C* - Final Report

Authors

Name	UNI	Role
Shannon Jin	sj2802	Manager
Khyber Sen	ks3343	Language Guru
Ryan Lee	dbl2127	System Architect
Joanne Wang	jyw2118	Tester

Table of Contents

1. Introduction

C* is a general-purpose systems programming language. It is between the level of C and Zig on a semantic level, and syntactically it also borrows a lot from Rust (pun intended). It is meant primarily for programs that would otherwise be implemented in C for the speed, simplicity, and explicitness of the language, but want a few simple higher-level language constructs, more expressiveness, and some safety, but not so many overwhelming language features and implicit costs like in Rust, C++, or Zig.

It has manual memory management (no GC) and uses LLVM as its primary codegen backend, so it can be optimized as well as C, or even better in cases. All of C*'s higher-level language constructs are zero-cost, meaning none of those features give it any overhead over C, which often lead to a highly-optimized style where in C you would take less efficient shortcuts (e.x. function pointers and type-erased generics) and use dangerous constructs like goto. In the future, it may also have a C backend so that it can target any architecture where there is a C compiler.

While a general-purpose language, C* will probably have the most advantages when used in systems and embedded programming. It's expressivity and high-level features combined with its relative simplicity, performance, and explicitness is a perfect match for many of these low-level systems and embedded programs.

2. Language Tutorial

The environment used for C* is based in the docker file provided in MicroC.

To build:

$ sudo apt install ocaml llvm llvm-runtime m4 opam cmake pkg-config ocamlbuild
$ opam init
$ opam install llvm.10.0.0
$ eval $(opam env)
$ make

Writing a Cstar Program (.cs)

Simple Program Example:

```
fn int main()
{
  print(20+10);
  return 0;
}
```

To write a Cstar program in it's current stage:

Functions

- Functions must be declared as "fn" + return type + "name" + "(" + params + ")" + "{" + body + "}"
- Functions may or may not have a return statement depending on the type of the function

Comments

- Comments can either be multilined using "/**/".
- Comments can also be singled lined using "//".

3. Language Manual - What the Language Should Have Been

Introduction

C* is a general-purpose systems programming language. It is between the level of C and Zig on a semantic level, and syntactically it also borrows a lot from Rust (pun intended). It is meant primarily for programs that would otherwise be implemented in C for the speed, simplicity, and explicitness of the language, but want a few simple higher-level language constructs, more expressiveness, and some safety, but not so many overwhelming language features and implicit costs like in Rust, C++, or Zig.

It has manual memory management (no GC) and uses LLVM as its primary codegen backend, so it can be optimized as well as C, or even better in cases. All of C*'s higher-level language constructs are zero-cost, meaning none of those features give it any overhead over C, which often lead to a highly-optimized style where in C you would take less efficient shortcuts (e.x. function pointers and type-erased generics) and use dangerous constructs like goto. In the future, it may also have a C backend so that it can target any architecture where there is a C compiler.

While a general-purpose language, C* will probably have the most advantages when used in systems and embedded programming. It's expressivity and high-level features combined with its relative simplicity, performance, and explicitness is a perfect match for many of these low-level systems and embedded programs.

A C* Program

A C* program is a top-level C* module.

Note that italics will be used here to refer to
placeholders for language items, not the items themselves.

Modules

Every C* file (by default using a `.cstar` extension)

must be UTF-8.
Each file is implicitly a module, though modules can also be declared inline with the `mod` *`name`* `{}` keyword*.
Everything between the braces belongs to the module `name`.

A module is composed of a series of top-level items (aka declarations), which may be one of:

- `use`
- `let`
- `fn`
- `struct`
- `enum`
- `union`
- `impl`

These items may be proceeded by a single *`publicity`* modifier and any number of annotations.

Comments may also appear anywhere.

C* is not whitespace sensitive, i.e.,
any consecutive sequence of whitespace may be replaced by
any other consecutive sequence of whitespace
without changing the meaning of the program.
A unicode character is considered whitespace if it matches the `\p{Pattern_White_Space}` unicode property.

Identifiers

Identifiers in C* may be any UTF-8 string
in which the first characters is `_`, `$`, or matches the `\p{XID_Start}` unicode property,
and the remaining characters match the `\p{XID_Continue}` unicode property,
except for the following exceptions:

Identifiers may begin with `$` but are only definable by the compiler as intrinsics.

There are no keywords at the lexer level, but identifiers may not be a C* keyword.
They may also not be the boolean literals `true` or `false`.

`_` is a valid C* identifier at the syntactic level,
but has a special meaning and cannot be used everywhere.
That is, it can only be assigned to.

Examples:

```
// valid identifiers
let validWord: u32 = 2;
fn get_num() = {}
enum 小笼包 {}

// invalid identifier
let 2words = 2;
struct const {}
```

Keywords

Keywords are reserved identifiers that cannot be used as regular identifiers for other purposes.

C* keywords:

- `use`
- `let`
- `mut`
- `pub`
- `try`
- `const`
- `impl`
- `fn`
- `struct`
- `enum`
- `union`
- `return`
- `break`
- `continue`
- `for`
- `while`
- `if`
- `else`
- `match`
- `defer`
- `undefer`

There are also reserved keywords:

- `trait`

Comments

C* contains multiple types of comments

- single-line
- nested multi-line
- structural comments

`//` Single-Line Comments

Tokens followed by `//` until a `\n` newline are considered single-line comments.

`///` Doc Comments

Tokens followed by `///` until a `\n` newline are considered doc comments. They are a form of single-line comments,
but may also be processed by tools for generating documentation.

`/* */` Nested, Multi-Line Comments

Tokens followed by `/*` are considered multi-line comments.
They can be nested, and end at the next `*/` that is not a part
of an inner multi-line comment.

They also do not have to be multi-line,
and can comment out only part of a line.

`/-` Structural Comments

`/-` denotes a structral comment. It comments out the next item in the AST, which could be the next expression, function, type definition, etc.

Example:

```
// This is a regular single line comment.

/// This is a doc comment for the function below.
fn foo() = {}

/* This is a multiline comment
Everything inside here is commented out until "* /"
*/

/* They can be /* nested */, too. */
fn /* and appear in-between things */ bar() = {}

/- let x = 25; // This comments out the entire let expression.
```

`pub` Publicity

All top-level items (except `impl` blocks)
may be prefixed with a publicity modifier.

The syntax for this is `pub`.

Following the `pub`, there may also be a module path
within parentheses, like this: `(` *`path`* `)`.

If there is no publicity modifier, i.e. no `pub`,
then the publicity of the item is private, i.e. `pub(self)`.

Only public items may be `use`d from other modules.
Private items may only be used for the current module or its descendants.

Annotations

All items may be prefixed with any number of annotations,
which annotate the item with certain metadata.

The syntax for this is `@` *`annotation`*,
where *`annotation`* is the name of the annotation.
Note that annotations may be imported (`use`d)
or referred to with their fully-qualified path.

They may also have an *`argument_list`* after the annotation.
Having no *`argument_list`* is equivalent to having an empty, 0-length *`argument_list`*.
The *`argument_list`* is a normal C* *`argument_list`*,
except this one must be a compile-time constant.

The exact annotations available is still being decided,
but a few of them may be:

- `@extern`
- `@abi(" abi ")`, like `@abi("C")` or the default `@abi("C*")`
- `@inline`
- `@noinline`
- `@impl( type1 , ... , typeN )`
- `@align( alignment )`
- `@packed`
- `@allow(" warning_name ")`
- `@non_exhaustive`

For now, any available annotations will be implemented in the compiler, though this could change in the future.

Annotations can also be applied to the current module.
In this case, they must appear before any other items in the module and are prefixed with an extra `@`, like `@@allow("unused_variable")`.

`use` Declarations

`use` declarations are used to import items/declarations from other modules, such as the standard library, external libraries, your own defined modules, or certain types.

Their syntax is `use = use path`,
where `path = identitifier . path`.

That is, it imports a path to an item to be used without path qualification within the current scope.

`path` can also end in `.*`. The `*` indicates all items, so this imports all items from the parent path.

`let` s

A `let` binds an expression to a name.
That expression can either be a value or a type.

Normally (in expressions), `let` bindings can be shadowed, but they cannot be at the module level.

Value `let` s

For values, the syntax of this is `let mut ? identifier : type = expr ; ?`.

