%"
 | Member -> "."
 | Exp -> "**"
 | Band -> "&"
 | Bor -> "|"
 | Bxor -> "^"
 | Lshift -> "<<"
 | Rshift -> ">>"

let string_of_uop = function
   Neg -> "-"
 | Decr -> "--"
 | Incr -> "++"
 | Not -> "!"
 | Bnot -> "!"

let string_of_char c = String.make 1 c
let rec string_of_expr = function
   Literal(l) -> string_of_int l
 | FLiteral(l) -> l
```

```
 | BoolLiteral(true) -> "true"
 | BoolLiteral(false) -> "false"
 | StrLiteral(s) -> s
 | ChLiteral(c) -> string_of_char c
 | Id(s) -> s
 | Binop(e1, o, e2) ->
   string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
 | Unop(o, e) -> string_of_uop o ^ string_of_expr e
 | Assign(e1, e2) -> string_of_expr e1 ^ " = " ^ string_of_expr e2
 | Call(f, el) ->
   f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
 | Noexpr -> ""
 | Null -> "null"
 | NewObject(s, args) -> "new " ^ s ^ "(" ^ String.concat ", " (List.map
string_of_expr args) ^ ")"
 | AccessMember(e1, e2) -> "member access"

let rec string_of_stmt = function
   Block(stmts) ->
   "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
 | Expr(expr) -> string_of_expr expr ^ ";\n";
 | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
 | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
 | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
                     string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
 | For(e1, e2, e3, s) ->
   "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
   string_of_expr e3  ^ ") " ^ string_of_stmt s
 | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
 | Break -> "break;"
 | Continue -> "continue;"


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

let string_of_edecl (t, id, e) = string_of_typ t ^ " " ^ id ^ " = " ^ string_of_expr e
^ ";\n"

let string_of_formal f = string_of_typ f.fvtyp ^ " " ^ f.fvname
let string_of_fdecl fdecl =
 string_of_typ fdecl.ftyp ^ " " ^
 fdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_formal fdecl.fformals) ^
```

```
 ")\n{\n" ^
 String.concat "" (List.map string_of_stmt fdecl.fbody) ^
 "}\n"

let string_of_cdecl cdecl = "class " ^ cdecl.cname
let string_of_program = function
 | Program(classes, main) -> String.concat "" (List.map string_of_cdecl classes) ^
"\n" ^ (string_of_fdecl main)
 | _ -> ""
```

12.2.4 Sast

```
(* Semantically-checked Abstract Syntax Tree and functions for printing it *)

open Ast

type sexpr =
   SLiteral of int
 | SFLiteral of string
 | SStrLiteral of string
 | SChLiteral of char
 | SBoolLiteral of bool
 | SId of string * typ
 | SBinop of sexpr * op * sexpr * typ
 | SUnop of uop * sexpr * typ
 | SAssign of sexpr * sexpr * typ
 | SCall of string * sexpr list * typ
 | SNoexpr
 | SNull
 | SAccessMember of sexpr * sexpr * typ
 | SNewObject of string * sexpr list * typ

type sformal = {
 sfvtyp: typ;
 sfvname: string;
}


type svdecl = {
 svaccess: access_modifier;
 svtyp:  typ;
 svname: string;
```

```
 svexpr: sexpr;
}

type sstmt =
   SBlock of sstmt list
 | SExpr of sexpr * typ
 | SReturn of sexpr * typ
 | SIf of sexpr * sstmt * sstmt
 | SFor of sexpr * sexpr * sexpr * sstmt
 | SWhile of sexpr * sstmt
 | SBreak
 | SContinue
 | SVariable of svdecl
 | SLocalVariable of typ * string * sexpr

type sfunc_decl = {
 sfaccess: access_modifier;
 sftyp : typ;
 sfname : string;
 sfformals : sformal list;
 sfbody : sstmt list;
}


type scbody = {
 sfields: svdecl list;
 sconstructors: sfunc_decl list;
 smethods: sfunc_decl list;
}

type sclass_decl = {
 scaccess: access_modifier;
 scname : string;
 scbody : scbody;
}

type sarr_decl = {
 sarrtyp : typ;
 ssize: int;
}

type sprogram = {
```

```
 classes: sclass_decl list;
 main: sfunc_decl;
 functions: sfunc_decl list;
}

(* Pretty-printing functions *)

(* let rec string_of_sexpr (t, e) =
 "(" ^ string_of_typ t ^ " : " ^ (match e with
       SLiteral(l) -> string_of_int l
     | SBoolLiteral(true) -> "true"
     | SBoolLiteral(false) -> "false"
     | SStrLiteral(s) -> s
     | SChLiteral(c) -> string_of_char c
     | SFLiteral(l) -> l
     | SId(s, t) -> string_of_typ t ^ s
     | SBinop(e1, o, e2, t) ->
       string_of_sexpr (t, e1) ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr (t, e2)
     | SUnop(o, e, t) -> string_of_typ t ^ string_of_uop o ^ string_of_sexpr (t, e)
     | SAssign(e, e2, t) -> string_of_sexpr (t,e) ^ " = " ^ string_of_sexpr (t,e2)
     | SCall(f, el, t) ->
       f ^ "(" ^ String.concat ", " (List.map (fun e -> string_of_sexpr(t,e)) el) ^
")"
     | SNoexpr -> ""
   ) ^ ")"

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

```
let string_of_sformal = function
| {sfvtyp = sfvtyp; sfvname = sfvname;} -> string_of_typ sfvtyp ^ " " ^ sfvname
| _ -> ""
let string_of_sfdecl fdecl =
 string_of_typ fdecl.sftyp ^ " " ^
 fdecl.sfname ^ "(" ^ String.concat ", " (List.map string_of_formal fdecl.sfformals) ^
 ")\n{\n" ^
 String.concat "" (List.map string_of_sstmt fdecl.sfbody) ^
 "}\n" *)

let string_of_sprogram sp = match sp with
| {classes; main; functions;} -> "sprogram"
| _ -> ""
```

