Poicent Final Report

Shengtan Mao, sm4954; 4115 Spring 2021

github.com/shengtanmao/Poicent

Introduction

Poicent is a general purpose imperative language based on the C programming language. It expands on the MicroC language created by Dr. Stephen A. Edwards to add support for C-style pointers.

Poicent adds pointer types, malloc and free, pointer referencing and dereferencing, pointer subscript, and pointer arithmetic to MicroC. Together, these form the toolset to support pointer functionalities as presented in C. The pointer features greatly increase the flexibility of Poicent while still keeping it as a fairly minimal language.

Tutorial

Setup

Poicent uses the same environment as MicroC through Docker. The README file from MicroC is a good resource:

http://www.cs.columbia.edu/~sedwards/classes/2021/4115-spring/microc.tar.gz

The simplest way to set up the environment is to set up docker and work inside the preconfigured docker image for MicroC (columbiasedwards/plt). This is the command I use to run the image from zsh:

```
docker run --rm -it -v `pwd`:/home/poicent \
-w=/home/poicent columbiasedwards/plt
```

One could also use the included Dockerfile to build the same docker image. There are also directions in MicroC's README file for building the appropriate environment directly on the machine.

To compile the Poicent compiler, run

```
make
```

It will compile Poicent and run its tests.

Basic Tutorial

Poicent source code will end with ".pc". To compile and execute the file "<name>.pc", use

```
sh poicent.sh <name>
```

The program will start executing at the beginning of the "main" function, so every program must have a main function somewhere. For more details, read the Poicent Language Reference Manual.

Here is an example program that prints every integer from 1 to 10 to console:

```
int main()
{
	int a;
	for (a = 1; a <= 10; a = a + 1)
		print(a);
	return 0;
}
```

Poicent Language Reference Manual

Shengtan Mao, sm4954

Introduction

Pointers are powerful tools in C. They offer a much greater level of flexibility. MicroC is a minimalized version of the C language, which currently does not have support for pointers. The goal of this project is to add support for pointers and related operations, which include pointer types, referencing, dereferencing, pointer arrays, and pointer arithmetic.

Types

Integer

An integer is a sequence of digits interpreted as a decimal.

Float

A float consists of an integer part, a decimal point, followed by a fraction part. The integer and fraction parts both consist of a sequence of digits and interpreted as a decimal.

Boolean

A boolean takes the value of true or false.

Pointer

A pointer points to an object of a given type.

Void

As the return type of a function, void means the function does not return any value. As the object type for a pointer, void means the pointer points to an object of an unspecified type. Void type variables are not supported.

Lexical Conventions

There are five kinds of tokens: identifiers, keywords, constants, expression operators, and separators. Spaces, tab characters, and newline characters are ignored aside from however they may separate tokens.

Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`.

Identifiers

An identifier is a sequence of letters, digits, and underscore; the first character must be a letter. Upper and lower case letters are considered different.

Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
if              else
for             while
return          int
bool            float
void            true
false
```

Constants

Constants are the actual values of integer, float, and boolean types.

Expressions

The precedence of expression operators is the same as the order of the major subsections of this section with highest precedence first. Within each subsection, the operators have the same precedence.

Primary expressions

Primary expressions involving subscripting and function calls group left to right.

1.1 *identifier*

An identifier is a primary expression. Its type is specified by its declaration.

1.2 *constant*

A decimal, floating, or boolean constant is a primary expression.

1.3 (*expression*)

A parenthesized expression is a primary expression. Its type and value are identical to those of the unadorned expression.

1.4 *primary-expression* [*expression*]

This expression is the subscript; it is a primary expression. Usually, the primary expression has type ‘‘pointer to *type*’’, the subscript expression is `int`, and the type of the result is “*type*”.

1.5 *primary-expression* (*expression-list*$_{optional}$)

This expression is the function call; it is a primary expression. The expression-list contains a possibly empty, comma-separated list of expressions which constitute the arguments to the function. The primary expression must be of type "*type*", and the result of the function call is of type "*type*".

Unary operators

Expressions with unary operators group right-to-left.

An object is a manipulatable region of storage; an lvalue is an expression referring to an object.

2.1 `*` *expression*

The unary `*` operator means indirection. The expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is ‘‘pointer to *type*’’, the type of the result is ‘‘*type* ’’.

2.2 & *lvalue-expression*

The result of the unary & operator is a pointer to the object referred to by the lvalue-expression. If the type of the lvalue-expression is "*type*", the type of the result is ‘‘pointer to *type*’’.

2.3 `!` *expression*

The result of the logical negation operator `!` is true if the value of the expression is false, false if the value of the expression is true. The type of the result is `bool` and this operator is applicable only to `bool`s.

Multiplicative operators

The multiplicative operators `*` and `/` group left-to-right.

3.1 *expression* `*` *expression*

The binary `*` operator indicates multiplication. The two expressions must both be `int`s or both `float`s; the result is the same type as the two expressions.

3.2 *expression* `/` *expression*

The binary `/` operator indicates division. The same type considerations as for multiplication apply.

Additive operators

4.1 *expression1* + *expression2*

The result is the sum of the expressions. If the two expressions are both `ints` or both `floats`; the result is the same type as the two expressions. If expression 1 is a pointer and expression 2 is an `int`, the latter is converted by multiplying it by the length of the object to which the pointer points and the result is a pointer of the same type as the original pointer.

4.2 *expression* - *expression*

The result is the difference of the expressions. The same type considerations as for addition apply.

Relational operators

The relational operators group left-to-right

5.1 *expression* < *expression*

5.2 *expression* < *expression*

5.3 *expression* <= *expression*

5.4 *expression* >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield `false` if the specified relation is false and `true` if it is true. The two expressions must both be `ints`, both `floats`, or both pointers; the result is `bool`.

Equality operators

6.1 *expression* == *expression*

6.2 *expression* != *expression*

The == (equal to) and the != (not equal to) operators are analogous to the relational operators except for their lower precedence.

Operators

7 *expression* && *expression*

The && operator returns `true` if both its operands are true, `false` otherwise. && groups left-to-right; moreover the second operand is not evaluated if the first operand is false. The operands both have the type `bool`, and the result is `bool`.

8 *expression* `||` *expression*

The `||` operator returns `true` if either of its operands is true, `false` otherwise. `||` groups left-to-right; moreover the second operand is not evaluated if the first operand is true. The operands both have the type `bool`, and the result is `bool`.

9 *lvalue* = *expression*

The value of the expression replaces that of the object referred to by the lvalue. This operator groups right-to-left. Both operands must be the same type with the exception of void pointers. The void pointer is compatible with any pointer type.

Declarations

Declarations are used within function definitions to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. They have the form:

declaration:
type-specifier identifier ;

The type-specifiers are

type-specifier:
`int`
`float`
`bool`
`void`
type-specifier *

They indicate the type and storage class of the object to which the declarators refer. If a type-specifier has the form `type *`, then its identifier has the type ‘‘pointer to *type*’’, where ‘‘t*ype*’’ is the type which the identifier would have had if the type-specifier had been simply `type`.

Statements

Expression statement

These have the form

expression ;

Compound statement

Compound statements let several statements to be used where one is expected. They have the form

compound-statement:
{ statement-list }

statement-list:
statement
statement statement-list

Conditional statement

The two forms for the conditional statement are

`if` (*expression*) *statement*

`if` (*expression*) *statement* `else` *statement*

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the "else" ambiguity is resolved by connecting an `else` with the last encountered elseless `if`.

While statement

The while statement has the form

`while` (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

For statement

The `for` statement has the form

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

This statement is equivalent to

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

Return statement

A function returns to its caller by means of the return statement, which has one of the forms

`return ;`

`return` (*expression*) ;

In the first case no value is returned. In the second case, the value of the expression is returned to the caller of the function. Flowing off the end of a function is equivalent to a return with no returned value.

Global Definitions

A program consists of a sequence of global definitions. Global definitions may be given for declarations and function definitions. Declarations are covered in an earlier section.

Function definitions

Function definitions have the form

function-definition:
type-specifier function-declarator function-body

Where

function-declarator:
identifier (parameter-list$_{opt}$)
parameter-list:
formal-id
formal-id , *parameter-list*
formal-id:
type-specifier identifier
function-body:
{ declaration-list$_{opt}$ statement-list }
declaration-list:
declaration declaration-list

Declaration-list inside the function body has scope only inside the function.

Library Functions

There are a few global functions available to support print and memory operations.

Print

`void print(int val)`
Prints the integer value of val to console.

`void printb(bool val)`
Prints 1 to console if val is true, 0 if otherwise.

`void printf(float val)`
Prints the float value of val to console.

Memory

`void *malloc(int size)`
Allocates “size” bytes of memory. Returns the void pointer corresponding to the start of the allocated memory that will be automatically casted to the pointer type of the lvalue.

`void free(void *pointer)`
Frees the memory allocated for “pointer”. If the input is a pointer type, it will be automatically casted to void pointer.

Project Plan

Process

The planning for this project is unusual since I worked on it alone over the summer starting on May 2, 2021. Because I had a 14-week internship beginning May 10, I could not guarantee any deadlines as I wanted to prioritize my internship. I worked on the side for this project and came up with a timeline after my internship ended.

I used Github to maintain the codebase. Because I worked alone, I did not worry about a code review process before merging.

Informal testing is done at every stage of development (parser, ast, sast, llvm ir, program output). This is done by coming up with an example program that utilizes the functionalities, executing the compiler up to the particular level I am working on, and fixing any issues I found. After implementing all the features of Poicent, I created integration tests that target the new features of Poicent over MicroC. I then copied the tests already present in MicroC to Poicent to check the new features did not break MicroC's original functionalities.

The tests are automated to run when building the compiler with "make". One can also use "make test" to just run the tests. I also used Valgrind to check memory leaks for a few cases. These are not automated, but the Valgrind output is included in the appendix.

Programming Style

Poicent is done almost entirely in OCaml. Since Poicent is an expansion on MicroC, the goal is to adhere to the programming style present in MicroC. Here are some guidelines:

1. Basic format: this is done by the OCaml formatter ocamlformat (https://opam.ocaml.org/packages/ocamlformat/).
2. Comments: in addition to comments for explanations, comments may also indicate features added to MicroC.
3. Naming: type and function names should be descriptive. Variable names could be brief.
4. Naming conventions: Function and variable names should be lowercase with underscores; AST type names should be camelcase, and SAST type names should be the corresponding AST type name with 'S' attached before it.

Project Timeline

Date (2021)	Description
May 2	Started project
May 5	Submitted project proposal
May 7	Wrapped up AST and parser

May 9	Wrapped up semantic checker
Jun 20	Wrapped up code generator
Aug 11	Submitted language manual
Aug 16	Submitted codebase with basic tests (hello world)
Sep 3	Presentation and finish project

Roles and Responsibilities

Not applicable. Worked alone.

Software Development Environment

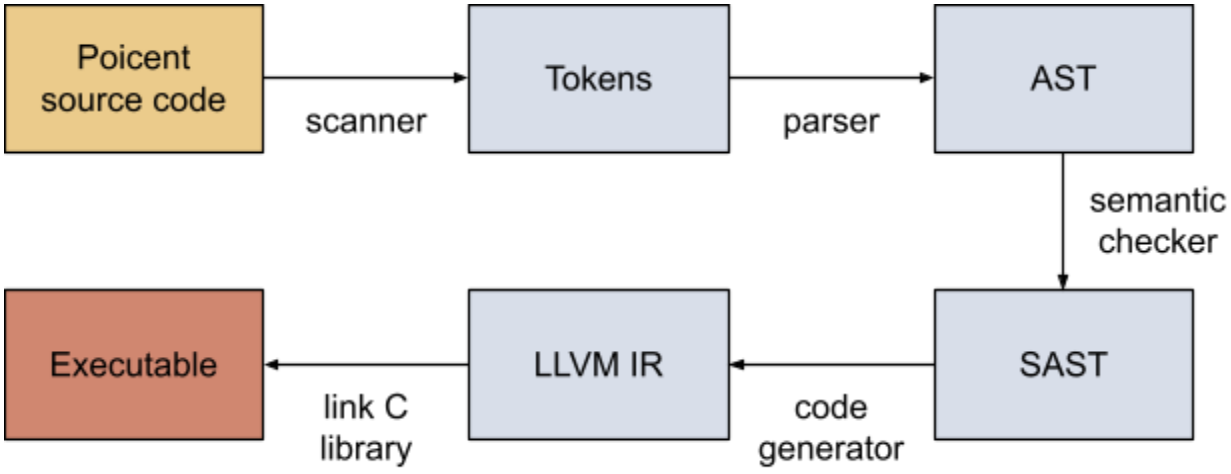
Operating System	Everything except text-editing and Valgrind is done through Ubuntu 20.04 Docker. Valgrind is done on Archlinux. Host machines: Archlinux, macOS.
Languages and Tools	OCaml and related tools, LLVM and related tools, clang, gcc, Valgrind.
Text Editor	NeoVim, VS Code.
Version Control	GitHub
Documents	Google Doc, LaTex.

Project Log

59 Github commits. See appendix for detail.

Architectural Design

Block Diagram



Components

I implemented all components of Poicent's additions to MicroC.

Scanner (scanner.mll)

The scanner takes in Poicent source code and builds the corresponding tokens. Comments and whitespace are ignored in the tokenization. The scanner catches illegal characters in the source code.

Parser (parser.mly) and AST (ast.ml)

The parser takes in tokens and builds the corresponding abstract syntax tree (AST). The parser outlines the syntax of Poicent and catches syntactical errors.

Semantic Checker (semant.ml) and SAST (sast.ml)

The semantic checker takes in the AST and builds the corresponding semantically checked AST (SAST). The SAST is an AST where the expressions have their types attached. The semantic checker ensures the expressions described by the AST are valid in the context of Poicent and will throw errors if not. This includes: ensuring variables and functions are declared before usage and do not have duplicate names; ensuring that expressions are valid type-wise.

Code Generator (codegen.ml)

The code generator takes in the SAST and builds the LLVM intermediate representation (LLVM IR). The code generator also declares global functions for external libraries (printf from C) so they could be linked during execution.

Test Plan

The test cases presented in the project are integration tests. Simple tests were used to ensure each component of Poicent worked as intended. For example, when I am working on the AST, I would come up with a test, run it up until the AST, and check that the result is what I wanted. Because my language is similar to C, I also used the LLVM IR created by clang from C as a resource. The unit tests are not presented in the submission. I did all the testing.

Test Suites

The feature tests can be split approximately into four categories: referencing and dereferencing, subscripts, pointer arithmetic, and MicroC tests. The MicroC tests are copied over from MicroC and will not be the focus of the report; they are there to ensure original features of MicroC still work.

For the MicroC tests, the printbig related tests are removed since that feature is dropped in Poicent.

For automation, one can run all tests by running “make test”; they also run when building the compiler using “make”. The testall.sh script is adapted from MicroC’s testall.sh script.

Poicent specific test files are available in the appendix. The original MicroC tests are only in the project files and are all passing. In the files, the new Poicent tests are prepended with -pc (example: test-pc-assign1.pc), but our discussion here omits the “-pc” for brevity.

I also used Valgrind to check memory leaks on a few test cases. This process is not automated. I generated the executable inside docker, and used Valgrind to run the executable directly on my Archlinux machine.

Referencing and Dereferencing

These tests include

`test-assign1`	
`test-ref_assign1`	`fail-ref_assign1`
`test-ref_assign2`	`fail-ref_assign2`
`test-func1`	
`test-func2`	

Test assign 1 checks setting a dereferenced pointer value. Test ref assign 1 and 2 check that a pointer can be set to a referenced non-pointer of the corresponding type. Fail ref assign 1 and 2 check that a pointer cannot be set to a referenced non-pointer if their types do not match. Test func 1 checks that functions can return various pointer types. Test func 2 checks that passing a pointer to a function and changing its dereferenced value works as intended.

Subscripts

These tests include

`test-sub_assign1`	`fail-sub_assign1`
`test-sub_assign2`	
`test-sub1`	