The `mut` is optional. If there is no `mut`,
then the variable is an immutable const.
If there is a `mut`, then it is a mutable global variable.

In normal `let` bindings, `expr` can be any C* expression,
and the `: type` may be omitted where inferrable,
but at the top, global level, the `expr` must be constant evaluated
and the `type` must be annotated.
The way to do the former is by using a `const { ... }` block,
which evaluates the block to a constant at compile time.

A value `let` can also create zero, one, or multiple bindings

at once through destructuring a pattern.
If the pattern is tautological, i.e. the pattern always matches,
then the bindings are always created.
If the pattern may not match, then the `let` expression is a `bool` and may be used in `if`s or `match`es.
In this case, the `let` binding(s) are only created if the pattern matches and the `let` expression evaluated to `true`.
Note that `match`ing a non-tautological `let` is possible
but very un-idiomatic, since the binding could simply be done in the match itself. Thus, it is normally used with `if`.

See pattern matching for more info on patterns and destructuring.

Type `let`s aka Type Aliases

For types, the syntax of this is `let` *`identifier generic_parameter_list?`* `=` *`type`* `;`.

The *`type`* here may be any type expression that
a value would be annotated with.
For example, this includes named types, tuples, arrays, slices, function pointers.

See below for info on the optional *`generic_parameter_list`*.

Note that this only creates an alias of the type,
but does not actually create a new type.
For example, the type alias cannot be used as a namespace
for methods or enum variants.

For example, you could have these type aliases:

```
let Option<T> = Result<T, ()>;
let Bool = Option<()>;
let Point = (f64, f64);
```

`fn` Function Declarations

`fn` declarations declare functions.

The syntax of this is `fn` *`identifier generic_parameter_list? parameter_list`* `:` *`type`* `=` *`expr`*.

The *`identifier`* is the name of the function,
the *`generic_parameter_list`* optional generic parameters,
the *`parameter_list`* required normal (non-generic) parameters,
the *`type`* the return type of the function,
and the *`expr`* the return value of the function.

Generic Parameters

A *`generic_parameter_list`* is delimited by `<` `>` angle brackets
and contains `,` comma-separated generic parameters.
A trailing comma is allowed.

Each generic parameter is a generic type or a generic constant*.
If it is a generic constant, then it requires a `:` *`type`* annotation.

Note that an empty *`generic_parameter_list`* like `<>`

is semantically distinct from no `generic_parameter_list` at all.
Generic functions are monomorphized (see generics for more).

Also, the `<` `>` angle brackets as used for generics
has higher precedence than the `<` `>` comparison operators.

Parameters

A `parameter_list` is delimited by `(` `)` parentheses
and contains a `,` comma-separated parameters.
A trailing comma is allowed.
Each parameter is a `let` binding except without the `let` keyword.
However, in function declarations, the parameters must have `:` `type` annotations.
Note that the similar function literals/values do not require this.

Return Type

The `:` `type` may be omitted if the type is the unit `()` type.

Return Value

The `expr` that the function returns may be any expression.
However, normally it is a `{ ... }` block,
which is necessary to include multiple statements in a function.
The block (like any) may also have modifiers,
like `try { ... }` or `const { ... }`.
Returning a `const { ... }` from a function in particular marks
that function as constant evaluatable*.

Normally a `;` is required to end the return value,
except if a block is used as the return value,
then it does not require the `;`.

A function return block is slightly special in that
`return` may be used within it, which is equivalent to a `break` from that top-level function block.

If a function is annotated with `@extern`,
then it must omit the `=` `expr` and end with a `;`.
In this case, only the function signature is specified
and the `@extern`ed function must be available as a function symbol at link time or else there will be a compile error.

Note that `@abi("C")` is usually specified along with `@extern`
because the default `@abi("C*")` is unstable.

In an `@extern @abi("C")` function,
the last (but not only) parameter may also be `...`,
which is a C varargs parameter and may be called with multiple arguments.
This is only for C FFI for functions like `syscall`,
which otherwise we'd need to implement with some assembly.

Note that `@extern` and `@abi("C")` may also be specified for an entire module,
in which case it applies to all items within that module.

Function Examples

For example, a non-generic function may look like this:

```
fn foo(_a: i32, b: usize, _c: String): usize = b * b;
```

or this:

```
fn string_len(c: String): usize = {
    c.len()
}
```

and a generic function may look like this:

```
fn equals<T>(a: T, b: T): bool = {
    a.equals(b)
}
```

`struct` Declarations

`struct` declarations declare a `struct` type,
which is a product type of its field types.
All fields are always initialized.

The syntax of this is `struct` *`identifier generic_parameter_list?`* `{` *`fields`* `}`,
where *`identifier`* is the name of the `struct` type,
`generic_parameter_list` are its generic parameters,
and *`fields`* is a `,` comma-separated list of fields.
A trailing comma is allowed.
Zero fields is also allowed.

The syntax of each field is a value `let` without the `let` and the `=` *`expr`* `;`.
Each field may also be prefixed by a *`publicity`* modifier.

Note that `mut` can be specified for these fields,
in which case they are have interior mutability,
i.e., they can be mutated through a non- `mut` pointer to the struct.

By default, `struct` s use `@abi("C*")`,
which means their layout and alignment is unspecified and unstable.
This allows for fields to be rearranged for optimizations.
If `@abi("C")` is specified, however, then the fields are
layed out in memory in the order they appear in,
and C alignment and padding rules are used.

`enum` Declarations

`enum` declarations declare an `enum` type,
which is a sum type of its variants.
That is, it is a discriminated union of variants,
each of which may have a value or not.
A value of an `enum` type is always one of its variants
and cannot be anything except those variants.
The discriminant value is stored.

The syntax of this is `enum` *`identifier generic_parameter_list?`* `{` *`variants`* `}`,
where *`identifier`* is the name of the `struct` type,
`generic_parameter_list` its generic parameters,

and `variants` is a `,` comma-separated list of variants.
A trailing comma is allowed.
Zero variants is also allowed, but note that this means that
the `enum` can never be instantiated because it has no variants.

Each variant may have a value or not.
If a variant does not have a value, then the syntax is `identifier`.
By default, the discriminant value of each variant is chosen by the compiler,
but this may be overridden for each variant
if all the variants of the `enum` have no value.
The syntax for this is `identifier` `=` `expr`,
where `expr` must be a `const { ... }` block
evaluating to the integer to be used for the discriminant.

If a variant does have a value, then the syntax is `identifier` `(` `type` `)`.
Note that only one `type` is allowed here.
If you wish to include multiple types,
simple use a tuple or `struct` instead.

All variants of an `enum` implicity use `pub` as their publicity modifier, which cannot be changed.

By default, `enum`s use `@abi("C*")`,
which means their layout and alignment is unspecified and unstable.
This allows for the layout, including the discriminant, to be optimized.
Generally, though, the size of an `enum` type is the
size of the discriminant plus the size of the largest variant data.

If all the variants have no values,
then `@abi("C")` may be specified.
In this case, you must also specify the size of the enum
by adding a `:` `type` following the `identifier` name,
where the `type` is a primitive integer type.
In this case, all the variant discriminants must fit within that type.

The `@non_exhaustive` attribute can also be applied to an `enum` type,
in which case matching all the variants is no longer considered an exhaustive match,
and a catch-all `_ =>` match arm is required.

`union` Declarations *

`union` declarations declare a `union` type,
which is a non-discriminated union similar to C `union`s.
It is meant for C FFI and thus defaults to `@abi("C")`.

The syntax of a `union` type declaration is
the same as a `struct` type declaration,
except the `struct` keyword is replaced by the `union` keyword.

The difference between the two is semantics.
The size of a union is the size of its largest field
and only one field may be active at any time.
Reading from an inactive field is undefined.

`impl` Blocks

`impl` blocks define associated items for a type, which includes methods.

The syntax for this is `impl` *`generic_parameter_list?`* *`type`* `{` *`items`* `}`,
where *`type`* is the type you are defining associated items for,
`generic_parameter_list` is any generic parameters needed for *`type`*, and *`items`* are items like those in a module.

Within an `impl` block, there is an implicit type alias defined:
`let Self =` *`type`* `;`, where *`type`* is the same type being `impl`emented.