12.2.5 Semant

```
open Ast
open Sast
open Env

module StringMap = Map.Make(String)

let classIndices: (string, int) Hashtbl.t =  Hashtbl.create 10

let createClassIndices cdecls=
 let classHandler index cdecl=
   Hashtbl.add classIndices cdecl.cname index in (*scope handling is missing*)
 List.iteri classHandler cdecls

let builtinMethods = ["print"; "println"]

let isMain f = f.sfname = "main"

let get_methods l classy =  List.concat [classy.scbody.smethods;l]

let get_main m = try (List.hd (List.filter isMain (List.fold_left get_methods [] m)))
with Failure("hd") -> raise(Failure("No main method was defined"))

let get_methods_minus_main m = snd (List.partition isMain (List.fold_left get_methods
[] m))

let newEnv env = {
```

```
 envClassName = env.envClassName;
 envClassMaps = env.envClassMaps;
 envClassMap  = env.envClassMap;
 envLocals    = env.envLocals;
 envParams    = env.envParams;
 envReturnType = env.envReturnType;
 envBuiltinMethods = env.envBuiltinMethods;
}

let typOfSexpr = function
   SLiteral(i)     -> Int
 |   SBoolLiteral(b)     -> Bool
 |   SFLiteral(f)      -> Float
 |   SStrLiteral(s)      -> String
 |   SChLiteral(c)       -> Char
 |   SId(_, d)       -> d
 |   SBinop(_, _, _, d)    -> d
 |   SAssign(_, _, d)    -> d
 |   SCall(_, _, d)  -> d
 |   SUnop(_, _, d)      -> d
 |   SNewObject(_,_,d) -> d
 |   SAccessMember(_,_,d) -> d
 | _ -> Void

let rec typOfExpr = function
   Literal(i)      -> Int
 |   BoolLiteral(b)      -> Bool
 |   FLiteral(f)     -> Float
 |   StrLiteral(s)       -> String
 |   ChLiteral(c)      -> Char
 |   Id(_)       -> Int
 |   Binop(_, _, e)    -> typOfExpr e
 |   Assign(_, e)    -> typOfExpr e
 |   Call(_, e)  -> Int
 |   Unop(_, e)      -> typOfExpr e
 |   NewObject(s,_)  -> Object(s)
 |   AccessMember(_,e) -> typOfExpr e
 | _ -> Void

let convertToSast classes main =

 let convertFormalToSast formal env =
```

```
    {
      sfvtyp = formal.fvtyp;
      sfvname = formal.fvname;
    }
  in
  let checkBinop e1 op e2 =
    ()
  in
  let checkUnop op e env =
    ()
  in
  let checkFuncCall s el env =
    ()
  in
  let checkAssign e1 e2 env =
    ()
  in
  let rec convertExprToSast expr env = match expr with
      Literal(i)  -> SLiteral(i)
    | BoolLiteral(b)  -> SBoolLiteral(b)
    | FLiteral(f) -> SFLiteral(f)
    | StrLiteral(s) -> SStrLiteral(s)
    | ChLiteral(c)  -> SChLiteral(c)
    | Null    -> SNull
    | Noexpr  -> SNoexpr
    | Id(id)  -> SId(id, Int (*getIdType id env*))
    | Binop(expr1, op, expr2) -> ignore(checkBinop expr1 op
expr2);SBinop(convertExprToSast expr1 env, op, convertExprToSast expr2 env, Int
(*getType expr1 env)*))
    | Assign(e1, e2)    -> ignore(checkAssign e1 e2 env); SAssign(convertExprToSast e1
env, convertExprToSast e2 env,Int(* getType e1 env*))
    | Call(s, el)   -> ignore(checkFuncCall s el env); SCall(s, (List.map (fun e ->
convertExprToSast e env) el), Int)
    | Unop(op, expr)    -> ignore(checkUnop op expr env); SUnop(op, convertExprToSast
expr env, Int (*getType expr env*))
    | NewObject(s,el) -> SNewObject(s, (List.map (fun e -> convertExprToSast e env)
el), Int (*Object(s)*))
    | AccessMember(e1,e2) -> SAccessMember(convertExprToSast e1 env, convertExprToSast
e2 env, Int (*getType e1 env*)) (* @TODO Double check type *)
  in
  let convertVdeclToSast vdecl env = {
    svaccess = vdecl.vaccess;
```

```
    svtyp = vdecl.vtyp;
    svname = vdecl.vname;
    svexpr = convertExprToSast vdecl.vexpr env;
  } in
  let checkLocalVariable dt id e env =
    env.envLocals <- StringMap.add id dt env.envLocals
  in
  let checkReturn e env =
    ()
  in
  let checkIf e s1 s2 env =
    ()
  in
  let checkFor e1 e2 e3 s env =
    ()
  in
  let checkWhile e s env =
    ()
  in
  let rec convertStmtToSast stmt env = match stmt with
      Block(sl)      -> SBlock(List.map (fun s -> convertStmtToSast s env) sl)
    | Expr(expr)       -> SExpr((convertExprToSast expr env), Int)
    | Variable(vdecl)    -> SVariable(convertVdeclToSast vdecl env)
    | LocalVariable(dt, id, expr) -> (checkLocalVariable dt id expr env;
SLocalVariable(dt, id, convertExprToSast expr env))
    | Return(expr)       -> ignore(checkReturn expr env); SReturn((convertExprToSast
expr env), Int)
    | If(expr, stmt1, stmt2)  -> ignore(checkIf expr stmt1 stmt2 env);
SIf(convertExprToSast expr env, convertStmtToSast stmt1 (newEnv env),
convertStmtToSast stmt2 (newEnv env))
    | For(expr1, expr2, expr3, stmt)-> ignore(checkFor expr1 expr2 expr3 stmt env);
SFor(convertExprToSast expr1 env, convertExprToSast expr2 env, convertExprToSast expr3
env, convertStmtToSast stmt (newEnv env))
    | While(expr, stmt)    -> ignore(checkWhile expr stmt env); SWhile(convertExprToSast
expr env, convertStmtToSast stmt (newEnv env))
  in

  (* Semantic Checking for class methods *)
  let rec strOfFormals fl = match fl with
    |  [] -> ""
    | h::t -> string_of_typ h ^ (strOfFormals t)
  in
```

```
 let getListOfFormalTypes c = List.map (fun f -> f.fvtyp) c.fformals
 in
 let hasDuplicateFormalNames l =
   let result = ref false
   in let names = List.map (fun f -> f.fvname) l
   in List.iter (fun e -> result := !result || e) (List.map (fun n -> List.length
(List.filter (fun s -> s = n) names) > 1) names);!result
 in
 let checkMethod func_decl classEnv =
   {
     formalAccess = Public;
     formalName = "";
     formalTyps = [];
     formalTyp = Int;
   }

 in
 let setEnvParams formals env =
   List.map (fun f -> env.envParams <- StringMap.add f.fvname f.fvtyp env.envParams)
formals

 in
 let convertMethodToSast func_decl classEnv =

   let methodSignature = checkMethod func_decl classEnv
   in
   let _ = classEnv.classMap.methodMap <- StringMap.add func_decl.fname