```
test-sub2
```

Test sub assign 1 checks that a value can be assigned to a pointer subscript; test sub assign 2 checks that a pointer subscript value can be assigned to a variable. Fail sub assign 1 checks that one cannot assign a value of a different type to a pointer subscript. Test sub 1 checks using a for loop to set multiple subscripts and then prints them out. Test sub 2 is similar to test sub 1 but checks two dimensional subscripts.

Pointer Arithmetic

These tests include

```
test-arith1          fail-arith1
test-arith2          fail-arith2
test-arith3
test-arith4
test-void1
test-cmp1            fail-cmp1
```

Test arith 1, 2, and 3 check that pointer addition / subtraction manipulates the pointer address correctly. Test arith 4 checks using pointer arithmetic to set multiple values and then prints them out. Fail arith 1 and 2 look at pointer addition / subtraction with invalid types. Test void 1 checks various void pointer manipulation works. Test cmp 1 checks that one can compare pointers based on their addresses. Fail cmp 1 checks that a pointer cannot be compared to an integer.

Others

There are some other tests that do not fit neatly into one of above categories:

```
test-edge1
test-edge2
fail-malloc1
fail-free1
```

Test edge 1 and 2 check combinations of pointer addition / subtraction and subscript work. Fail malloc 1 and free 1 checks that these functions throw errors if the argument type is invalid.

Valgrind

I used Valgrind to check these tests from before for memory leaks:

```
test-sub1
test-sub2
test-arith4
```

Test sub 1 and arith 4 deal with one-dimensional arrays, and test sub 2 deals with two-dimensional arrays. The Valgrind outputs are available in the appendix with the corresponding test cases. These cases are the most involved memory-wise, this should be a good sign that malloc and free are working as intended.

Examples

Creating an Array (subscript)

This example will create an array with values 1, 2, 3, 4, 5 and print them back in the same order. It will focus on using the subscript functionality. This example is the same as `test-pc-sub1.pc`.

Source Code:

```
int main()
{
    int *a;
    int i;

    a = malloc(5*4);

    for (i=0; i<5; i=i+1) {
        a[i] = i+1;
    }

    for (i=0; i<5; i=i+1) {
        print(a[i]);
    }

    free(a);
    return 0;
}
```

Output:

```
1
2
3
4
5
```

LLVM IR:

```
; ModuleID = 'Poicent'
source_filename = "Poicent"

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%g\0A\00"

declare i32 @printf(i8*, ...)
```

```
define i32 @main() {
entry:
  %a = alloca i32*
  %i = alloca i32
  %malloccall = tail call i8* @malloc(i32 mul (i32 ptrtoint (i1**
getelementptr (i1*, i1** null, i32 1) to i32), i32 20))
  %malloc = bitcast i8* %malloccall to i8**
  %vpcast = bitcast i8** %malloc to i32*
  store i32* %vpcast, i32** %a
  store i32 0, i32* %i
  br label %while

while:                                            ; preds =
%while_body, %entry
  %i7 = load i32, i32* %i
  %tmp8 = icmp slt i32 %i7, 5
  br i1 %tmp8, label %while_body, label %merge

while_body:                                       ; preds = %while
  %i1 = load i32, i32* %i
  %tmp = add i32 %i1, 1
  %a2 = load i32*, i32** %a
  %i3 = load i32, i32* %i
  %tmp4 = getelementptr inbounds i32, i32* %a2, i32 %i3
  store i32 %tmp, i32* %tmp4
  %i5 = load i32, i32* %i
  %tmp6 = add i32 %i5, 1
  store i32 %tmp6, i32* %i
  br label %while

merge:                                            ; preds = %while
  store i32 0, i32* %i
  br label %while9

while9:                                           ; preds =
%while_body10, %merge
  %i15 = load i32, i32* %i
  %tmp16 = icmp slt i32 %i15, 5
  br i1 %tmp16, label %while_body10, label %merge17

while_body10:                                     ; preds = %while9
  %a11 = load i32*, i32** %a
  %i12 = load i32, i32* %i
```

```
  %gep = getelementptr inbounds i32, i32* %a11, i32 %i12
  %deref = load i32, i32* %gep
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i32 %deref)
  %i13 = load i32, i32* %i
  %tmp14 = add i32 %i13, 1
  store i32 %tmp14, i32* %i
  br label %while9

merge17:                                          ; preds = %while9
  %a18 = load i32*, i32** %a
  %0 = bitcast i32* %a18 to i8*
  tail call void @free(i8* %0)
  ret i32 0
}

declare noalias i8* @malloc(i32)

declare void @free(i8*)
```

Creating an Array (pointer arithmetic)

This example will create an array with values 1, 2, 3, 4, 5 and print them back in reverse order. It will focus on using the pointer arithmetic functionality. This example is the same as `test-pc-arith4.pc`.

Source Code:

```
int main()
{
    int *a;
    int *a0;
    int i;

    a = malloc(5*4);
    a0 = a;
    for (i=1; i<=5; i=i+1) {
        *a = i;
        a = a + 1;
    }

    while (a>a0) {
        a = a - 1;
```

```
        print(*a);
    }

    free(a0);
    return 0;
}
```

Output:

```
5
4
3
2
1
```

LLVM IR:

```
; ModuleID = 'Poicent'
source_filename = "Poicent"

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%g\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main() {
entry:
  %a = alloca i32*
  %a0 = alloca i32*
  %i = alloca i32
  %malloccall = tail call i8* @malloc(i32 mul (i32 ptrtoint (i1**
getelementptr (i1*, i1** null, i32 1) to i32), i32 20))
  %malloc = bitcast i8* %malloccall to i8**
  %vpcast = bitcast i8** %malloc to i32*
  store i32* %vpcast, i32** %a
  %a1 = load i32*, i32** %a
  store i32* %a1, i32** %a0
  store i32 1, i32* %i
  br label %while

while:                                            ; preds =
%while_body, %entry
  %i7 = load i32, i32* %i
  %tmp8 = icmp sle i32 %i7, 5
  br i1 %tmp8, label %while_body, label %merge
```

```
while_body:                                       ; preds = %while
  %i2 = load i32, i32* %i
  %a3 = load i32*, i32** %a
  store i32 %i2, i32* %a3
  %a4 = load i32*, i32** %a
  %tmp = getelementptr inbounds i32, i32* %a4, i32 1
  store i32* %tmp, i32** %a
  %i5 = load i32, i32* %i
  %tmp6 = add i32 %i5, 1
  store i32 %tmp6, i32* %i
  br label %while

merge:                                            ; preds = %while
  br label %while9

while9:                                           ; preds =
%while_body10, %merge
  %a14 = load i32*, i32** %a
  %a015 = load i32*, i32** %a0
  %tmp16 = icmp sgt i32* %a14, %a015
  br i1 %tmp16, label %while_body10, label %merge17

while_body10:                                     ; preds = %while9
  %a11 = load i32*, i32** %a
  %tmp12 = getelementptr inbounds i32, i32* %a11, i32 -1
  store i32* %tmp12, i32** %a
  %a13 = load i32*, i32** %a
  %deref = load i32, i32* %a13
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i32 %deref)
  br label %while9