Items defined within an `impl` block are available
through the type as if it were a module.
The exception is methods, which may be called in another way as well.
A method is a function in an `impl` block whose first parameter
is `self: Self`.
The `: Self` may be inferred (an exception for function declarations).
To call a method, you may also call it using `.` syntax on a value of the `impl` *`type`*.
That is, *`value`* `.` *`method`* `(` *`args`* `)` is syntactic sugar
for *`type`* `.` *`method`* `(` *`value`* `,` *`args`* `)` where *`value`* `:` *`type`*.

Type System

C* types can be split up into three kinds of types:

- primitive types
- compound types
  - built-in
  - user-defined

Primitive Types

The primitive types in C* are:

- the `()` unit type
- integer types
- float types
- the `char`acter type

`()` Unit Type

`bool` Type

`bool` is the boolean type in C*,
except it is actually defined as an enum:

```
@allow("non_title_case_types")
enum bool {
    false = const { 0 },
    true = const { 1 },
}
```

Normally operator overloading is not allowed in C*.
The exception is `bool`, which defines the normal boolean operators.
See operators for details on them.

Integer Types

Float Types

`char`acter Type

Built-In Compound Types

The built-in compound types in C* are:

- reference types
- slice types
- array types
- pointer types
- tuple types
- function types

Reference Types

In C*, you can have a reference to any type.
That reference is either immutable or mutable.

There is one exception to this.
`type.$bit_size_of()` must be a multiple of 8.
That is, bit fields like `u1` or `i5` may not be referenced.

The syntax for an immutable reference is `type&`,
and the syntax for a mutable reference is `type&mut`.

An immutable reference can be created using the postfix
`.&` reference operator from either an immutable or mutable binding.
A mutable reference can be created using the postfix
`.&mut` mutable reference operator, but only from a mutable binding.

Both immutable and mutable references can be dereferenced
using the postfix `.*` dereference operator.
This creates a temporary, unnamed, non-copied, immutable binding.
A mutable reference can also be dereferenced mutably
using the postfix `.*mut` mutable dereference operator.
This is the same as the `.*` deference operator,
except the resultant temporary is mutable.

Note that references can only be created by referencing an existing value.
Thus, null references are impossible to create.
Instead, `Option` should be used, like `Option<T&>`.

Table of Contents

Slice Types

In C*, you can also have a slice of a type, a contiguous collection of values of the same type. The number of values is only known at runtime.

The syntax for this is `type[]`.

A slice `T[]` is similar to the struct

```
struct SliceT {
    len: usize,
    ptr: T&,
}
```

but there are a few important differences.
Slices store their values inline.
They are thus unsized (i.e. dynamically sized) (`.$size_of()` is non- `const` for them).
However, references to slices are sized.
They are so-called fat pointers, i.e. the length and raw pointer both constitute the reference.

Slices are the only fundamentally unsized types.
Other compounds may only contain at most one unsized type,
and if they do, then they themselves are unsized.
Like slices, references to any unsized type are fat pointers.

To access the values of a slice,
the `[]` index operator may be used: `value` `[` `index` `]` ,
where `index` is a value of an unsigned integer type
and `value` is a reference to a value of slice type.
Note that if you have a slice reference,
it must be derefenced before indexing the slice directly.

Indexing a slice reference `T[]&` evaluates to `Result<T&, IndexBoundsError>` ,
and indexing a mutable slice reference `T[]&mut` evaluates to `Result<T&mut, IndexBoundsError>` .
Thus, it is always bounds checked.
To panic on an out-of-bounds index, simply `.unwrap()`
the `Result` to get the `T&` or `T&mut` ,
which can then be dereference to access.
To elimiate bounds checking, the `Result` can instead be `.unwrap_unchecked()` to get the `T&` or `T&mut`
without checking if there was an error,
thus eliminating the bounds check.

Bounds checking can also be eliminated in many other safe ways.
Bounds checking is usually only a problem when it is done for many elements of a slice when it only needs to be done once.
For this case, multiple elements can be indexed using a slice pattern (see patterns),
or an iterator can be used, which will eliminate redundant bounds checking.

Slices can also be sliced to yield a smaller view of the original slice.
This is also done by the same `[]` indexing operator,
except now the syntax is `value` `[` `range` `]` ,
where `range` is a value of range type.

Slicing a slice reference `T[]&` evaluates to `Result<T[]&, SliceBoundsError>` ,
and slicing a mutable slice reference `T[]&mut` evaluates to `Result<T[]&mut, SliceBoundsError>` .

Array Types

In C*, there also arrays of a type,
which, like slices, are a contiguous collection of values of the same type,
but unlike slices, have a length known at compile time and not stored at runtime.
Thus, they are sized unliked slices.

The syntax for this type is `type` `[` `size` `]` , where `size` is a const of an unsized integer type.

Arrays can also be indexed and sliced,
but since the length is known at compile time,
if the index or range is also known at compile time,
then indexing and slicing always succeeds at runtime
(i.e. there is no `Result`) yielding another array,
or else is a compile error.
The same syntax is used for indexing and slicing as is for slices.

To explicitly turn an array into a slice reference,
`.$cast<T[]>()` can be used.

Pointer Types

In C*, you can have a pointer to any type,
That reference is either immutable or mutable.

There is one exception to this.
`type` `.$bit_size_of()` must be a multiple of 8.
That is, bit fields like `u1` or `i5` may not be referenced.

The syntax for an immutable reference is `type` `*`,
and the syntax for a mutable reference is `type` `*mut`.

A pointer can point to 0, 1, or any number of the pointee type.

A pointer can only be created from
an explicit cast from a reference type
and through the return type of an `@extern` function.
It is just meant primarily for FFI.

A pointer cannot be dereferenced directly.
It must be explicitly cast to one of these types to be dereferenced:

- a reference if it points to 1 pointee type
- a slice if it points to any number of pointee types of runtime-known amount
- an array if it points to any number of pointee types of compile-time-known amount
- `None` if it is a null pointer

Tuple Types

In C*, you can also have a contiguous collection values of different types, i.e. a heterogenous array of sorts.
This is called a tuple and its length must be known at compile time.

The syntax for this type is `(` `types` `)`,
where `types` is a list of `,` comma-separated `type` s.
A trailing `,` comma is allowed.
However, in a single-element tuple, a trailing comma is required
to differentiate from general parentheses.

The elements of a tuple can be accessed as fields like in a `struct`.
In fact, a tuple is syntax sugar for an anonymous `struct`
with all public fields, though there is one caveat.
The fields of a tuple are decimal integer literals (the index),
which would not otherwise be allowed as an identifier for a field name.

Note that like `struct`s, tuple elements may be not layed out in memory in order.

Function Types

The type of a function `fn(a: A, b: B): C` is `fn(A, B): C`.

The syntax for this is `fn` *`tuple_type`* `:` *`type`*,
where *`tuple_type`* is a tuple type of the arguments
and *`type`* is the return type.

Other postfix type modifiers (e.x. `*`, `&`, `[]`)
applied at the end by default apply to the return type.
To apply them to the entire function type,
the function type must be parenthesized,
like `(fn(A): B)&`.

User-Defined Compound Types

The user-defined compound types in C* are:

- `struct` types
- `enum` types
- `union` types

They correspond to the item declarations of the same name.

`struct` Types

See `struct` declarations for more.

`enum` Types

See `enum` declarations for more.

`union` Types

See `union` declarations for more.

Destructive Moves

Passing a variable (to a function, to another variable, etc.)
are done by moving destructively.
That is, a simple `memcpy` to the new location.
There are no move constructors or anything like that.
Clones must be explicit with a `.clone()` call for `Clone` types (`@impl(Clone)`).
The exception is `Copy` types (`@impl(Copy)`),
for which clones are implicit.

Expressions

Almost everything that is not a type in C* is an expression.
This includes all control flow constructs.

Literals

C* Literals:

- unit
- bool
- int
- float
- char
- string
- struct
- tuple
- array
- enum
- union
- function
- closure
- range

Table of Contents

Unit Literals

In C*, every expression has a type. Even statements that return “nothing”,
they really return unit, or `()`.
The type of this unit literal is also called unit and written `()` as well.

Boolean Literals

There are two boolean literals of type `bool`: `true` and `false`.
These are actually enum variants of the `enum bool`.
See the `bool` Type.

Number Literals

In C*, number literals are composed of 4 (potentially optional) parts (in order):

- the integral part
- the floating part (optional)
- the exponent (optional)
- the suffix (optional)

For each of the integral part, floating part, and exponent,
they contain an optional sign, optional base,
and then a series of one or more digits.
Note that each part may specify a different base.