methodSignature classEnv.classMap.methodMap
   in
   let env = {
     envClassName = classEnv.className;
     envClassMaps = classEnv.classMaps;
     envClassMap  = classEnv.classMap;
     envLocals    = StringMap.empty;
     envParams    = StringMap.empty;
     envReturnType= func_decl.ftyp;
     envBuiltinMethods = classEnv.builtinMethods;
   } in
   let _ = setEnvParams func_decl.fformals env
   in
   {
     sfaccess = func_decl.faccess;
```

```
    sfname = func_decl.fname;
    sfformals = List.map (fun f -> convertFormalToSast f env) func_decl.fformals;
    sftyp = func_decl.ftyp;
    sfbody = List.map (fun s -> convertStmtToSast s env) func_decl.fbody;
  }

in
(* Semantic checking for class constructor *)
let checkConstructor constructor classEnv =
  (*    let formalTypes = getListOfFormalTypes constructor
        in
        let checking =

          if(StringMap.mem (strOfFormals formalTypes)
classEnv.classMap.constructorMap)
            then raise (Failure("Duplicate Constructor Definition"))
          else if ((List.length formalTypes <> 0) && (hasDuplicateFormalNames
constructor.fformals = true))
            then raise(Failure("Formal names must be unique"))
        in checking;
              {
                    mscope = constructor.fscope;
                          mname  = constructor.fname;
                          mformalTypes = formalTypes;
          mReturn = JVoid;
                  }
  *)
  {
    formalAccess = Public;
    formalName = "";
    formalTyps = [];
    formalTyp = Int;
  }
in
let convertConstructorToSast constructor classEnv =
  let constructorSignature = checkConstructor constructor classEnv
  in
  let _ = classEnv.classMap.constructorMap <- StringMap.add (strOfFormals
constructorSignature.formalTyps) constructorSignature classEnv.classMap.constructorMap
  in
  let env = {
    envClassName = classEnv.className;
```

```
    envClassMaps = classEnv.classMaps;
    envClassMap  = classEnv.classMap;
    envLocals    = StringMap.empty;
    envParams    = StringMap.empty;
    envReturnType= constructor.ftyp;
    envBuiltinMethods = classEnv.builtinMethods;
  } in
  let _ = setEnvParams constructor.fformals env
  in
  {
    sfaccess = constructor.faccess;
    sfname = constructor.fname;
    sfformals = List.map (fun f -> convertFormalToSast f env) constructor.fformals;
    sftyp = constructor.ftyp;
    sfbody = List.map (fun s -> convertStmtToSast s env) constructor.fbody;
  }

in
(* Sematic checking for class variable *)
let checkVdecl vdecl env =
  "placeholder"
  (*    let check =
          if StringMap.mem vdecl.vname env.envClassMap.variableMap
            then raise (Failure("Variable name already used"))
          else if vdecl.vexpr <> Ast.Noexpr && getType vdecl.vexpr env <> vdecl.vtype
            then raise (Failure(str_of_expr vdecl.vexpr ^ " is of type " ^
str_of_type (getType vdecl.vexpr env) ^ " but type " ^ str_of_type vdecl.vtype ^ " is
expected"))
        in check
  *)  in

let convertVariableToSast vdecl classEnv =
  let env = {
    envClassName = classEnv.className;
    envClassMaps = classEnv.classMaps;
    envClassMap  = classEnv.classMap;
    envLocals    = StringMap.empty;
    envParams    = StringMap.empty;
    envReturnType= Void; (*@TODO make sure return is not possible here*)
    envBuiltinMethods = classEnv.builtinMethods;
  } in
  let _ = checkVdecl vdecl env
```

```
    in
    let _ =  classEnv.classMap.fieldMap <- StringMap.add vdecl.vname vdecl
classEnv.classMap.fieldMap
    in {
      svaccess = vdecl.vaccess;
      svtyp  = vdecl.vtyp;
      svname  = vdecl.vname;
      svexpr  = convertExprToSast vdecl.vexpr env;
    }
 in
 let convertCbodyToSast cbody classEnv =
    {
      sfields = List.map (fun v -> convertVariableToSast v classEnv) (List.rev
cbody.fields);
      sconstructors = List.map (fun cst -> convertConstructorToSast cst classEnv)
(List.rev cbody.constructors);
      smethods = List.map (fun m -> convertMethodToSast m classEnv) (List.rev
cbody.methods);
    }

 in
 let checkClass class_decl classEnv =
    let firstChar = String.get class_decl.cname 0
    in
    let lowerChar = Char.lowercase firstChar
    in
    let checking =
      if lowerChar = firstChar
      then raise (Failure ("Class name not capitalized: " ^ class_decl.cname))
      else if StringMap.mem class_decl.cname classEnv.classMaps
      then raise (Failure ("Duplicate Class Name: " ^ class_decl.cname))
    in checking
 in
 let convertClassToSast class_decl classEnv =
    classEnv.className <- class_decl.cname;
    checkClass class_decl classEnv;
    let classMap = {
      fieldMap = StringMap.empty;
      constructorMap = StringMap.empty;
      methodMap = StringMap.empty;
    } in
    classEnv.classMap <- classMap;
```

```
    let result =
      {
        scaccess = class_decl.caccess;
        scname  = class_decl.cname;
        scbody  = convertCbodyToSast class_decl.cbody classEnv;
      } in
    classEnv.classMaps <- StringMap.add classEnv.className classEnv.classMap
classEnv.classMaps;
    result

 in
 let classEnv = {
    className = "";
    classMaps = StringMap.empty;
    classMap  = {
      fieldMap = StringMap.empty;
      constructorMap = StringMap.empty;
      methodMap = StringMap.empty;
    };
    builtinMethods = builtinMethods;
 }
 in
 let get_classes =
    List.map (fun c -> convertClassToSast c classEnv) classes
 in
 let sprogram =
    {
      classes = get_classes;
      main = convertMethodToSast main classEnv;
      functions = [];
    }
 in
 sprogram

(* Translates Ast to Sast *)
let check program = match program with
    Program (classes, main) -> ignore (createClassIndices classes); convertToSast
classes main
```