merge17:                                          ; preds = %while9
  %a018 = load i32*, i32** %a0
  %0 = bitcast i32* %a018 to i8*
  tail call void @free(i8* %0)
  ret i32 0
}

declare noalias i8* @malloc(i32)

declare void @free(i8*)
```

Lessons Learned

The three most important areas I learned about are: functional programming through OCaml, the C language through mimicking its pointer functionalities, and building a compiler.

This class is my introduction to functional programming. It is distinctly different from imperative programming, and I found it very interesting. Due to the nature of Poicent, all my work is done in OCaml (no external libraries), so I got a good amount of practice in functional programming.

There are a few ways I could have chosen to expand MicroC; I chose pointers because it's an area in C that I was not fully comfortable with. After working on the project, my knowledge of C pointers is much improved.

I also learned more about building a compiler and the different stages of translating from source code to LLVM IR. I have never thought much about compilers before this class, so it is cool that I now have insight about how this process works.

My advice for future teams is to set weekly deliverables for each member and make sure everyone contributes to the code base at the beginning of the project.

I will also address the split from my team. It's not ideal that I had to split and work on the project over the summer, but in the end, I feel like I have learned more working on a smaller project alone than I could have as a fairly contributing member of a team.

Near the beginning of the project, I wrote basically all of the language reference manual; they edited it to reflect the changes from the blueprint they made while coding. Looking back, I should have insisted that they work more on the LRM so I can work on the code base. Throughout the project, I was mostly given updates to what was done, but I should have asked the manager and the team to give me a timetable with frequent deliverables on what I need to do.

Appendix

Project Log (Github Commits)

```
commit 7a9d0dc91ec50cfcd6b886ada58b95fca786a3c2
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Fri Sep 3 12:55:58 2021 -0400

    tests: fail malloc and free

commit db2d0077cda0d9e070c89b80c501a37216bd53ac
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Fri Sep 3 12:33:59 2021 -0400

    tests: added return statement to func1

commit d50918f22b9c590517874c569dba967b46d05a5d
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Fri Sep 3 12:30:21 2021 -0400

    tests: more function tests

commit 2b146ee8a52b30d16de07586963f339c076e5745
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Thu Sep 2 20:01:33 2021 -0400

      tests: add pointer arithmetic to set array

commit 35b0570be7fdafed3aebaec3ec1dfda3c826dbba
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Thu Sep 2 18:16:39 2021 -0400

      tests: assign test

commit f085592bfda0ed9ae513988d9be9e1bf4ac644f1
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Thu Sep 2 17:37:41 2021 -0400

      formatting adjustments

commit ed7f389e017b80aa65778c03cc4cdb1c78d26acf
Author: Shengtan Mao <maoshengtan2011@gmail.com>
```

```
Date:   Thu Sep 2 16:58:28 2021 -0400

    tests: restored sub1 and renamed sub-multi

commit 392d9229404e35a0067f063ee74524fbf33f0e2b
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Thu Sep 2 16:52:28 2021 -0400

    codegen: fixed free bug

    tests: multi subscript test

commit 06f05b429ba5cde73c54e25690f4072932cbdf73
Author: Shengtan Mao <44749017+shengtanmao@users.noreply.github.com>
Date:   Thu Sep 2 15:56:19 2021 -0400

    Update README.md

commit decbc191fdfb31e02fee5328e8a3a855caff75f9
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Thu Sep 2 15:50:40 2021 -0400

    README: updated

commit 381a4db3d86fe42fbb73564b2bae8342ac8fe047
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Thu Sep 2 14:54:14 2021 -0400

    tests: tweaked test-pc-edge1

commit 8db501d64284abb02df1dbc570690453d41d33e2
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Thu Sep 2 14:25:24 2021 -0400

    tests: more tests

commit ca97f242cd8f553688bd02d7c06c37449f641e71
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Wed Sep 1 17:17:50 2021 -0400

    add script to run poicent
```

```
commit 7f5490cc9f5524dd2ee2e46b3c726acd11f61cd5
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Mon Aug 30 16:52:59 2021 -0400

    fixed more comments

commit e2d0a78cf46883e2471d3a1a305d373d21d7f3b3
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Mon Aug 30 16:50:37 2021 -0400

    adjusted comments

commit 036654aa2cfd4f1b383c62d3141dcea6dbfcb644
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Mon Aug 30 16:47:36 2021 -0400

    cleaned up code

commit 7d5a8d22cf3afb7b3ee40ff348952d694e0d5fe5
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Mon Aug 30 16:30:01 2021 -0400

    formatted code

commit 1e24d033023c5f00bc1f98ac6ddf5bd7a3dd6c6f
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Fri Aug 27 21:16:32 2021 -0400

    fixed test-func2 bug

commit 0304ac43b9be67c1e542c1ec5cdf0fe2c4835b2f
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Fri Aug 27 20:40:28 2021 -0400

    working on Makefile

commit 7ec06a983122d52c5b27da6a0fef671eaa9d6ad8
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Thu Aug 26 09:29:12 2021 -0400

    tests: fixed bugs; reorganized tests
```

```
commit 12d062df21456674540e6aae6f24cdfecb58b46c
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Thu Aug 26 01:07:23 2021 -0400

    *-tests: tweaked poicent tests; added microc tests

commit 729f0f4c2056f6cbe902cde20217120e3a8827fd
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Wed Aug 25 10:12:06 2021 -0400

    poicent-tests: add tests

commit 6401240f77f07348cc0e9cd57c911d58ed9722b0
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Mon Aug 23 20:02:03 2021 -0400

    codegen: fixed issue of free loading before branch

commit 1117fe0b893925c6af1ee38731f86fc59600d984
Merge: 75525b9 69c8008
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Sun Aug 22 16:07:53 2021 -0400

    Merge branch 'main' of github.com:shengtanmao/Poicent

commit 75525b9e569948daab5fe115f90e2decad3ef706
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Sun Aug 22 16:06:44 2021 -0400

    codegen: malloc, free does not call external func

commit 69c80084740b87af75dca94bf468cdc936dc07ea
Author: Shengtan Mao <44749017+shengtanmao@users.noreply.github.com>
Date:   Mon Aug 16 15:48:39 2021 -0400

    Update README.md

commit 81b4373a2c90548f113e6a9a44a29458ca78e629
Author: Shengtan Mao <44749017+shengtanmao@users.noreply.github.com>
Date:   Mon Aug 16 15:47:36 2021 -0400

    Update README.md
```

```
commit bbccfbc0a425ab9bc5e837c54b2e9064aa9969a8
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Mon Aug 16 15:44:29 2021 -0400

    removed Documents

commit 4ebd0921e6340763d287c7236d9084ea24bcd9ff
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Mon Aug 16 15:43:08 2021 -0400

    add basic tests

commit d56a5360c464b8ae6af7be622165146df17bcb57
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Mon Aug 16 14:56:01 2021 -0400

    fixed error messages and removed unused variables

commit 6aa2b15235b910b68e18fb43edf5996af357ab73
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Sun Aug 15 21:54:50 2021 -0400

    parser.mly: removed unused tokens

commit 75c62ddb7f12625cccf62602831d7abc6f960ab0
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Sun Jun 20 17:48:12 2021 -0400

    fixed issue with free

commit a7ae3eae7c23a93a1ab3a485f50f4242c33425ee
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Sat Jun 19 23:04:28 2021 -0400

    pointer comparison works

commit 9d0f0b28629c5a1c9d7c36c5e163fd3a10fec56e
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Sat Jun 19 14:29:40 2021 -0400

    Pointer add/sub done
```

```
commit d9bfc41f320295e30f21946cd34ed7d8d3ae37c1
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Sat Jun 19 14:06:24 2021 -0400

    fixed bug: functions after free not called

commit ecb70617d3671cb7cefabb6fbb58cf83fdb4c9f7
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Sat Jun 19 13:53:16 2021 -0400

    starting ptr arith

commit f6a23f16a8d8479ca9a871f4a0f87d82840f42ed
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Sun May 30 22:06:21 2021 -0400