The sign may be `+` for positive numbers, `-` for negative numbers, or nothing, which defaults to `+`.

The base and corresponding digits may be:

Prefix	Name	Base	Digits
none	decimal	10	`0-9`
`0b`	binary	2	`0-1`
`0o`	octal	8	`0-8`
`0x`	hexadecimal	16	`0-9`, `A-F`

The series of digits may also be separated by
any number of `_` underscores between the digits.
It cannot begin or end with `_` underscores, however.

If there is a floating part, then a decimal point `.`
separates it from the preceeding integral part.
The floating part may not have a sign and is always positive (in itself).

If there is an exponent, then an `e` precedes it.

The (optional) suffix contains the type of number and a bit size.

The type of number may be:

- `u`: unsigned integer
- `i`: signed integer
- `f`: floating-point number

The bit size is usually a literal power of 2 number,
but may be any positive integer for integer types.
It may also be a word whose bit size is architecture-dependent.

For integers (`u` and `i`), the common bit sizes are:

- `8`
- `16`
- `32`
- `64`
- `128`
- `size` (bit size necessary to store an array index)
- `ptr` (bit size necessary to store a pointer
  or the difference between them)

For floats (`f`), the bit sizes are:

- `16`
- `32`
- `64`
- `128`

These suffixes are the primitive number types.
Thus, in total, they are (with their C equivalent for FFI):

C*	C
`u8`	`uint8_t`
`i8`	`int8_t`
`u16`	`uint16_t`
`i16`	`int16_t`
`u32`	`uint32_t`
`i32`	`int32_t`
`u64`	`uint64_t`
`i64`	`int64_t`
`u128`	`unsigned __int128`
`i128`	`__int128`
`usize`	`size_t`
`isize`	`ssize_t`
`uptr`	`uintptr_t`
`iptr`	`intptr_t`
`f16`	`_Float16`
`f32`	`float`
`f64`	`double`
`f128`	`_Float128`

Integers always use 2's-complement
and floats always are IEEE 754 floating point numbers.

If the type is a float, then it must contain
a `.` decimal point and a floating part.
If the type is an integer, then it must not.
Both can contain exponents, though for integers,
the exponent (in scientific notation) cannot cause
the integer to exceed its finite size.

If there is no suffix type, then the type is inferred.
If there is a `.` decimal point, then the type must be a float, and vice versa with integers.
If there is a `-` sign for the integral part,
then the type must be a float or a signed integer.
To infer the bit size of the number,
general type inference is used.
If it cannot be unambiguously inferred,
then it is an error and the user must
explicitly specify the suffix type.

Character Literals

In C*, character literals are of type `char` and are denoted with single `''` quotes.
They are [unicode scalar values](),
which are slightly different from [unicode code points]().

This means they are always 32 bits on all architectures.

For the actual char literal within the quotes,
it may be any unicode scalar value,
but some characters need to be or may be escaped.
The ascii values that must be escaped are:

- `\n` : newline
- `\r` : carriage return
- `\t` : tab
- `\0` : null char
- `\\` : backslash
- `\'` : single quote

Other ascii values may also be escaped as well using the syntax `\x7F` ,
where `7F` is the hexadecimal value of the ascii character,
from 0 to 127 (aka `0x7F`).
Thus it may only be two digits.

Unicode scalar values can also be escaped with the syntax `\u{7FFF}` .
The hexadecimal value is the 24-bit unicode character code.

Character literals can also be prefixed with a `b` : `b' '` ,
in which case they are byte literals, i.e. a `u8` .
The required ascii escapes are the same,
though the `\xFF` escape can now go up to 255 (aka `0xFF`),
and there may not be unicode escapes
(since it's only a `u8` byte literal now).

String Literals

There are multiple types of strings in C* owing to
the inherent complexity of string-handling without incurring overhead.
The default string literal type is `String` , which is UTF-8 encoded and
wraps a `*[u8]` . This is a borrowed slice type and can't change size.
To have a growable string, there is the `StringBuf` type,
but there is no special syntactic support for this owned string.
`String` s are made of `char` s, unicode scalar values, when iterating
(even though they are stored as `*[u8]`).

Then there are byte strings, which are just `*[u8]` and
do not have to be UTF-8 encoded.
String literals for this are prefixed with `b` , like `b"hello"` .
The owning version of this is just a `Box<[u8]>`
(notice the unsized slice use), and
the growable owning version is just a `Vec<u8>` .

Furthermore, for easier C FFI, there is also `CString` and `CStringBuf` ,
which are explicitly null-terminated. All other string types are
not null-terminated, since they store their own length,
which is way more efficient and safe.
Literal `CString` s have a `c` prefix, like `c"/home"` .

And finally, there are format strings. Written `f"n + m = {n + m}"` ,
they can interpolate expressions within `{}` .

Format, or `f` -strings, don't actually evaluate to a string,
but rather evaluate to an anonymous struct that has methods to
convert it all at once into a real string. Thus, `f` -strings do not allocate.

For the character literals allowed in C* strings,
that depends on the string type, which are:

Prefix	Name	Type
none	string	`String`
`b`	byte-string	`*[u8]`
`r`	raw-string	type without the `r`
`c`	c-string	`CString`
`f`	f-string	anonymous struct with methods

All of these string prefixes can be combined with each other,
except for `r` and `f` , since f-strings require escaping,
which goes against raw strings.

For `r` raw strings, no escapes are allowed.

For normal UTF-8 strings (which includes the `r` , `c` , and `f` modifiers),
the string must contain character literals, except there are no single `'` quotes anymore,
double `"` quotes delimit strings,
and double quotes must escaped (`\"`) instead of single quotes (`\'`).
Obviously the escapes don't apply to raw `r` strings.
For `f` -strings, braces must also be escaped: `\{` and `\}` ,
since they are used to delimit expressions within the string.
And for `c` -strings, they must not contains any `\0` null characters.

For byte `b` strings, the string must contains byte literals.
The other string modifiers apply in the same way,
and again, double quotes (`\"`) must be escaped instead of single quotes (`\'`).

Struct Literals

Struct literals are literals that create a value of a struct type.
That is, if we have a struct `Example` :

```
struct Example {
    a: u32,
    b: f64,
    c: String,
}
```

then we can create a value of type `Example` with the struct literal

```
Example {
    a: 0,
    b: 0.0,
    c: "",
}
```

That is, we first have the struct type name, an open `{` brace,

the list of fields and their values, and then a closing `}` brace.
The fields are separate by `,` commas (a trailing `,` comma is allowed),
and `:` colons separate the field name and its value.

If the name of a field and its value expression are the same,
then the `:` colon and value may be omitted, like so:

```
let c = "";
Example {
    a: 0,
    b: 0.0,
    c,
}
```

Furthermore, `..` can be used to spread the fields of another struct into a struct literal, like so:

```
struct SmallExample {
    a: u32,
    b: f64,
}

let x = SmallExample {
    a: 0,
    b: 0.0,
};

Example {
    ..x,
    c: "",
}
```

Note that the struct type does not have to be the same,
but the fields that are being spread must match between the struct types in name and type.

Tuple Literals

C* has tuples, but they are simply shorthand and syntax sugar for structs.
A tuple type is a finite, heterogenous list of types,
such as `(i32, usize, String)`,
and its field names are unsigned integers (`.0`, `.1`, and `.2` for this tuple).
This is the only difference between tuples and desugaring them to structs:
struct field names must be valid C* identifiers,
but tuple field names begin with digits.
Otherwise, they are exactly the same.
The tuple type with 0 element types, `()`, is also valid,
but it is equivalent to the `()` unit type.

Tuple literals mirror tuple types.
The field names are unnamed (unlike struct literals),
so it is just a `,` comma separated list of values of any type delimited by open `(` and close `)` parentheses.
There may be a trailing `,` comma separator,
and for 1-element tuple literals, this trailing `,` comma
is required to distinguish it from using `()` parentheses for associating general expressions.

Array Literals

In C*, arrays are finite, homogenous lists of a single type.
There are delimited by open `[` and close `]` brackets,
as opposed to `()` parentheses for tuples.
Their values are also `,` comma separated.
Trailing `,` commas are allowed but never required,
unlike in 1-element tuple literals.

Array types are denoted `[T; N]`, where `T` is any type
and `N: usize`.