12.2.6 Codegen

```
open Env
open Llvm
open Hashtbl
open Ast
open Sast
open Semant

module L = Llvm
module A = Ast

(* Borrowed heavily from Java+-
http://www.cs.columbia.edu/~sedwards/classes/2016/4115-fall/reports/Java+-.pdf with
modifications to fit our SAST*)
let context = global_context ()
let the_module = create_module context "gassp"
let builder = builder context

let i32_t = L.i32_type context;;
let i8_t = L.i8_type context;;
let i1_t = L.i1_type context;;
let i64_t = L.i64_type context ;;
let f_t = L.double_type context;;

let boolean_True =  L.const_int i1_t 1;;
let boolean_False = L.const_int i1_t 0;;

let str_t = L.pointer_type i8_t;;
let void_t = L.void_type context;; (* void *)

let global_var_table:(string, llvalue) Hashtbl.t = Hashtbl.create 100
let class_private_vars:(string, llvalue) Hashtbl.t = Hashtbl.create 100
let local_var_table:(string, llvalue) Hashtbl.t = Hashtbl.create 100
let struct_typ_table:(string, lltype) Hashtbl.t = Hashtbl.create 100
let struct_field_idx_table:(string, int) Hashtbl.t = Hashtbl.create 100

let rec get_ptr_type datatype = match datatype with
 |   _ -> raise(Failure("InvalidStructType Array Pointer Type"))

and get_llvm_type datatype = match datatype with (* LLVM type for AST type *)
   A.Char -> i8_t
 | A.Void -> void_t
 | A.Bool -> i1_t
```

```
 | A.Float -> f_t
 | A.Int -> i32_t
 | A.Object(s) -> L.pointer_type(find_llvm_struct_type s)
 | _ -> raise(Failure("Invalid Data Type"))

and find_llvm_struct_type name =
 try Hashtbl.find struct_typ_table name
 with | Not_found -> raise(Failure ("undeclared struct "^ name))

let find_func_in_module fname =
 match (L.lookup_function fname the_module) with
   None -> raise(Failure("Function: " ^ fname ^ " not found in module."))
 | Some f -> f


let translate sast =
 let classes = sast.classes in
 let main = sast.main in
 let functions = sast.functions in

 let util_func () =
   let printf_t = L.var_arg_function_type i32_t [| pointer_type i8_t |] in
   let malloc_t = L.function_type (str_t) [| i32_t |] in
   let lookup_t = L.function_type (pointer_type i64_t) [| i32_t; i32_t |] in

   let _ = L.declare_function "printf" printf_t the_module in
   let _ = L.declare_function "malloc" malloc_t the_module in
   let _ = L.define_function "lookup" lookup_t the_module in
   ()
 in
 let _ = util_func () in

 let zero = const_int i32_t 0 in


 let add_classes_to_hashTable c =
   let struct_typ = L.named_struct_type context c.scname in
   Hashtbl.add struct_typ_table c.scname struct_typ
 in
 let _ = List.map add_classes_to_hashTable classes in

 let define_vardecl c =
```

```
          List.iteri (
            fun i v ->
              Hashtbl.add global_var_table v.svname boolean_True;
          )
            c.scbody.sfields;
        in
        let _ = List.map define_vardecl classes in

        let define_classes c =
          let struct_t = Hashtbl.find struct_typ_table c.scname in
          let type_list = List.map (function sv -> get_llvm_type sv.svtyp)
c.scbody.sfields in
          let name_list = List.map (function sv -> sv.svname) c.scbody.sfields in
          let type_list = i32_t :: type_list in
          let name_list = ".key" :: name_list in
          let type_array = (Array.of_list type_list) in
          List.iteri (
            fun i f ->
              let n = c.scname ^ "." ^ f in
              Hashtbl.add struct_field_idx_table n i;
          )
            name_list;
          L.struct_set_body struct_t type_array true
        in
        let _ = List.map define_classes classes in

        let define_functions f =
          let fname = f.sfname in
          let is_var_arg = ref false in
          let types_of_parameters = List.rev ( List.fold_left
                                                 (fun l ->
                                                    (function
                                                        sformal-> get_llvm_type
sformal.sfvtyp::l
                                                      | _ -> ignore(is_var_arg = ref
true); l
                                                    )
                                                 )
                                                 []  f.sfformals)
          in

          let fty =
```

```
            if !is_var_arg
            then L.var_arg_function_type (get_llvm_type f.sftyp)
                (Array.of_list types_of_parameters)
            else L.function_type (get_llvm_type f.sftyp)
                (Array.of_list types_of_parameters)
          in
          L.define_function fname fty the_module
        in
        let _ =  List.map define_functions functions in

        let define_constructors c =
          List.map define_functions c.scbody.sconstructors
        in
        let _ = List.map define_constructors classes in



        let rec stmt_gen llbuilder = function
            SBlock sl         -> generate_block sl llbuilder
          | SExpr (se, _)     ->   expr_gen llbuilder se
          | SVariable sv           ->  generate_vardecl sv.svaccess sv.svtyp sv.svname
sv.svexpr llbuilder
          | SLocalVariable (dt, vname, vexpr)   -> generate_local_vardecl dt vname
vexpr llbuilder
          | SIf(e, s1, s2) -> generate_if e s1 s2 llbuilder
          | SWhile(e, s) -> generate_while e s llbuilder
          | SFor(e1, e2, e3, s) -> generate_for e1 e2 e3 s llbuilder
          | SReturn(e, d)   -> generate_return e d llbuilder

        and generate_block sl llbuilder =
          try List.hd (List.map (stmt_gen llbuilder) sl) with
          | Failure(_) -> raise(Failure("No body"));

        and generate_return e d llbuilder =
          match e with