    Added free

    malloc then free works

commit 7f76e4a55cd4022c4efd76a5bdd3113b7b02f4c2
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Sun May 30 13:30:14 2021 -0400

    Fixed bug with subscript

    same issue with assignment/general as deref

commit c2a1757011b24f41b3dee9c196aac1fcb365bd11
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Sun May 30 13:04:04 2021 -0400

    Fixed deref bug

    Moved the load instruction so now deref
    works for assign and in general

commit 1db9a9a50f84a3a0450a91383579b8de0415b5ec
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Sun May 30 11:28:09 2021 -0400
```

```
    malloc cast works, basic deref works

commit 3ce7bed597c9946861088ac6a1ef13730ebe8262
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Sat May 29 20:10:41 2021 -0400

    Added malloc

    still need casting when assigning

commit cf37f85826379d76d3a245ecdf103418199c7f10
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Sat May 29 12:42:46 2021 -0400

    Added referencing

commit 239c20b9b82eaca9ff7630d1e7436cdea1760906
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Fri May 28 19:02:05 2021 -0400

    Changed codegen void ptr

    added two scripts to run programs

commit 4c1ec5d1fb6c22469bab318e9278140456e6f349
Author: Shengtan Mao <maoshengtan2011@gmail.com>
Date:   Fri May 28 17:16:14 2021 -0400

    Deref seems to work

commit 14638a627a2295afad01fa09b222e6795e5b8379
Author: shengtanmao <maoshengtan2011@gmail.com>
Date:   Sun May 16 14:37:53 2021 -0400

    Basic indexing works
            modified:   codegen.ml
            modified:   poicent.ml

commit 70cb6b7ccc7c76499c20b334fa5932cc5416e10a
Author: shengtanmao <maoshengtan2011@gmail.com>
Date:   Sat May 15 12:23:15 2021 -0400
```

```
    Subscript changes compiles
            modified:   codegen.ml
            modified:   poicent.ml

commit c47c3d4308b29f4f40242c9a098f0d08391f6e87
Author: shengtanmao <maoshengtan2011@gmail.com>
Date:   Sun May 9 23:03:23 2021 -0400

    Working on codegen subscript assignment
            modified:   codegen.ml

commit c9e6b02b4d1a22656fbc0e76de495073710302c6
Author: shengtanmao <maoshengtan2011@gmail.com>
Date:   Sun May 9 12:01:28 2021 -0400

    Started codegen
            new file:   codegen.ml

commit d055f981160fb4b02e780a82797f9375612fa827
Author: shengtanmao <maoshengtan2011@gmail.com>
Date:   Fri May 7 18:14:49 2021 -0400

    Formatted ml files

commit fddd8a653329da45f4a83a42632ea35935e0b829
Author: shengtanmao <maoshengtan2011@gmail.com>
Date:   Fri May 7 18:07:27 2021 -0400

    Basic working semantic checker

    Works with the examples in tests
            modified:   ast.ml
            modified:   parser.mly
            modified:   poicent.ml
            modified:   sast.ml
            modified:   semant.ml

commit 3e504c873b05fc94140beb22a86a9e231f7583d4
Author: shengtanmao <maoshengtan2011@gmail.com>
Date:   Fri May 7 14:54:45 2021 -0400

    Working on semant
```

```
    Draft of semantically-checked expression for assign, unop, binop,
    subscript
            modified:   semant.ml

commit 2997031af00b19d76774c598ebcf9ac676de57f6
Author: shengtanmao <maoshengtan2011@gmail.com>
Date:   Thu May 6 19:29:10 2021 -0400

    Started semant
            new file:   semant.ml

commit 6c9f402c4816301220b788e8fef053b3d58afaac
Author: shengtanmao <maoshengtan2011@gmail.com>
Date:   Thu May 6 16:37:39 2021 -0400

    Added sast
            new file:   sast.ml

commit c65aca1f5812e19ec65a3111ef0091f1e7b7e6c5
Author: shengtanmao <maoshengtan2011@gmail.com>
Date:   Thu May 6 16:36:07 2021 -0400

    Removed inc / dec operations

    changed proposal code snippets to conform to microc.
    removed inc / dec ops
            modified:   Documents/proposal.tex
            modified:   ast.ml
            modified:   parser.mly
            modified:   tests/test-pointer-arith.pc

commit 536f5325572c761d74b9b4a44bd8f47ccbaf9d4f
Author: shengtanmao <maoshengtan2011@gmail.com>
Date:   Tue May 4 18:36:13 2021 -0400

    Good progress on parser and ast

    Should work for the current version of the proposal
            modified:   Documents/proposal.tex
            new file:   ast.ml
            modified:   parser.mly
```

```
            new file:   poicent.ml
            modified:   scanner.mll

commit d595c93d0cad4f263aea3089638acea75413b7b6
Author: shengtanmao <maoshengtan2011@gmail.com>
Date:   Tue May 4 16:03:40 2021 -0400

    Poicent parser and scanner
            modified:   parser.mly
            new file:   scanner.mll

commit 8768f9f48d08eb6bdff0299c9f8c277d5bebd7e7
Author: shengtanmao <maoshengtan2011@gmail.com>
Date:   Tue May 4 15:37:04 2021 -0400

    Added microc parser
            new file:   parser.mly

commit bc7474a0b32cc721957e558d0efcdd67e2f61aae
Author: shengtanmao <maoshengtan2011@gmail.com>
Date:   Mon May 3 16:18:37 2021 -0400

    Initial proposal
            new file:   Documents/proposal.tex

commit 757bae54ac5c59dc71dfca4817bc15dee653608f
Author: Shengtan Mao <44749017+shengtanmao@users.noreply.github.com>
Date:   Mon May 3 12:44:37 2021 -0400

    Initial commit
```

Poicent Compiler

All files authored by MicroC creator(s) and Shengtan Mao

scanner.mll

```
(* Ocamllex scanner for Poicent *)

{ open Parser }

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

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "bool"   { BOOL }
| "float"  { FLOAT }
| "void"   { VOID }
| "true"   { BLIT(true)  }
| "false"  { BLIT(false) }
| "&" {AMPER}
```

```
| '[' {LB}
| ']' {RB}
| digits as lxm { LITERAL(int_of_string lxm) }
| digits '.'  digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm { FLIT(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*     as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

parser.mly

```
%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA PLUS MINUS TIMES DIVIDE ASSIGN
%token NOT EQ NEQ LT LEQ GT GEQ AND OR
%token RETURN IF ELSE FOR WHILE INT BOOL FLOAT VOID
%token <int> LITERAL
%token <bool> BLIT
%token <string> ID FLIT
%token EOF
%token AMPER LB RB

%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
%right NOT AMPER
%left LB

%%

program:
  decls EOF { $1 }
```

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

fdecl:
   typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { typ = $1;
       fname = $2;
       formals = List.rev $4;
       locals = List.rev $7;
       body = List.rev $8 } }

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

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

typ:
    INT   { Int   }
  | BOOL  { Bool  }
  | FLOAT { Float }
  | VOID  { Void  }
  | typ TIMES {Pointer $1}

