FYP Interim Report 2025-2026
Department of Computer Science

Student Name	David P. Callanan	**Student Number**	21444104
Supervisor	Dr. Phil Maguire	**ECTS Credits**	5
Project Title	Studying link-time optimizations in programming language development to facilitate the continuum of static and dynamic modules.		

Contents

1 Project Objectives 2

2 Description of Work Completed 2
  2.1 Evidence of Work Completed . . . 3
  2.2 Literature Review . . . 3
  2.3 Use of GenAI and Tools . . . 3

3 Future Work 4

A Appendix: Screenshots 6

B Appendix: Source Code 9

C Appendix: LLM Prompts 67

1 Project Objectives

The goal of this project is to develop a proof-of-concept programming language ("Essence C") that facilitates bespoke optimization techniques via novel language constructs. The specific objective is to research the continuum of static and dynamic modules in low-level software development. A module system will be designed to promote dispatch flexibility by abstracting away the underlying dispatch mechanism, enabling aggressive link-time optimizations when circumstances allow. The development of a prototype compiler will demonstrate the efficacy of the proposed solution.

Success will be evidenced by the following deliverables:

(1) A functional compiler that implements the designed module system;

(2) Documentation of the module system and other core language features;

(3) A simple IDE extension that provides syntax highlighting to the programmer;

(4) A body of research into the continuum of static and dynamic modules in software development, with kernel development as a case study;

(5) An implementation of link-time optimization techniques that leverage the designed module system;

(6) An evaluation of the effectiveness of these optimizations through the benchmarking of sample programs;

(7) A monorepo containing extensive git history, demonstrating ongoing development progress;

(8) A final written report detailing the research, design, implementation and evaluation of the project.

Some considerations will be out of scope for this project, such as the performance of compilation, the level of detail of compilation errors, and the implementation of standard library functions and advanced type system features.

2 Description of Work Completed

Preliminary work on syntax design and language parsing code has been completed, and background research into the LLVM toolchain has been conducted. AI assistance was also briefly assessed.

2.1 Evidence of Work Completed

Initially, a handful of pseudocode files were created to brainstorm syntax ideas for the language. Then, a simple VS Code extension was developed to provide syntax highlighting for these early syntax ideas to reduce eye strain. Next, work commenced on the compiler frontend, written in JavaScript. Some previously written code from personal projects was used (including a simple parsing library), justifying the decision to use JavaScript. Basic parse-tree generation was implemented for a portion of the preliminary syntax, and a custom "source scope" system was developed to handle the translation between filesystem resources and source references within the programming language. Then, the build system for the compiler backend (written in C++) was organized. A Makefile was used within a Dockerized environment, itself orchestrated by a Linux/WSL environment for best reproducibility. C++ was justified due to its first-class integration with the LLVM toolchain [1].

2.2 Literature Review

LLVM was found to be a practical compilation target for low-level programming languages due to (1) its target-independent code generation [2] and (2) its integration capabilities with existing programming languages thanks to ABI interoperability (including some recent improvements [3]). Furthermore, the extensive optimization passes shared amongst languages targetting LLVM [4] should make *Essence C* directly comparable with existing languages, allowing benchmarks to be solely influenced by the additional optimization techniques employed. For instance, C can be compiled with LLVM via the popular Clang compiler [5]. It was also noticed that LLVM has experimental support for inline assembly [6], preventing the layer of indirection of a separate compilation unit (useful in kernel development [7]). Additionally, LLVM benefits greatly from intermodular optimizations at link-time [8], because its intermediate representation (IR) preserves the necessary higher-level constructs. Critically, the LLVM API exposes a powerful interface to programatically manipulate existing LLVM IR files [9] [10]. It follows that inlined compatibility wrappers can be generated to minimize optimization barriers between compilation units originating from languages other than *Essence C*.

2.3 Use of GenAI and Tools

Two notable GenAI tools have been used to assist with the research and development of this project so far.

(1) The first is t3.chat, a web application that consolidates numerous large language

models into a streamlined chat interface [11]. The user can experiment with different models to understand which models are best suited for the task at hand. Both research and programming tasks have benefited from the use of this tool. The tool was primarily used to steer the work in the right direction, and to identify the concepts, libraries and resources through which further research could be conducted. For transparency, all LLM prompts have been included in **Appendix C**.