          | SNoexpr -> L.build_ret_void llbuilder
          | _ -> L.build_ret (expr_gen llbuilder e) llbuilder

        and generate_vardecl scope datatype vname expr llbuilder =
          let allocatedMemory = L.build_alloca (get_llvm_type datatype) vname
llbuilder in
          Hashtbl.add
```

```
            (match scope with
             | A.Public -> global_var_table
             | A.Private -> class_private_vars) vname allocatedMemory;

          let variable_value = expr_gen llbuilder expr in
          match expr with
          | SNoexpr -> allocatedMemory
          | _ -> L.build_store variable_value allocatedMemory llbuilder;
variable_value

        and generate_local_vardecl datatype vname expr llbuilder =
          let allocatedMemory = L.build_alloca (get_llvm_type datatype) vname
llbuilder in
          Hashtbl.add local_var_table vname allocatedMemory;
          let variable_value = expr_gen llbuilder expr in
          match expr with
          | SNoexpr -> allocatedMemory
          | _ -> ignore (L.build_store variable_value allocatedMemory llbuilder);
variable_value

        and generate_while e s llbuilder =
          let start_block = L.insertion_block llbuilder in
          let parent_function = L.block_parent start_block in

          let pred_block = L.append_block context "while" parent_function in
          L.build_br pred_block llbuilder;

          let body_block = L.append_block context "while_body" parent_function in
          let body_builder = L.builder_at_end context body_block in

          let stmt = stmt_gen body_builder s in
          L.build_br pred_block body_builder;

          let pred_builder = L.builder_at_end context pred_block in

          let boolean_condition = expr_gen pred_builder e in
          let merge_block = L.append_block context "merge" parent_function in

          let whileStatement = L.build_cond_br boolean_condition body_block
merge_block pred_builder in

          L.position_at_end merge_block llbuilder;
```

```
    whileStatement

  and generate_for e1 e2 e3 s llbuilder =
    expr_gen llbuilder e1;
    let whileBody = SBlock [s; SExpr(e3, Int)] in  (* Int hardcoded*)
    generate_while e2 whileBody llbuilder

  and generate_if e s1 s2 llbuilder =
    let boolean_condition =
      match e with
      | SId (n, dt) -> get_value true n llbuilder
      | _ -> expr_gen llbuilder e
    in

    let start_block = L.insertion_block llbuilder in
    let parent_function = L.block_parent start_block in

    let merge_block = L.append_block context "merge" parent_function in

    let then_block = L.append_block context "then" parent_function in
    L.position_at_end then_block llbuilder;

    let stmt1 = stmt_gen llbuilder s1 in
    L.build_br merge_block llbuilder;

    let else_block = L.append_block context "else" parent_function in
    L.position_at_end else_block llbuilder;
    let stmt2 = stmt_gen llbuilder s2 in
    L.build_br merge_block llbuilder;

    L.position_at_end start_block llbuilder;
    let ifStatement = L.build_cond_br boolean_condition then_block else_block
llbuilder in
    L.position_at_end merge_block llbuilder;

    ifStatement

  and expr_gen llbuilder = function
      SLiteral(i)     ->  L.const_int i32_t i
    | SBoolLiteral (b) -> if b then L.const_int i1_t 1 else L.const_int i1_t 0
    | SFLiteral(_)   -> L.const_float f_t  0.0
```

```
          | SChLiteral(c)    -> L.const_int i8_t (Char.code c)
          | SStrLiteral(s)  ->  build_global_stringptr s "tmp" llbuilder
          | SId (n, dt)   -> get_value false n llbuilder
          | SBinop(e1, op, e2, dt) -> binop_gen e1 op e2 llbuilder
          | SUnop(op, e, dt)      -> unop_gen op e llbuilder
          | SAssign (e1, e2, dt)  -> assign_to_variable e1 e2 llbuilder
          | SNewObject(id, el, d) -> generate_object_create id el llbuilder
          | SAccessMember(e1, e2, dt) -> generate_object_access e1 e2 llbuilder
          | SCall (fname, expr_list, d) -> generate_function_call fname expr_list d
llbuilder
          | SNoexpr -> L.build_add (L.const_int i32_t 0) (L.const_int i32_t 0) "nop"
llbuilder
          | _ -> raise(Failure("No match for expression"))

        and generate_object_access e1 e2 llbuilder =
          let objectMemory = match e1 with
            | SId(id, dt) -> get_value true id llbuilder
            | _ -> raise(Failure("Not an id of object"))
          in
          let get_variable n llbuilder =
            let index = Hashtbl.find struct_field_idx_table ("."^n) in
            let var = L.build_struct_gep objectMemory index "temp" llbuilder in
            L.build_load var n llbuilder
          in
          let rhs = match e2 with
            | SId(id, dt) -> get_variable id llbuilder
            | _ -> raise(Failure("Function acces not yet supported"))
          in
          rhs

        and generate_create_tuples dt_list expr_list llbuilder =
          let type_list = List.map (function dt -> get_llvm_type dt) dt_list in
          let type_array = (Array.of_list type_list) in
          let struct_type = L.packed_struct_type context type_array in
          let vname = "dummy" in
          let allocatedMemory = L.build_alloca struct_type vname llbuilder in
          List.iteri (
            fun i f ->
              let tuple_value = L.build_struct_gep allocatedMemory i "temp" llbuilder
in
              ignore(L.build_store (match f with
                  | SId(id, d) -> get_value true id llbuilder
```

```
                | _ -> expr_gen llbuilder f) tuple_value llbuilder);
        ) expr_list;

        L.build_pointercast allocatedMemory (L.pointer_type struct_type)