Enum Literals

In an enum, such as

```
enum Example {
    A,
    B(i32),
}
```

there are two possible forms of enum literals
depending on if the variant has any data or not.

In the case of the variant `A`, which has no data attached,
the enum literal `Example.A` (or just `A` if `A` is imported)
is a value of type `Example`.

In the case of the variant `B`, which has data attached,
the enum literal `Example.B` is a function of type `fn(i32): Example`
that returns the `B` variant with the given data attached.
Thus, `Example.B(0)` or `Example.B(100)` is normally written,
though the function can also be referred to by itself.

Union Literals

Union literals are the same as struct literals
except only one field may be specified.

Function Literals

In C*, there is very little difference between function declarations
and function literals (using them as values).

In function declarations, they are written

```
PUBLICITY fn FUNC_NAME GENERIC_ARGS ARGS = BODY_EXPRESSION
```

such as

```
fn foo<T>(t: T): T = { t * t }
```

In function literals, there is no more publicity modifier
and the function name is optional,
since it usually specified as the let binding instead if named:

```
fn<T>(t: T): T = { t * t }
```

Furthermore, type inference of function arguments and return type
is allowed for function literals, since they cannot be public declarations.
If the types are ambiguous, though, type annotations are still required of course.

The type of a function literal is unique and opaque,
but can be casted to a function pointer like `fn(T): T`.

Note that annotations like `@abi("C")` can still be applied
to function literals just like function declarations.

Closure Literals

Closure literals are very similar to function literals—in fact,
they are a superset of function literals—except they also have a closure context.
That is, they can “enclose” over values in the current scope.

The syntax for a closure literal is simply a normal function literal with an anonymous struct literal, the closure context, following the `fn`.

The closure context is an anonymous struct literal
in that it has no named struct type. That is, instead of

```
Example {a: 0, b: 0.0, c: ""}
```

it would just be

```
{a: 0, b: 0.0, c: ""}
```

The fields in this closure context struct
are then immediately available within the function body
as if they were immediately destructured.

The type of a closure literal is unique and opaque.
Unlike function literals (in which there is no context),
the type of closure literals cannot be casted to a bare function pointer.
The closure function corresponds to a method on the closure context struct,
and as such, cannot be casted to a function pointer
since there is an implicit `*Self` argument.
Thus, the only way to accept a closure as an argument is by using generics,
which ensures there is no pointer indirection
and the closure can be inlined into the call site.

Range Literals

Range literals denote an integer range.
There are a few different forms of ranges,
which we will define in terms of set interval notation
as to what integers the range includes.
Here, `n` refers to the parent length that the range applies to.

Range	Interval
`a..b`	`[a, b)`
`a..`	`[a, n)`
`..b`	`[0, b)`
`..`	`[0, n)`
`a..=b`	`[a, b]`
`..=b`	`[0, b]`
`a..+b`	`[a, a + b)`
`a..+=b`	`[a, a + b]`
`a..-b`	`[a, n - b)`
`a..-=b`	`[a, n - b]`
`..-b`	`[0, n - b)`
`..-=b`	`[a, n - b]`

Function Calls

Method Calls

Blocks

Control Flow

Pattern Matching

Conditionals

`match`

`if`

`if` evaluates a block conditionally.

The syntax for this is `expr` `.if` `block`.
It is syntax sugar for a `match`:

`expr` `.match { true =>` `block` `, false => (), }`

`else`

An `else` may immediately follow an `if` expression,
in which case the whole thing becomes an if-else expression.

The syntax for this is `expr` `.if` `block` `else` `block`.
It is syntax sugar for a `match`:

`expr` `.match { true =>` `block` `, false =>` `block` `, }`,
where the `block` are in the same order as in the if-else expression.

Normally the `expr` following an `else` must be a `block`,
but it can also be another if expression.

Labels

Loops

`while`

`for`

A `for` loop allows you to iterate through an iterator.
An iterator is just a type `Iter` that has
a `fn next(self: Self) -> Option<T>` method,
where `T` is the element type we are iterating over.

The syntax for this is `*expr*.for *binding* *block*`,
where the `*expr*` is a value that has
a `.into_iter()` method returning the iterator,
the `*binding*` is the binding for the element name,
and `*block*` is the block of the `for` loop.

It is syntax sugar for:

`{ let iter = *expr*.into_iter(); true.while { let *binding* = iter.next().?; *block* } }`

`defer`

Error Handling

`try`

Panicking

In C*, all fallible functions and operations return either
`Result` or `Option` to indicate an error or exceptional case.
Normally errors are handled by bubbling up the error with `.?`
or handling the error directly in a `match` or other `Option`/`Result` methods.
However, in certain cases you either don't care about
handling the exceptional case or you can determine that
the error case is statically impossible but the compiler cannot.
In this case, you may wish to simply get the `Some` or `Ok` value
out of the `Option` or `Result`.
This can be done by panicking on a `None` or `Err`.

Panicking in C* means the program will immediately print out an error message
and then `abort`, i.e., calls the libc function `abort`.
No cleanup or unwinding is done in this case.
In particular, `defer`s on the stack are not run because the stack is not unwound.
Because of this, panicking should only be done under extreme circumstances,
such as statically determining the error case is impossible.
If you want unwinding and `defer`s to run,
simply use `.?` to bubble up the errors.

The way to panic is to call `.unwrap()` on a `Result`.
This is the only fundamental way to panic in C*.
All other functions that panic or may panic ultimately call `Result.unwrap`.
For example, `Option.unwrap` converts the `Option`
into a `Result` and then calls `.unwrap()` on it.

The same is true for `Option.expect` and `Result.expect`,
which allow you to set an error message to be printed.

The error message that `Result.unwrap` prints to `stderr` is implementation defined,
but it calls `E.error_message` to obtain the error message of the `e: E` in `Err(e)`.
Thus, to `.unwrap()` a `Result<T, E>`, `E` must have such a `.error_message()` method.
It may also print a (function call) stack trace or error return trace,
but that is not guaranteed.

There is one other option as well besides panicking.
If you know for certain that the error case is impossible,
you may call `Result.unwrap_unchecked()`.
This does not panic if the `Result` is `Err`,
but it is undefined behavior.

Operators

Operator	Arity	In-Place	Type	Description	Example
`+`	binary	no	arithmetic	addition	`2 + 2`, `4.0 + 2.0`
`-`	binary	no	arithmetic	subtraction	`2 - 2`, `4.2 - 2.2`
`*`	binary	no	arithmetic	multiplication	`2 * 2`, `4.0 * 2.0`
`/`	binary	no	arithmetic	division	`2 / 2`, `4.0 / 2.0`
`%`	binary	no	arithmetic	modulus	`2 % 2`
`-`	unary	no	arithmetic	negation	`-a`
`==`	binary	no	relational	equal to	`a == 2`
`!=`	binary	no	relational	not equal to	`a != 2`
`>`	binary	no	relational	greater than	`a > 2`
`<`	binary	no	relational	less than	`a < 2`
`>=`	binary	no	relational	greater than or equal to	`a >= 2`
`<=`	binary	no	relational	less than or equal to	`a <= 2`
`&&`	binary	no	logical	and	`a && b`
`||`	binary	no	logical	or	`a || b`
`!`, `.!`	unary	no	logical	not	`!a`
`&`	binary	no	bitwise	and	
`|`	binary	no	bitwise	or	
`^`	binary	no	bitwise	xor	
`~`, `.~`	unary	no	bitwise	not	
`<<`	binary	no	bitwise	left shift	
`>>`	binary	no	bitwise	right shift	
`[]`	binary	no	indexing	index a slice	`a[1]`

`[]`	binary	no	indexing	index a slice	`a[i]`
`-=`	binary	yes	arithmetic	subtraction	
`*=`	binary	yes	arithmetic	multiplication	
`/=`	binary	yes	arithmetic	division	
`%=`	binary	yes	arithmetic	modulus	
`&&=`	binary	yes	logical	and	
`||=`	binary	yes	logical	or	
`&=`	binary	yes	bitwise	and	
`|=`	binary	yes	bitwise	or	
`^=`	binary	yes	bitwise	xor	
`<<=`	binary	yes	bitwise	left shift	
`>>=`	binary	yes	bitwise	right shift	
`++`	unary	yes	arithmetic	increment	
`--`	unary	yes	arithmetic	decrement	
`.&`	unary	no	reference	reference	
`.&mut`	unary	no	reference	mutable reference	
`.*`	unary	no	reference	dereference	
`.*mut`	unary	no	reference	mutable dereference	
`.?`	unary	no	control flow	try	

Arithmetic operators operate on expressions of the same number type
and evaluate to the same number type as well.
`.$cast<>()` can be used here when the operands are of different type.
`%`, `++`, and `--` are not allowed for floats.