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

vdecl:
   typ 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)         }
```

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

expr:
    LITERAL          { Literal($1)            }
  | FLIT               { Fliteral($1)           }
  | BLIT             { BoolLit($1)            }
  | ID               { Id($1)                 }
  | expr PLUS   expr { Binop($1, Add,   $3)   }
  | expr MINUS  expr { Binop($1, Sub,   $3)   }
  | expr TIMES  expr { Binop($1, Mult,  $3)   }
  | expr DIVIDE expr { Binop($1, Div,   $3)   }
  | expr 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 AND    expr { Binop($1, And,   $3)   }
  | expr OR     expr { Binop($1, Or,    $3)   }
  | MINUS expr %prec NOT { Unop(Neg, $2)      }
  | NOT expr         { Unop(Not, $2)          }
  | TIMES expr %prec NOT { Deref $2           }
  | AMPER ID         { Refer $2               }
  | expr LB expr RB  { Subscript($1,$3)       }
  | expr ASSIGN expr { Assign($1, $3)         }
  | ID LPAREN args_opt RPAREN { Call($1, $3)  }
  | LPAREN expr RPAREN { $2                   }

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

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

ast.ml

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

type op =
  | Add
  | Sub
  | Mult
```

```
  | Div
  | Equal
  | Neq
  | Less
  | Leq
  | Greater
  | Geq
  | And
  | Or

type uop = Neg | Not

type typ = Int | Bool | Float | Void | Pointer of typ

type bind = typ * string

type expr =
  | Literal of int
  | Fliteral of string
  | BoolLit of bool
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of expr * expr
  | Call of string * expr list
  | Subscript of expr * expr
  | Refer of string
  | Deref of expr
  | Noexpr

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

type func_decl =
  { typ: typ
  ; fname: string
  ; formals: bind list
  ; locals: bind list
  ; body: stmt list }

type program = bind list * func_decl list

(* Pretty-printing functions *)
```

```
let string_of_op = function
  | Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

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

let rec string_of_expr = function
  | Literal l -> string_of_int l
  | Fliteral l -> l
  | BoolLit true -> "true"
  | BoolLit false -> "false"
  | 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) -> string_of_expr v ^ " = " ^ string_of_expr e
  | Call (f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Subscript (e, s) -> string_of_expr e ^ "[" ^ string_of_expr s ^ "]"
  | Refer s -> "&" ^ s
  | Deref e -> "*" ^ string_of_expr e
  | 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
```

```
let rec string_of_typ = function
  | Int -> "int"
  | Bool -> "bool"
  | Float -> "float"
  | Void -> "void"
  | Pointer t -> string_of_typ t ^ "*"

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

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^ fdecl.fname ^ "("
  ^ String.concat ", " (List.map snd fdecl.formals)
  ^ ")\n{\n"
  ^ String.concat "" (List.map string_of_vdecl fdecl.locals)
  ^ String.concat "" (List.map string_of_stmt fdecl.body)
  ^ "}\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)
```

sast.ml

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

open Ast

type sexpr = typ * sx

and sx =
  | SLiteral of int
  | SFliteral of string
  | SBoolLit of bool
  | SId of string
  | SBinop of sexpr * op * sexpr
  | SUnop of uop * sexpr
  | SAssign of sexpr * sexpr
  | SCall of string * sexpr list
  | SSubscript of sexpr * sexpr
  | SRefer of string
  | SDeref of sexpr
  | 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

type sfunc_decl =
  { styp: typ
  ; sfname: string
  ; sformals: bind list
  ; slocals: bind list
  ; sbody: sstmt list }

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"
    | 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) -> string_of_sexpr v ^ " = " ^ string_of_sexpr e
    | SCall (f, el) ->
        f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
    | SSubscript (e, s) -> string_of_sexpr e ^ "[" ^ string_of_sexpr s ^ "]"
    | SRefer s -> "&" ^ s
    | SDeref e -> "*" ^ string_of_sexpr e
    | 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

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)
```

semant.ml

```
(* Semantic checking for the Poicent compiler *)

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 (typ, name, ty) =
    StringMap.add name
      {typ; fname= name; formals= [(ty, "x")]; locals= []; body= []}
      map
  in
  List.fold_left add_bind StringMap.empty
    [ (Void, "free", Pointer Void)
    ; (Pointer Void, "malloc", Int)
    ; (Void, "print", Int)
    ; (Void, "printb", Bool)
    ; (Void, "printf", 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 *)
(* check if type is a pointer *)
let is_pointer p = match p with Pointer _ -> true | _ -> false in
let check_function func =
  (* Make sure no formals or locals are void or duplicates *)
  check_binds "formal" func.formals ;
  check_binds "local" func.locals ;
  (* Raise an exception if the given rvalue type cannot be assigned to
     the given lvalue type *)
```

```
    (* special handling for pointers *)
    let check_assign lvaluet rvaluet err =
      let typ =
        match lvaluet with
        | Pointer Void ->
            if is_pointer rvaluet then rvaluet else raise (Failure err)
        | Pointer _ ->
            if rvaluet = Pointer Void || rvaluet = lvaluet then lvaluet
            else raise (Failure err)
        | _ -> if lvaluet = rvaluet then lvaluet else raise (Failure err)
      in
      typ
    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.formals @ func.locals)
    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
    let deref p =
      match p with
      | Pointer s -> s
      | _ -> raise (Failure "cannot dereference expression")
    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)
      | Noexpr -> (Void, SNoexpr)
      | Id s -> (type_of_identifier s, SId s)
      | Assign (e1, e2) as ex ->
          let t1, e1' = expr e1 and t2, e2' = expr e2 in
          let err =
            "illegal assignment " ^ string_of_typ t1 ^ " = " ^ string_of_typ
t2
            ^ " in " ^ string_of_expr ex
          and vt =
            match e1 with
            | Id _ | Subscript (_, _) | Deref _ -> t1
            | _ -> raise (Failure "left expression is not assignable")
          in
```

```
          (check_assign t1 t2 err, SAssign ((vt, e1'), (t2, e2')))
      | 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
           * or when one is a pointer and the other is an int *)
          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 || is_pointer t1) ->
                Bool
            | (And | Or) when same && t1 = Bool -> Bool
            (* pointer addition and subtraction *)
            | (Add | Sub) when is_pointer t1 && t2 = Int -> t1
            | _ ->
                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.formals 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.formals args in
            (fd.typ, SCall (fname, args'))
      (* subscript main expr must be a pointer and the subscript must be
integer *)
      | Subscript (e, s) ->
          let te, e' = expr e and ts, s' = expr s in
          if ts != Int then raise (Failure "subscript expression not
integral")
          else
            let ts =
              match te with
              | Pointer p -> p
              | _ -> raise (Failure "main expression not a pointer")
            in
            (ts, SSubscript ((te, e'), (ts, s')))
      | Refer s -> (Pointer (type_of_identifier s), SRefer s)
      | Deref e ->
          let t, e' = expr e in
          if is_pointer t then (deref t, SDeref (t, e'))
          else raise (Failure "cannot dereference expression")
    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)
      | 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.typ then SReturn (t, e')
          else
```