"tupleMemAlloc" llbuilder

      and generate_tuple_access deref e1 e2 llbuilder =
        let vname = "dummy" in
        let index =  match e2 with
          | SLiteral(i) -> i
          | _ -> raise(Failure("Not an int"))
        in
        let tuple = match e1 with
          | SId(id, d) -> get_value true id llbuilder
          | _ -> raise(Failure("Not an id"))
        in
        let tuple_value = L.build_struct_gep tuple index vname llbuilder in
        if deref
        then L.build_load tuple_value vname llbuilder
        else tuple_value

      and generate_array_access deref e el llbuilder =
        match el with
        | [h] -> let index = match h with
            | SId(id, d) -> get_value true id llbuilder
            | _ -> expr_gen llbuilder h
          in
          let index = L.build_add index (const_int i32_t 1) "1tmp" llbuilder in
          let arr = match e with
            | SId(n, dt) -> get_value true n llbuilder
            | _ -> raise(Failure("Can't access array!"))
          in
          let _val = L.build_gep arr [| index |] "2tmp" llbuilder in
          if deref
          then build_load _val "3tmp" llbuilder
          else _val
        | _ ->  raise(Failure("Two dimentional array not supported"))

      and generate_array datatype expr_list llbuilder =
        match expr_list with
        | [h] -> generate_one_d_array datatype (expr_gen llbuilder h) llbuilder
        | _ ->  raise(Failure("Two dimentional array not supported"))
```

```
        (*| [h;s] -> generate_one_d_array datatype (expr_gen llbuilder h) (expr_gen
llbuilder s) llbuilder *)

        and generate_one_d_array datatype size llbuilder =
          let t = get_llvm_type datatype in

          let size_t = L.build_intcast (L.size_of t) i32_t "4tmp" llbuilder in
          let size = L.build_mul size_t size "5tmp" llbuilder in
          let size_real = L.build_add size (L.const_int i32_t 1) "arr_size" llbuilder
in

          let arr = L.build_array_malloc t size_real "6tmp" llbuilder in
          let arr = L.build_pointercast arr (pointer_type t) "7tmp" llbuilder in

          let arr_len_ptr = L.build_pointercast arr (pointer_type i32_t) "8tmp"
llbuilder in

          ignore(L.build_store size_real arr_len_ptr llbuilder);
          initialise_array arr_len_ptr size_real (const_int i32_t 0) 0 llbuilder;
          arr

        and initialise_array arr arr_len init_val start_pos llbuilder =
          let new_block label =
            let f = L.block_parent (L.insertion_block llbuilder) in
            L.append_block (context) label f
          in
          let bbcurr = L.insertion_block llbuilder in
          let bbcond = new_block "array.cond" in
          let bbbody = new_block "array.init" in
          let bbdone = new_block "array.done" in
          ignore (L.build_br bbcond llbuilder);
          L.position_at_end bbcond llbuilder;

          (* Counter into the length of the array *)
          let counter = L.build_phi [const_int i32_t start_pos, bbcurr] "counter"
llbuilder in
          add_incoming ((build_add counter (const_int i32_t 1) "tmp" llbuilder),
bbbody) counter;
          let cmp = build_icmp Icmp.Slt counter arr_len "tmp" llbuilder in
          ignore (build_cond_br cmp bbbody bbdone llbuilder);
          position_at_end bbbody llbuilder;
```

```
    (* Assign array position to init_val *)
    let arr_ptr = build_gep arr [| counter |] "tmp" llbuilder in
    ignore (build_store init_val arr_ptr llbuilder);
    ignore (build_br bbcond llbuilder);
    position_at_end bbdone llbuilder

  and generate_function_call fname expr_list d llbuilder =
    match fname with
    | "print" -> print_func_gen "" expr_list llbuilder
    | "println" -> print_func_gen "\n" expr_list llbuilder
    | _ ->  let f = find_func_in_module fname in
      let map_param_to_llvalue llbuilder e = match e with
        | SId(id, d) -> get_value true id llbuilder
        | _ -> expr_gen llbuilder e
      in
      let params = List.map (map_param_to_llvalue llbuilder) expr_list in (*Fix
passing variable to function*)
      L.build_call f (Array.of_list params) (fname^"_result") llbuilder

  and generate_object_create id el llbuilder =
    let f = find_func_in_module id in
    let params = List.map (expr_gen llbuilder) el in
    let obj = L.build_call f (Array.of_list params) "tmp" llbuilder in
    obj

  and binop_gen e1 op e2 llbuilder =
    let value1 =  match e1 with
      | SId(id, d) -> get_value true id llbuilder
      | _ -> expr_gen llbuilder e1
    in
    let value2 = match e2 with
      | SId(id, d) -> get_value true id llbuilder
      | _ -> expr_gen llbuilder e2
    in

    let int_binop value1 value2 llbuilder = (match op with
          Add     -> L.build_add
        | Sub     -> L.build_sub
        | Mult    -> L.build_mul
        | Div     -> L.build_sdiv
        | Equal   -> L.build_icmp L.Icmp.Eq
        | Neq     -> L.build_icmp L.Icmp.Ne
```

```
    | Less    -> L.build_icmp L.Icmp.Slt
    | Leq     -> L.build_icmp L.Icmp.Sle
    | Greater -> L.build_icmp L.Icmp.Sgt
    | Geq     -> L.build_icmp L.Icmp.Sge
    | And   -> L.build_and
    | Or    -> L.build_or
    | _     -> raise(Failure("Invalid operator for ints"))
  ) value1 value2 "binop_int" llbuilder
in

let float_binop value1 value2 llbuilder = (match op with
      Add     -> L.build_fadd
    | Sub     -> L.build_fsub
    | Mult    -> L.build_fmul
    | Div     -> L.build_fdiv
    | Equal   -> L.build_fcmp L.Fcmp.Oeq
    | Neq     -> L.build_fcmp L.Fcmp.One
    | Less    -> L.build_fcmp L.Fcmp.Ult
    | Leq     -> L.build_fcmp L.Fcmp.Ole