Relational operators operate on expressions of the same type
and evaluate to a `bool`.

Logical operators operate on `bool` expressions and evaluate to a `bool`.

Bitwise operators operate on expressions of the same number type
and evaluate to the same number type as well.
The except is the shift operators: `<<`, `>>`, `<<=`, and `>>=`,
whose right operand is the minimum unsigned integer type
that may be shifted by (i.e. the bit size of the left operand).
Otherwise it would be UB.
For example, if the left operand is `u64`, then the right operand is `u6`.
For signed integer types as the left operand,
the sign bit is extended when shifting.

For indexing operators, see slices
and arrays, which may be indexed.

In-place *`operator`*`=`s evalute to `()`.

Generics

Generics in C* are always monomorphized.

Constant Evaluation

Builtin Functions

Lang Types

Lang types are standard library types that the compiler knows about and may use.
They are:

- `Option`
- `Result`

For example, they are used for the `.?` try operator.

`Option`

```
enum Option<T> {
    Some(T),
    None,
}
```

`Result`

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

4. Project Plan

Our team originally planned to have weekly meetings with most of our communication over Discord. We began by specifying our goals for the project and how we would accomplish them. As we were creating the language, we realized we were overambitious without the amount of functionalities we wanted to include. This led to us having a session cutting down unnecessary features that could be implemented later. We began by creating each component of the compiler separately. We began by creating the scanner, then the ast, then the parser and so on. Although this may be a feasible way to approach the project, we realized that this was not the most efficient nor was it the easiest way to create the language. We found that it was better to attack the problem functionality by functionality which meant having the end-to-end setup and then slowly incorporating more features. We found that this was also an easier way to assign tasks and test the correctness of our compiler. Because of team and time management breakdowns, we decided to separate our project and complete it individually. Due to the limited time and the bad communication between our team members I decided to utilize MicroC as a starting off point so that I could then add and modify it to the features that Cstar has. When adding each new feature, I tested before merging with the other features. All the test cases are added to the testing suite which automatically tests using the test script.

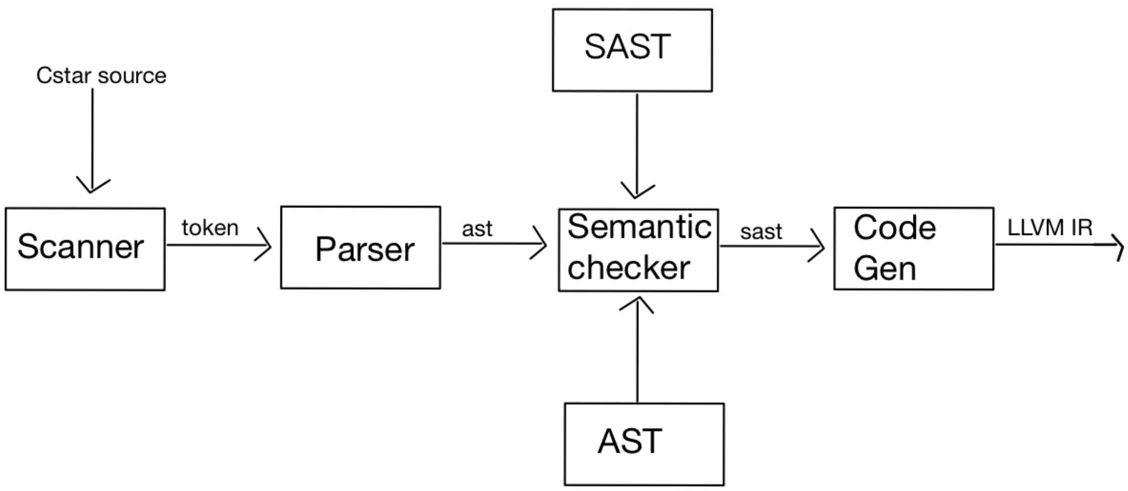
The software develompment tools used are as follows:
Libraries and Languages: Ocaml Version 4.11.1 and LLVM Version 10.0.0
Software: Visual Studio Code
OS: Ubuntu 20.04

Original Roles and Responsibilities

Name	Role
Shannon Jin	Manager
Khyber Sen	Language Guru
Ryan Lee	System Architect
Joanne Wang	Tester

5. Architectural Design

The block diagram for the Cstar compiler is shown below:



Scanner - scanner.mll

The scanner takes in a Cstar source program of ASCII inputs and generates tokens for identifiers, keywords, operators, and values. The commenting definition and logic is done through this step and ignores whitespace. The tokens are then sent the parser.

Parser - parser.mly

The parser takes in the generated tokens from the scanner and creates an abstract syntax tree (AST) which defines the context-free grammar for Cstar. The parser is implemented by Ocamlyacc and parses the token stream into a list which is then used match using the grammar.

Semantic Checker -semant.ml

The semantic checker traverses the AST and converts it into a semantically checked abstract syntax tree (SAST). The SAST has generally the same structure as the AST but with a type associated with it. It checks whether the source code is semantically correct including errors like duplicates or function declarations.

Code Generator - codegen.ml

The code generator takes in the SAST and generates code the LLVM IR. It traverses the SAST and turns each node into LLVM code in order to build the LLVM module. It also generates the code necessary to initialize a function instance. It is written using the OCaml LLVM library.

6. Test Plan

The test plan covers all the functionality that Cstar entails and tests different edge cases. An example test program is shown below along with its expected output. The automation is done using testall.sh so that it automatically compares the outcome and outputs and error if it differs. All tests are located in the cstar/tests folder and will generate either a .out or .err file. An example program test program and the output is shown below:

test-if2.cs

```
fn int main()
{
  if (false) print(20);
  print(10);
  return 0;
}
```

test-if2.out

10

7. Lessons Learned

This project has taught me a lot about not only the stages that go into creating a language and a compiler but more importantly the teamwork and the importance of having a well-balanced, well communicating team. I think our project started off on the wrong foot when only one member had a strong grasp and opinion of the functionalities we were building. Looking back I wish we had taken more ownership of what we were creating instead of relying one team member. I think this was also attributed to the fact that we felt that the person with each specific role should take charge of that section so we allowed more leniency on foreign parts of the project. However this was not beneficial to the team overall and led to a breakdown. I also learned that it's critical to set milestones throughout the semester and to stick to them even if there isn't a hard deadline. Having a strong team leader who manages and pushes the team is extremely important so that the team doesn't fall behind and push the project to the side. Overall, this project has taught me so much about working with other people when developing code. I realized that it's more important to emphasize teamwork and strong communication rather than indivitual abilities. It's also important to have an explicit breakdown of responsibilities when it comes to creating the compiler. When there are question it's critical to clarify them so that the entire team is on the same page and no duplicate work is done. Time management and laying out the project timeline at the very beginning is also a very necessary step that I believe will lead to sucess.

8. Appendix

Source Files

cstar.ml

```
(* Top-level of the Cstar 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: ./cstar.native [-a|-s|-l|-c] [file.cs]" 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)
    | 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)
```

scanner.mll

```
(* Ocamllex scanner for Cstar *)

{ open Parser }

let digit = ['0' - '9']
let digits = digit+
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let string_lit = '"'((ascii|escape)* as lxm)'"'

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) }
| '.'      { DOT }
| '?'      { QMARK }
| "=>"     { ARROW }
| "||"     { OROR }
| "&&"     { ANDAND }
| '@'      { AT }
| '%'      { PERCENT }
| "<<"     { LSHIFT }
| ">>"     { RSHIFT }
| "let"    { LET }
| "continue" { CONTINUE }
| "break"  { BREAK }
| "try"    { TRY }
| "match"  { MATCH }
| "defer"  { DEFER }
| "undefer" { UNDEFER }
| "in"     { IN }
| "mut"    { MUT }
| "use"    { USE }
| "fn"     { FN }
| "pub"    { PUB }
| ".."     { DOTDOT }
| "union"  { UNION }
| "enum"   { ENUM }
| "struct" { STRUCT }
| "impl"   { IMPL }
| "const"  { CONST }
| "string" { STRING }
| "trait"  { TRAIT }
| "//"     { scomment lexbuf } (* Single-line Comment *)

| 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) }
| string_lit { STRING_LITERAL(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

and scomment = parse
  "\n" { token lexbuf }
| _    { scomment lexbuf }
```