```
            raise
              (Failure
                 ( "return gives " ^ string_of_typ t ^ " expected "
                 ^ string_of_typ func.typ ^ " 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.typ
    ; sfname= func.fname
    ; sformals= func.formals
    ; slocals= func.locals
    ; sbody=
        ( match check_stmt (Block func.body) with
        | SBlock sl -> sl
        | _ -> raise (Failure "internal error: block didn't become a
block?")
        ) }
  in
  (globals, List.map check_function functions)
```

codegen.ml

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR

LLVM tutorial: Make sure to read the OCaml version of the tutorial

http://llvm.org/docs/tutorial/index.html

Detailed documentation on the OCaml LLVM library:

http://llvm.moe/
http://llvm.moe/ocaml/

*)
```

```
module L = Llvm
module A = Ast
open Sast
module StringMap = Map.Make (String)

(* translate : Sast.program -> Llvm.module *)
let translate (globals, functions) =
  let context = L.global_context () in
  (* Create the LLVM compilation module into which
     we will generate code *)
  let the_module = L.create_module context "Poicent" 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 void_t = L.void_type context in
  let vpoint_t = L.pointer_type i8_t in
  (* Return the LLVM type for a Poicent type *)
  let rec ltype_of_typ = function
    | A.Int -> i32_t
    | A.Bool -> i1_t
    | A.Float -> float_t
    | A.Void -> void_t
    | A.Pointer p ->
        if p == A.Void then vpoint_t else L.pointer_type (ltype_of_typ p)
  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
        | A.Pointer p -> L.const_pointer_null (ltype_of_typ p)
        | _ -> 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
  (* 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 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
      | SNoexpr -> L.const_int i32_t 0
      (* need to allocate sizeof(type) to hold referenced var when declaring
poitners *)
      | SId s -> L.build_load (lookup s) s builder
      (* special handling for deref expr, subscript expr, and malloc *)
      | SAssign (e1, e2) ->
          let t1, s1 = e1 and _, s2 = e2 and e2'' = expr builder e2 in
          let e2' =
            match s2 with
            | SCall ("malloc", [_]) ->
                L.build_bitcast e2'' (ltype_of_typ t1) "vpcast" builder
            | _ -> e2''
          in
          let e =
            match s1 with
            | SId s ->
                ignore (L.build_store e2' (lookup s) builder) ;
                e2'
            | SSubscript (s, i) ->
                let e1' =
                  let s' = expr builder s and i' = expr builder i in
                  L.build_in_bounds_gep s' (Array.of_list [i']) "tmp"
builder
                in
                ignore (L.build_store e2' e1' builder) ;
                e2'
            | SDeref s ->
                let e1' = expr builder s in
                ignore (L.build_store e2' e1' builder) ;
                e2'
            | _ -> raise (Failure "error: failed to assign value")
          in
          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 ->
              raise
                (Failure
                   "internal error: semant should have rejected and/or on
float")
          )
            e1' e2' "tmp" builder
      | SBinop (((A.Int, _) as e1), op, e2)
       |SBinop (((A.Bool, _) as 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.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 )
            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.Not -> L.build_not )
            e' "tmp" builder
      (* pointer addition and subtraction *)
      | SBinop (s, op, ((A.Int, _) as i)) ->
          let s' = expr builder s in
          let i' =
            match op with
            | A.Add -> expr builder i
            | A.Sub -> expr builder (A.Int, SUnop (A.Neg, i))
            | _ -> raise (Failure "error: invalid pointer manipulation")
          in
          L.build_in_bounds_gep s' (Array.of_list [i']) "tmp" builder
      (* pointer comparison *)
      | SBinop (p1, op, p2) ->
          let p1' = expr builder p1 and p2' = expr builder p2 in
          ( match op with
          | 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
          | _ -> raise (Failure "error: invalid pointer comparison") )
            p1' p2' "tmp" builder
      | SSubscript (s, i) ->
          let e =
            let s' = expr builder s and i' = expr builder i in
            L.build_in_bounds_gep s' (Array.of_list [i']) "gep" builder
          in
          L.build_load e "deref" builder
      | SDeref s -> L.build_load (expr builder s) "deref" builder
      | SRefer s -> lookup s
      | SCall ("malloc", [e]) ->
          L.build_array_malloc vpoint_t (expr builder e) "malloc" builder
      | SCall ("free", [e]) ->
          L.build_free (expr builder e) builder
      | SCall ("print", [e]) | SCall ("printb", [e]) ->
          L.build_call printf_func
            [|int_format_str; expr builder e|]
            "printf" builder
      | SCall ("printf", [e]) ->
          L.build_call printf_func
            [|float_format_str; expr builder e|]
            "printf" 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 builder = function
      | SBlock sl -> List.fold_left stmt builder sl
      | SExpr e ->
          ignore (expr builder e) ;
          builder
      | SReturn e ->
          ignore
            ( match fdecl.styp with
            (* Special "return nothing" instr *)
            | A.Void -> L.build_ret_void builder
            (* 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 (L.builder_at_end context then_bb) then_stmt)
            build_br_merge ;
          let else_bb = L.append_block context "else" the_function in
          add_terminal
            (stmt (L.builder_at_end context else_bb) else_stmt)
            build_br_merge ;
          ignore (L.build_cond_br bool_val then_bb else_bb builder) ;
          L.builder_at_end context merge_bb
      | SWhile (predicate, body) ->
          let pred_bb = L.append_block context "while" the_function in
          ignore (L.build_br pred_bb builder) ;
          let body_bb = L.append_block context "while_body" the_function in
          add_terminal
            (stmt (L.builder_at_end context body_bb) body)
            (L.build_br pred_bb) ;
          let pred_builder = L.builder_at_end context pred_bb in
          let bool_val = expr pred_builder predicate in
          let merge_bb = L.append_block context "merge" the_function in
          ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder) ;
          L.builder_at_end context merge_bb
      (* Implement for loops as while loops *)
      | SFor (e1, e2, e3, body) ->
          stmt builder
            (SBlock [SExpr e1; SWhile (e2, SBlock [body; SExpr e3])])
    in
    (* Build the code for each statement in the function *)
    let builder = stmt builder (SBlock fdecl.sbody) in
    (* 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
```

poicent.ml

```
(* Top-level of the Poicent compiler: scan & parse the input,
   check the resulting AST and generate an SAST from it, generate LLVM IR,
   and dump the module *)

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: ./poicent.native [-a|-s|-l|-c] [file.pc]" 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 = Parser.program Scanner.token lexbuf in
  match !action with
  | Ast -> print_string (Ast.string_of_program ast)
  | _ -> (
      let sast = Semant.check ast in
      match !action with
      | Ast -> ()
      | Sast -> print_string (Sast.string_of_sprogram sast)
      (* ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis poicent.native
*)
      | 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) )
```

Poicent Tests

Referencing and Dereferencing

test-pc-assign1.pc	`void test_int()` `{` `    int *a;` `    a = malloc(4);` `    *a = 2;` `    print(*a);` `}` `void test_float()` `{` `    float *a;` `    a = malloc(8);` `    *a = 3.3;` `    printf(*a);` `}` `void test_bool()` `{` `    bool *a;` `    a = malloc(1);` `    *a = true;` `    printb(*a);` `}` `int main()` `{` `    test_int();` `    test_float();` `    test_bool();` `    return 0;` `}`
test-pc-assign1.out	`2` `3.3` `1`
test-pc-ref_assign1.pc	`void test_int()` `{` `    int *a;` `    int b;` `    b = 1;` `    a = &b;` `    print(*a);` `}`

	`` void test_float() { float *a; float b; b = 1.1; a = &b; printf(*a); } void test_bool() { bool *a; bool b; b = true; a = &b; printb(*a); } int main() { test_int(); test_float(); test_bool(); return 0; } ```
test-pc-ref_assign1.out	``` 1 1.1 1 ```
test-pc-ref_assign2.pc	``` int main() { int **a; int *b; int c; c = 1; b = &c; a = &b; print(**a); return 0; } ```
test-pc-ref_assign2.out	``` 1 ```
test-pc-func1.pc	``` int *int_p() { int *a; ```

```
    return a;
}

int **int_pp()
{
    int **a;
    return a;
}

float *float_p()
{
    float *a;
    return a;
}

float **float_pp()
{
    float **a;
    return a;
}

bool *bool_p()
{
    bool *a;
    return a;
}

bool **bool_pp()
{
    bool **a;
    return a;
}

int main()
{
    int *ip;
    int **ipp;
    float *fp;
    float **fpp;
    bool *bp;
    bool **bpp;

    ip = int_p();
    ipp = int_pp();
    fp = float_p();
    fpp = float_pp();
    bp = bool_p();
    bpp = bool_pp();
```