    | Greater -> L.build_fcmp L.Fcmp.Ogt
    | Geq     -> L.build_fcmp L.Fcmp.Oge
    | And   -> L.build_and
    | Or    -> L.build_or
    | _     -> raise(Failure("Invalid operator for ints"))
  ) value1 value2 "binop_float" llbuilder

in

let decide_on_type e1 e2 llbuilder =
  match ((Semant.typOfSexpr e1), (Semant.typOfSexpr e2)) with
  | (Int,Int) -> int_binop value1 value2 llbuilder
  | (Int, Float) -> float_binop value1 value2 llbuilder
  | (Float, Int) -> float_binop value1 value2 llbuilder
  | (Float, Float) -> float_binop value1 value2 llbuilder
  | (Int, _) -> int_binop value1 value2 llbuilder
  | (_, Int) -> int_binop value1 value2 llbuilder
  | (_, _) -> int_binop value1 value2 llbuilder

in

decide_on_type e1 e2 llbuilder
```

```
and unop_gen op e llbuilder =
  let exp_type = Semant.typOfSexpr e in
  let value = expr_gen llbuilder e in
  (match (op, exp_type) with
     (Not, Bool) -> L.build_not
   | (Neg, Int) -> L.build_neg
   | (Neg, Float) -> L.build_fneg
   | _ -> raise(Failure("Invalid unop usage"))
  ) value "tmp" llbuilder

and get_value deref vname llbuilder =
  if deref then
    let var = try Hashtbl.find global_var_table vname with
      | Not_found -> try Hashtbl.find local_var_table vname with
        | Not_found -> raise (Failure("unknown variable name " ^ vname))
    in
    L.build_load var vname llbuilder

  else
    let var = try Hashtbl.find global_var_table vname with
      | Not_found -> try Hashtbl.find local_var_table vname with
        | Not_found -> raise (Failure("unknown variable name " ^ vname))
    in
    var

and assign_to_variable e1 e2 llbuilder =
  let vmemory = match e1 with
    | SId(s, d) -> get_value false s llbuilder
  in
  let value = match e2 with
    | SId(id, d) -> get_value true id llbuilder
    | _ -> expr_gen llbuilder e2
  in
  L.build_store value vmemory llbuilder

and print_func_gen newLine expr_list llbuilder =
  let printf = find_func_in_module "printf" in
  let tmp_count = ref 0 in
  let incr_tmp = fun x -> incr tmp_count in
  let map_expr_to_printfexpr expr = match expr with
    | SId(id, d) -> get_value true id llbuilder (*(match d with
```

```
                                                                    | A.JBoolean
-> incr_tmp ();
                                                                        let
tmp_var = "print_bool" in
                                                                        let
trueStr = SString_Lit("true") in
                                                                        let
falseStr = SString_Lit("false") in
                                                                        let id
= SId(tmp_var, Arraytype(JChar, 1)) in

ignore(stmt_gen llbuilder (SLocalVarDecl(Arraytype(JChar, 1), tmp_var, SNoexpr)));

ignore(stmt_gen llbuilder (SIf(expr,

SExpr(SAssign(id, trueStr, Arraytype(JChar, 1)), Arraytype(JChar, 1)),

SExpr(SAssign(id, falseStr, Arraytype(JChar, 1)), Arraytype(JChar, 1)))));

expr_gen llbuilder id
                                                                    |
A.Arraytype(JChar, _) -> let llvalue = get_value true id llbuilder in

L.build_load llvalue "string" llbuilder

                                                                    | _ ->
get_value true id llbuilder)*)
            | SBoolLiteral(b) ->  if b then (expr_gen llbuilder (SStrLiteral("true")))
else (expr_gen llbuilder (SStrLiteral("false")))
            | _ -> expr_gen llbuilder expr
          in
          let params = List.map map_expr_to_printfexpr expr_list in
          let expr_types = List.map (typOfSexpr) expr_list in

          let map_expr_to_type e = match e with
              Int     ->  "%d"
            | Bool    -> "%s" (*needs to be implemented*)
            | Float   -> "%f"
            | Char     -> "%c"
            | _      -> raise (Failure("Print invalid type"))

          in
```

```
          let print_types = List.fold_left (fun s t -> s ^ map_expr_to_type t) ""
expr_types in
          let s = build_global_stringptr (print_types ^ newLine) "printf" llbuilder in

          (** let zero = const_int i32_t 0 in**)
          let s = build_in_bounds_gep s [| zero |] "printf" llbuilder in

          L.build_call printf (Array.of_list (s :: params)) "printf" llbuilder
        in

        (*Function generation*)

        let build_function sfunc_decl =
          Hashtbl.clear local_var_table;

          let f = find_func_in_module sfunc_decl.sfname in
          let llbuilder = L.builder_at_end context (L.entry_block f) in

          (*L.position_at_end (L.entry_block f) llbuilder; *)

          let init_formals f sfformals =
            let sfformals = Array.of_list (sfformals) in
            Array.iteri (
              fun i a ->
                let formal = sfformals.(i) in
                let allocatedMemory = stmt_gen llbuilder
(SLocalVariable(formal.sfvtyp, formal.sfvname, SNoexpr)) in
                let n = formal.sfvname in
                set_value_name n a;
                ignore (L.build_store a allocatedMemory llbuilder);
            )
              (params f)
          in
          let _ = init_formals f sfunc_decl.sfformals in
          let _ = stmt_gen llbuilder (SBlock (sfunc_decl.sfbody)) in
          if sfunc_decl.sftyp = Void
          then ignore (L.build_ret_void llbuilder);
          ()
        in
        let _ = List.map build_function functions in

        let build_constructors class_name =
```

```
          (*If a class has multiple constructors it will get overwritten at the
moment*)
          let build_constructor constructor =
            Hashtbl.clear local_var_table;