parser.mly

```
/* Ocamlyacc parser for Cstar */

%{
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 STRING
%token DOT QMARK ARROW OROR ANDAND AT PERCENT LSHIFT RSHIFT DOTDOT
%token LET CONTINUE BREAK TRY MATCH DEFER UNDEFER IN MUT USE PUB
%token FN UNION ENUM STRUCT IMPL CONST TRAIT
%token <int> LITERAL
%token <bool> BLIT
%token <string> ID FLIT
%token <string> STRING_LITERAL
%token EOF

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR OROR
%left AND ANDAND
%left LSHIFT RSHIFT
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS PERCENT
%left TIMES DIVIDE
%right NOT

%%

program:
  decls EOF { $1 }

decls:
   { ([], [])               }
 | decls vdecl { (($2 :: fst $1), snd $1) }
 | decls fdecl { (fst $1, ($2 :: snd $1)) }

fdecl:
   FN typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { typ = $2;
  fname = $3;
  formals = List.rev $5;
  locals = List.rev $8;
  body = List.rev $9 } }

formals_opt:
    { [] }
  | 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  }

vdecl_list:
    { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
   typ ID SEMI { ($1, $2) }

stmt_list:
    { [] }
  | 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)        }

expr_opt:
    { Noexpr }
  | expr          { $1 }

expr:
    LITERAL          { Literal($1)            }
  | FLIT      { Fliteral($1)           }
  | STRING_LITERAL   { StringLit($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)   }
  | expr ANDAND expr { Binop($1, BitAnd, $3)  }
  | expr OROR   expr { Binop($1, BitOr, $3)   }
  | expr LSHIFT expr { Binop($1, Lshift, $3)  }
  | expr RSHIFT expr { Binop($1, Rshift, $3)  }
  | expr PERCENT expr { Binop($1, Mod,  $3)   }
  | MINUS expr %prec NOT { Unop(Neg, $2)      }
  | NOT expr         { Unop(Not, $2)          }
  | ID ASSIGN expr   { Assign($1, $3)         }
  | ID LPAREN args_opt RPAREN { Call($1, $3)  }
  | LPAREN expr RPAREN { $2                   }

args_opt:
    { [] }
  | args_list  { List.rev $1 }

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

ast.ml

```
(* Abstract Syntax Tree *)

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq |
          And | Or | Mod | BitAnd| BitOr | Lshift | Rshift

type publicity = Public | Private

type mutability = {mut : bool}

type uop = Neg | Not

type typ = Int | Bool | Float | Void | Null | String

type bind = typ * string

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

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * 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 *)

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

```
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
  | StringLit(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
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""
  | Nullexpr -> "null"

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

let string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Float -> "float"
  | Void -> "void"
  | Null -> "null"
  | String -> "string"

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 *)

open Ast

type sexpr = typ * sx
and sx =
    SLiteral of int
  | SFliteral of string
  | SBoolLit of bool
  | SStringLit of string
```

```
  | SId of string
  | SBinop of sexpr * op * sexpr
  | SUnop of uop * sexpr
  | SAssign of string * sexpr
  | SCall of string * sexpr list
  | SNoexpr
  | SNullexpr

type sstmt =
    SBlock of sstmt list
  | SExpr of sexpr
  | SReturn of sexpr
  | SIf of sexpr * sstmt * 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 *)

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
  | SStringLit(s) -> s
  | 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 -> ""
  | SNullexpr -> "null"
      ) ^ ")"

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

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 Cstar compiler *)

open Ast
open Sast

module StringMap = Map.Make(String)

let check (globals, functions) =

  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_binds "global" globals;

  let built_in_decls =
    let add_bind map (name, ty) = StringMap.add name {
      typ = Void;
      fname = name;
      formals = [(ty, "x")];
      locals = []; body = [] } map
    in List.fold_left add_bind StringMap.empty [ ("print", Int);
                                ("printb", Bool);
                                ("printf", Float);
                                ("printbig", Int) ]
  in

  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
    in match fd with
         _ 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

  let function_decls = List.fold_left add_func built_in_decls functions
  in

  let find_func s =
    try StringMap.find s function_decls
    with Not_found -> raise (Failure ("unrecognized function " ^ s))
  in

  let _ = find_func "main" in

  let check_function func =
    check_binds "formal" func.formals;
    check_binds "local" func.locals;
```

```
    let check_assign lvaluet rvaluet err =
       if lvaluet = rvaluet then lvaluet else raise (Failure err)
    in

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

    let type_of_identifier s =
      try StringMap.find s symbols
      with Not_found -> raise (Failure ("undeclared identifier " ^ s))
    in

    let rec expr = function
        Literal  l -> (Int, SLiteral l)
      | Fliteral l -> (Float, SFliteral l)
      | BoolLit l  -> (Bool, SBoolLit l)
      | StringLit l -> (String, SStringLit l)
      | Noexpr     -> (Void, SNoexpr)
      | Nullexpr   -> (Null, SNullexpr)
      | 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
          let same = t1 = t2 in
          let ty = match op with
            Add | Sub | Mult | Div | Mod | Lshift | Rshift 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 | BitAnd | BitOr 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.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'))
```

```
          in (fd.typ, 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

    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)
      | 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))

      | 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)
            | s :: ss         -> check_stmt s :: check_stmt_list ss
            | []              -> []
          in SBlock(check_stmt_list sl)

    in
    { 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

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

module StringMap = Map.Make(String)

let translate (globals, functions) =
  let context    = L.global_context () in

  let the_module = L.create_module context "MicroC" in

  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 i8_pt      = L.pointer_type (L.i8_type context)
  and void_t     = L.void_type   context in

  let ltype_of_typ = function
      A.Int   -> i32_t
    | A.Bool  -> i1_t
    | A.Float -> float_t
    | A.Void  -> void_t
```

```
    | A.Null  -> i32_t
    | A.String -> i8_pt
  in

  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 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

  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

    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

      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

    let lookup n = try StringMap.find n local_vars
                   with Not_found -> StringMap.find n global_vars
    in

    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
      | SStringLit s -> L.build_global_stringptr s "tmp_string" builder
      | SNoexpr     -> L.const_int i32_t 0
      | SNullexpr   -> L.const_null i32_t
```

```
      | 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.BitAnd  -> L.build_and
          | A.BitOr   -> L.build_or
          | A.Mod     -> L.build_srem
          | A.Lshift  -> L.build_shl
          | A.Rshift  -> L.build_ashr
   | 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 (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.BitAnd  -> L.build_and
          | A.BitOr   -> L.build_or
          | A.Mod     -> L.build_srem
          | 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
   ) 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
      | SCall ("print", [e]) | SCall ("printb", [e]) ->
   L.build_call printf_func [| int_format_str ; (expr builder e) |]
     "printf" builder
      | SCall ("printbig", [e]) ->
   L.build_call printbig_func [| (expr builder e) |] "printbig" 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

    let add_terminal builder instr =
      match L.block_terminator (L.insertion_block builder) with
 Some _ -> ()
      | None -> ignore (instr builder) in

    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
                            A.Void -> L.build_ret_void builder
                          | _ -> 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

  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

    in

    let builder = stmt builder (SBlock fdecl.sbody) in

    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
```

Test Files

fail-assign1.cs

```
{
  int i;
  bool b;

  i = false; /* Fail: assigning a bool to an integer */
}
```

fail-assign1.err

```
Fatal error: exception Failure("illegal assignment int = bool in i = false")
```

fail-assign2.cs

```
fn int main()
{
  int i;
  bool b;

  b = 20; /* Fail: assigning an integer to a bool */
}
```

fail-assign2.err

```
Fatal error: exception Failure("illegal assignment bool = int in b = 20")
```

fail-assign3.cs

```
fn void voidfn()
{
  return;
}

fn int main()
{
  int i;

  i = voidfn(); /* Fail: assigning a void to an integer */
}
```

fail-assign3.err

```
Fatal error: exception Failure("illegal assignment int = void in i = voidfn()")
```

fail-assign4.cs

```
fn int main()
{
  int i = 4;

  i = null; /* Fail: assigning a null to an integer */
}
```

fail-assign4.err

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

fail-dead.cs

```
fn int main()
{
  int i = 2;
  return i;
  i = 32; /* Error: code after a return */
}
```

fail-dead.err

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

fail-expr.cs

```
fn void foo(int a, bool b)
{
  a + b; /* Error: int + bool */
}

fn int main()
{
  foo(2, true)
  return 0;
}
```

fail-expr.err

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

fail-func1.cs

```
fn int foo() {}

fn int bar() {}

fn int baz() {}

fn void bar() {} /* Error: duplicate function bar */

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

fail-func1.err