	`return 0;` `}`
test-pc-func1.out	
test-pc-func2.pc	`int set_one(int *a)` `{` `  *a = 1;` `}` `int main()` `{` `  int *a;` `  a = malloc(4);` `  set_one(a);` `  print(*a);` `  free(a);` `  return 0;` `}`
test-pc-func2.out	1
fail-pc-ref_assign1.pc	`int main()` `{` `  int **a;` `  int b;` `  b = 1;` `  a = &b;` `  print(*a);` `  return 0;` `}`
fail-pc-ref_assign1.err	Fatal error: exception Failure("illegal argument found int* expected int in *a")
fail-pc-ref_assign2.pc	`int main()` `{` `  int *a;` `  float b;` `  b = 1.1;` `  a = &b;` `  print(*a);` `  return 0;` `}`
fail-pc-ref_assign2.err	Fatal error: exception Failure("illegal assignment

	int* = float* in a = &b")

Subscripts

test-pc-sub_assign1.pc	void test_int() { int *a; a = malloc(2*4); a[1] = 2; print(a[1]); free(a); } void test_float() { float *a; a = malloc(2*8); a[1] = 2.2; printf(a[1]); free(a); } void test_bool() { bool *a; a = malloc(2*1); a[1] = true; printb(a[1]); free(a); } int main() { test_int(); test_float(); test_bool(); return 0; }
test-pc-sub_assign1.out	2 2.2 1
test-pc-sub_assign2.pc	int main() { int *a; int b; a = malloc(8);

	a[1] = 11; b = a[1]; print(b); free(a); return 0; }
test-pc-sub_assign2.out	11
test-pc-sub1.pc	int main() { int *a; int i; a = malloc(5*4); for (i=0; i<5; i=i+1) { a[i] = i+1; } for (i=0; i<5; i=i+1) { print(a[i]); } free(a); return 0; }
test-pc-sub1.out	1 2 3 4 5
sub1 valgrind	==12241== Memcheck, a memory error detector ==12241== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al. ==12241== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info ==12241== Command: ./test-pc-sub1.exe ==12241== 1 2 3 4 5 ==12241== ==12241== HEAP SUMMARY:

	==12241== in use at exit: 0 bytes in 0 blocks ==12241== total heap usage: 2 allocs, 2 frees, 4,256 bytes allocated ==12241== ==12241== All heap blocks were freed -- no leaks are possible ==12241== ==12241== For lists of detected and suppressed errors, rerun with: -s ==12241== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
test-pc-sub2.pc	int main() { int **a; int i; int j; a = malloc(5*8); for (i=0; i<5; i=i+1) { a[i] = malloc(5*4); for (j=0; j<5; j=j+1) { a[i][j] = (i+1) * (j+1); } } for (i=0; i<5; i=i+1) { for (j=0; j<5; j=j+1) { print(a[i][j]); } free(a[i]); } free(a); return 0; }
test-pc-sub2.out	1 2 3 4 5 2 4 6 8 10 3

	6 9 12 15 4 8 12 16 20 5 10 15 20 25
sub2 valgrind	==12308== Memcheck, a memory error detector ==12308== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al. ==12308== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info ==12308== Command: ./test-pc-sub2.exe ==12308== 1 2 3 4 5 2 4 6 8 10 3 6 9 12 15 4 8 12 16 20 5 10 15 20 25 ==12308==

	==12308== HEAP SUMMARY: ==12308== in use at exit: 0 bytes in 0 blocks ==12308== total heap usage: 7 allocs, 7 frees, 5,216 bytes allocated ==12308== ==12308== All heap blocks were freed -- no leaks are possible ==12308== ==12308== For lists of detected and suppressed errors, rerun with: -s ==12308== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
fail-pc-sub_assign1.pc	int main() { int *a; a = malloc(8); a[1] = 2.1; print(a[2]); free(a); return 0; }
fail-pc-sub_assign1.err	Fatal error: exception Failure("illegal assignment int = float in a[1] = 2.1")

Pointer Arithmetic

test-pc-arith1.pc	void test_int() { int *a; a = malloc(2*4); a[0] = 1; a[1] = 2; print(*(a+1)); free(a); } void test_float() { float *a; a = malloc(2*8); a[0] = 1.1; a[1] = 2.2; printf(*(a+1));

	free(a); } void test_bool() { bool *a; a = malloc(2*1); a[0] = true; a[1] = false; printb(*(a+1)); free(a); } int main() { test_int(); test_float(); test_bool(); return 0; }
test-pc-arith1.out	2 2.2 0
test-pc-arith2.pc	int main() { int *a; int *a0; a = malloc(12); a0 = a; a[0] = 1; a[1] = 2; a = a + 2+4-5; print(*a); free(a0); return 0; }
test-pc-arith2.out	2
test-pc-arith3.pc	int main() { int *a; int *a0;

	`    int b;` `    a = malloc(12);` `    a0 = a;` `    b = 3;` `    a[0] = 1;` `    a[1] = 2;` `    a[2] = 3;` `    a = a + 5;` `    a = a - b;` `    print(*a);` `    free(a0);` `    return 0;` `}`
test-pc-arith3.out	3
test-pc-arith4.pc	`int main()` `{` `  int *a;` `  int *a0;` `  int i;` `  a = malloc(5*4);` `  a0 = a;` `  for (i=1; i<=5; i=i+1) {` `  *a = i;` `  a = a + 1;` `  }` `  while (a>a0) {` `  a = a - 1;` `  print(*a);` `  }` `  free(a0);` `  return 0;` `}`
test-pc-arith4.out	5 4 3 2 1

arith4 valgrind	==12143== Memcheck, a memory error detector ==12143== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al. ==12143== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info ==12143== Command: ./test-pc-arith4.exe ==12143== 5 4 3 2 1 ==12143== ==12143== HEAP SUMMARY: ==12143== in use at exit: 0 bytes in 0 blocks ==12143== total heap usage: 2 allocs, 2 frees, 4,256 bytes allocated ==12143== ==12143== All heap blocks were freed -- no leaks are possible ==12143== ==12143== For lists of detected and suppressed errors, rerun with: -s ==12143== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
test-pc-void1.pc	int main() { void *a; void *a0; a = malloc(1); a0 = a; a = a + 13; a = a - 31; free(a0); return 0; }
test-pc-void1.out	
test-pc-cmp1.pc	int main() { void *a; void *b; b = a + 1;

	`if (a < b)` `print(1);` `else` `print(0);` `return 0;` `}`
test-pc-cmp1.out	1
fail-pc-arith1.pc	`int main()` `{` `void *a;` `a = a + 1.1;` `return 0;` `}`
fail-pc-arith1.err	Fatal error: exception Failure("illegal binary operator void* + float in a + 1.1")
fail-pc-arith2.pc	`int main()` `{` `void *a;` `a = 1 + a;` `return 0;` `}`
fail-pc-arith2.err	Fatal error: exception Failure("illegal binary operator int + void* in 1 + a")
fail-pc-cmp1.pc	`int main()` `{` `int *a;` `int b;` `if (a<b)` `print(1);` `return 0;` `}`
fail-pc-cmp1.err	Fatal error: exception Failure("illegal binary operator int* < int in a < b")

Others

test-pc-edge1.pc	`int main()` `{`

	`    int *a;` `    int *a0;` `    a = malloc(5*4);` `    a0 = a;` `    a[4] = 4115;` `    a = a - 1;` `    print(a[5]);` `    free(a0);` `    return 0;` `}`
test-pc-edge1.out	4115
test-pc-edge2.pc	`int main()` `{` `    int *a;` `    int *a0;` `    a = malloc(5*4);` `    a0 = a;` `    a[2] = 4115;` `    a = a + 4;` `    print(a[-2]);` `    free(a0);` `    return 0;` `}`
test-pc-edge2.out	4115
fail-pc-malloc1.pc	`int main()` `{` `    int *a;` `    a = malloc(2.3);` `    return 0;` `}`
fail-pc-malloc1.err	Fatal error: exception Failure("illegal argument found float expected int in 2.3")
fail-pc-free1.pc	`int main()` `{`

	`    free(1);` `    return 0;` `}`
`fail-pc-free1.err`	`Fatal error: exception Failure("illegal argument found int expected void* in 1")`