            let f = find_func_in_module class_name.scname in
            let llbuilder = L.builder_at_end context (L.entry_block f) in

            let struct_type = find_llvm_struct_type class_name.scname in
            let allocatedMemory = L.build_alloca struct_type "object" llbuilder in
            List.iteri (
              fun i f ->
                let expr = f.svexpr in
                let tuple_value = L.build_struct_gep allocatedMemory i "temp"
llbuilder in
                ignore(L.build_store (match expr with
                    | SId(id, d) -> get_value true id llbuilder
                    | _ -> expr_gen llbuilder expr) tuple_value llbuilder);

                set_value_name f.svname tuple_value;

                let access = f.svaccess in
                Hashtbl.add (match access with
                    | A.Public -> global_var_table
                    | A.Private -> class_private_vars) f.svname tuple_value;
            ) class_name.scbody.sfields;

            let pointer_to_class = L.build_pointercast allocatedMemory (L.pointer_type
struct_type) "tupleMemAlloc" llbuilder in

            let init_formals f sfformals =
              let sfformals = Array.of_list (sfformals) in
              Array.iteri (
                fun i a ->
                  let formal = sfformals.(i) in
                  let varMem = stmt_gen llbuilder (SLocalVariable(formal.sfvtyp,
formal.sfvname, SNoexpr)) in
                  let n = formal.sfvname in
                  set_value_name n a;
                  ignore (L.build_store a varMem llbuilder);
              )
```

```
            (params f)
        in
        let _ = init_formals f constructor.sfformals in
        let _ = stmt_gen llbuilder (SBlock (constructor.sfbody)) in

        L.build_ret pointer_to_class llbuilder
      in
      List.map build_constructor class_name.scbody.sconstructors in
    let _ =  List.map build_constructors classes in


    (*Main method generation*)
    let build_main main =
      let fty = L.function_type i32_t[||] in
      let f = L.define_function "main" fty the_module in
      let llbuilder = L.builder_at_end context (L.entry_block f) in

      let _ = stmt_gen llbuilder (SBlock (main.sfbody)) in

      L.build_ret (L.const_int i32_t 0) llbuilder
    in
    let _ = build_main main in

    (*Class generation *)

    let build_classes sclass_decl =
      let rt = L.pointer_type i64_t in
      let void_pt = L.pointer_type i64_t in
      let void_ppt = L.pointer_type void_pt in

      let f = find_func_in_module "lookup" in
      let llbuilder = L.builder_at_end context (entry_block f) in

      let len = List.length sclass_decl in

      let total_len = ref 0 in

      let scdecl_llvm_arr = L.build_array_alloca void_ppt (const_int i32_t len)
"tmp" llbuilder in

      let handle_scdecl scdecl =
        let index = try Hashtbl.find Semant.classIndices scdecl.scname with
```

```
            | Not_found -> raise (Failure("can't find classname" ^ scdecl.scname))
in
            let len = List.length scdecl.scbody.smethods in
            let sfdecl_llvm_arr = L.build_array_alloca void_pt (const_int i32_t len)
"tmp" llbuilder in

            let handle_fdecl i sfdecl =
              let fptr = find_func_in_module sfdecl.sfname in
              let fptr = L.build_pointercast fptr void_pt "tmp" llbuilder in

              let ep = L.build_gep sfdecl_llvm_arr [| (const_int i32_t i) |] "tmp"
llbuilder in
              ignore(L.build_store fptr ep llbuilder);
            in
            List.iteri handle_fdecl scdecl.scbody.smethods;
            total_len := !total_len + len;

            let ep = L.build_gep scdecl_llvm_arr [| (const_int i32_t index) |] "tmp"
llbuilder in
            ignore(build_store sfdecl_llvm_arr ep llbuilder);
          in
          List.iter handle_scdecl sclass_decl;

          let c_index = param f 0 in
          let f_index = param f 1 in
          L.set_value_name "c_index" c_index;
          L.set_value_name "f_index" f_index;

          if !total_len == 0 then
            L.build_ret (const_null rt) llbuilder
          else
            let vtbl = L.build_gep scdecl_llvm_arr [| c_index |] "tmp" llbuilder in
            let vtbl = L.build_load vtbl "tmp" llbuilder in
            let fptr = L.build_gep vtbl [| f_index |] "tmp" llbuilder in
            let fptr = L.build_load fptr "tmp" llbuilder in

            L.build_ret fptr llbuilder
        in
        let _ = build_classes classes in

        the_module;
```

12.2.7 Gassp.ml

```
(* Top-level gassp compiler, appropriated from microc *)

type action = Ast | Sast | LLVM_IR | Compile

let () =
 let action = ref Compile in
 let set_action a () = action := a in
 let speclist = [
   ("-a", Arg.Unit (set_action Ast), "Print the AST");
   ("-s", Arg.Unit (set_action Sast), "Print the SAST");
   ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
   ("-c", Arg.Unit (set_action Compile),
     "Check and print the generated LLVM IR (default)");
 ] in
 let usage_msg = "usage: ./gassp.native [-a|-s|-l|-c] [file.mc]" in
 let channel = ref stdin in
 Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;
  let lexbuf = Lexing.from_channel !channel in
 let ast = Gasspparser.program Scanner.token lexbuf in
 match !action with
   Ast -> print_string (Ast.string_of_program ast)
 | _ -> let sast = Semant.check ast in
   match !action with
     Ast     -> ()
   | Sast    -> print_string (Sast.string_of_sprogram sast)
   | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
   | Compile -> let m = Codegen.translate sast in
 Llvm_analysis.assert_valid_module m;
 print_string (Llvm.string_of_llmodule m)
```