```
Fatal error: exception Failure("duplicate function bar")
```

fail-func2.cs

```
fn int foo(int a, bool b, int c) { }

fn void bar(int a, void b, int c) {} /* Error: illegal void formal b */

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

fail-func2.err

```
Fatal error: exception Failure("illegal void formal b")
```

fail-func3.cs

```
fn void foo(int a, bool b)
{
}

fn int main()
{
  foo(20, true);
  foo(20); /* Wrong number of arguments */
}
```

fail-func3.err

```
Fatal error: exception Failure("expecting 2 arguments in foo(20)")
```

fail-func4.cs

```
fn void foo(int a, bool b)
{
}

fn void bar()
{
}

fn int main()
{
  foo(20, true);
  foo(20, bar()); /* int and void, not int and bool */
}
```

fail-func4.err

```
Fatal error: exception Failure("illegal argument found void expected bool in bar()")
```

fail-func5.cs

```
fn void foo(int a, bool b)
{
}

fn int main()
{
  foo(20, true);
  foo(20, 20); /* Fail: int, not bool */
}
```

fail-func5.err

```
Fatal error: exception Failure("illegal argument found int expected bool in 20")
```

fail-if1.cs

```
fn int main()
{
  if (20) {} /* Error: int not bool type */
}
```

fail-if1.err

```
Fatal error: exception Failure("expected Boolean expression in 20")
```

fail-if2.cs

```
fn int main()
{
  if (true) {
    a; /* Error: undeclared variable */
  }
}
```

fail-if2.err

```
Fatal error: exception Failure("undeclared identifier a")
```

fail-nomain.cs

```

```

fail-nomain.err

```
Fatal error: exception Failure("unrecognized function main")
```

fail-return.cs

```
fn int main()
{
  return true; /* Should return int */
}
```

fail-return.err

```
Fatal error: exception Failure("return gives bool expected int in true")
```

test-add.cs

```
fn int main()
{
  print(20+10);
  return 0;
}
```

test-add.out

```
30
```

test-func1.cs

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

fn int main()
{
  int a;
  a = add(20, 30);
  print(a);
  return 0;
}
```

test-func1.out

```
50
```

test-func2.cs

```
fn void printall(int a, int b, int c, int d)
{
  print(a);
  print(b);
  print(c);
  print(d);
}

fn int main()
{
  printall(10, 20, 30, 40);
  return 0;
}
```

test-func2.out

```
10
20
30
40
```

test-func3.cs

```
fn int sub(int a, int b)
{
  int c;
  c = a - b;
  return c;
}

fn int main()
{
  int d;
  d = sub(30, 20);
  print(d);
  return 0;
}
```

test-func3.out

```
10
```

test-func4.cs

```
fn int foo(int a)
{
  return a;
}

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

test-func4.out

```
```

test-func5.cs

```
int a;

fn void foo(int x)
{
  a = x + 20;
}

fn int main()
{
  foo(20);
  print(a);
  return 0;
}
```

test-func5.out

```
40
```

test-global1.cs

```
bool i;

fn int main()
{
  int i; /* Should hide the global i */

  i = 20;
  print(i + i);
  return 0;
}
```

test-global1.out

```
40
```

test-if1.cs

```
fn int main()
{
  if (true) print(20);
  print(10);
  return 0;
}
```

test-if1.out

```
20
10
```

test-if2.cs

```
fn int main()
{
  if (false) print(20);
  print(10);
  return 0;
}
```

test-if2.out

```
10
```

test-local.cs

```
### fn void foo(bool i)
{
  int i; /* Should hide the bool i */

  i = 20;
  print(i + i);
}

fn int main()
{
  foo(true);
  return 0;
}
```

test-local.out

```
40
```

test-num.cs

```
fn int main()
{
  print(20);
  return 0;
}
```

test-num.out

```
20
```

test-ops1.cs

```
fn int main()
{
  print(1 + 2);
  print(1 - 2);
  print(1 * 2);
  print(1 % 2);
  print(100 / 2);
  print(99);
  printb(1 == 2);
  printb(1 == 1);
  print(99);
  printb(1 != 2);
  printb(1 != 1);
  print(99);
  printb(1 < 2);
  printb(2 < 1);
  print(99);
  printb(1 <= 2);
  printb(1 <= 1);
  printb(2 <= 1);
  print(99);
  printb(1 > 2);
  printb(2 > 1);
  print(99);
  printb(1 >= 2);
  printb(1 >= 1);
  printb(2 >= 1);
  /*print(99);
  printb(1 || 2);
  printb(1 || 1);
  printb(2 || 1);
  print(99);
  printb(1 && 2);
  printb(1 && 1);
  printb(2 && 1);
  print(99);
  printb(1 << 2);
  printb(1 << 1);
  printb(2 << 1);
  print(99);
  printb(1 >> 2);
  printb(1 >> 1);
  printb(2 >> 1);
  return 0;*/
}
```

test-ops1.out

```
3
-1
2
1
50
99
0
1
99
1
0
99
1
0
99
1
1
0
99
0
1
99
0
1
1
```

test-ops2.cs

```
fn int main()
{
  printb(true);
  printb(false);
  printb(true & true);
  printb(true & false);
  printb(false & true);
  printb(false & false);
  printb(true | true);
  printb(true | false);
  printb(false | true);
  printb(false | false);
  printb(!false);
  printb(!true);
  print(-10);
  print(--42);
}
```

test-ops2.out

```
1
0
1
0
0
0
1
1
1
0
1
0
-10
42
```

test-printbig.cs

```
fn int main()
{
  printbig(72); /* H */
  printbig(69); /* E */
  printbig(76); /* L */
  printbig(76); /* L */
  printbig(79); /* O */
  printbig(32); /*   */
  printbig(87); /* W */
  printbig(79); /* O */
  printbig(82); /* R */
  printbig(76); /* L */
  printbig(68); /* D */
  return 0;
}
```

test-printbig.out

```
  XXXXXXXXXXXXXX
  XXXXXXXXXXXXXX
        XX
        XX
        XX
  XXXXXXXXXXXXXX
  XXXXXXXXXXXXXX


  XXXXXXXXXXXXXX
  XXXXXXXXXXXXXX
  XX    XX    XX
  XX    XX    XX
  XX    XX    XX
  XX          XX


  XXXXXXXXXXXXXX
  XXXXXXXXXXXXXX
  XX
  XX
  XX
  XX


  XXXXXXXXXXXXXX
```

```
XXXXXXXXXXXXXX
XX
XX
XX
XX

  XXXXXXXXXX
XXXXXXXXXXXXXX
XX          XX
XX          XX
XX          XX
XXXXXXXXXXXXXX
  XXXXXXXXXX









    XXXXXXXXXX
XXXXXXXXXXXXXX
  XXXXXX
    XXXXXX
  XXXXXX
XXXXXXXXXXXXXX
    XXXXXXXXXX

  XXXXXXXXXX
XXXXXXXXXXXXXX
XX          XX
XX          XX
XX          XX
XXXXXXXXXXXXXX
  XXXXXXXXXX

XXXXXXXXXXXXXX
XXXXXXXXXXXXXX
    XX      XX
  XXXX      XX
XXXXXXXX    XX
XXXX  XXXXXXXX
XX    XXXXXX


XXXXXXXXXXXXXX
XXXXXXXXXXXXXX
XX
XX
XX
XX

XXXXXXXXXXXXXX
XXXXXXXXXXXXXX
XX          XX
XX          XX
XXXX      XXXX
  XXXXXXXXXX
    XXXXXX
```

test-var1.cs

```
fn int main()
{
  int i;
  i = 20;
  print(i);
  return 0;
}
```

test-var1.out

```
20
```

test-var2.cs

```
int i;

fn void foo(int x)
{
  i = x * 20;
}

fn int main()
{
  foo(1);
  print(i);
  return 0;
}
```

test-var2.out

```
20
```