(2) The second is the tab-complete feature of GitHub Copilot. This tool has encouraged rapid development and iteration through its behaving as a glorified autocomplete. From experience, the tool's context is severely limited, forcing the developer to have genuine knowledge of the codebase and its architecture. This is beneficial for projects that require significant unique input from the developer, in contrast to rehash projects that benefit more directly from LLM assistance.

Some experimentation with GitHub Copilot agent mode is ongoing, with notable issues identified. Firstly, the development speed is impeded by slow model response times, lack of funds, and internet connectivity issues. Secondly, there is a great difficulty in referencing contributions made by this tool due to (1) tool calls that modify the codebase directly and (2) significant implicit pollution of the context window as the agent navigates the respository. As such, usage of this tool is temporarily suspended pending further assessment.

3 Future Work

The core of the project revolves around the design of the module system. The accompanying research will be a significant undertaking, including static vs dynamic dispatch, module instance tracking, constraint propagation, and the feasibility of IR interoperability persuant to the proposed optimization goals. This research will be tackled first, in parallel with the development of any necessary boilerplate code for the compiler backend. Then, some experimentation will be needed with LLVM IR generation before continuing with the compiler backend. Everything technical other than the compiler backend is expected to be completed by the end of January, leaving adequate time for the implementation and evaluation of the compiler proper.

References

[1] The LLVM Project. “Frequently asked questions (FAQ),” Accessed: Nov. 17, 2025. [Online]. Available: https://llvm.org/docs/FAQ.html#in-what-language-is-llvm-written

[2] The LLVM Project. “The LLVM target-independent code generator,” Accessed: Nov. 17, 2025. [Online]. Available: https://llvm.org/docs/CodeGenerator.html

[3] N. Sreekumar. “GSoC 2025: Introducing an ABI lowering library,” Accessed: Nov. 7, 2025. [Online]. Available: https://blog.llvm.org/posts/2025-08-25-abi-library/

[4] The LLVM Project. “LLVM’s analysis and transform passes,” Accessed: Nov. 17, 2025. [Online]. Available: https://llvm.org/docs/Passes.html#domtree-dominator-tree-construction

[5] The Clang Project. “Clang: C language family frontend for LLVM,” Accessed: Nov. 17, 2025. [Online]. Available: https://clang.llvm.org/

[6] The LLVM Project. “LLVM language reference manual,” Accessed: Nov. 17, 2025. [Online]. Available: https://llvm.org/docs/LangRef.html#inline-assembler-expressions

[7] D. Roy. “GCC inline assembly and its usage in the linux kernel,” Accessed: Nov. 6, 2025. [Online]. Available: https://dl.acm.org/doi/fullHtml/10.5555/3024956.3024958

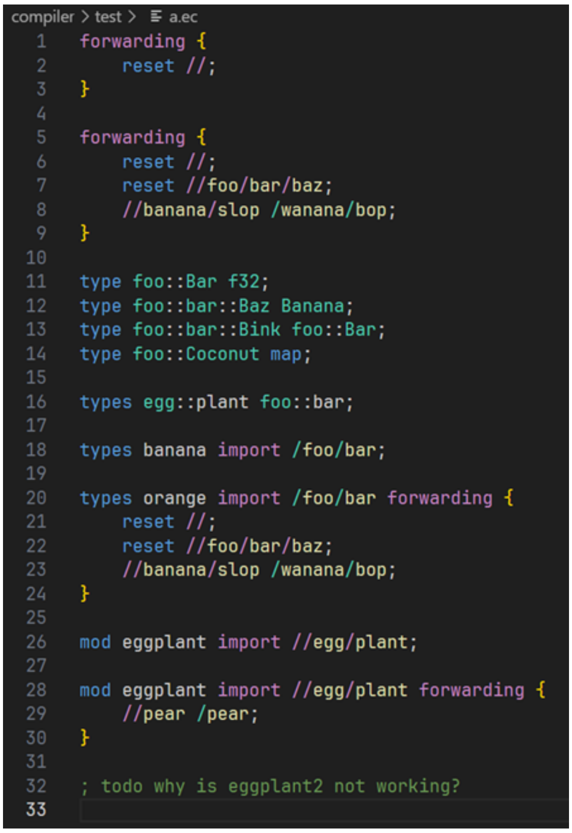
[8] The LLVM Project. “LLVM link time optimization: Design and implementation,” Accessed: Nov. 17, 2025. [Online]. Available: https://llvm.org/docs/LinkTimeOptimization.html

[9] The LLVM Project. “LLVM programmer’s manual,” Accessed: Nov. 17, 2025. [Online]. Available: https://llvm.org/docs/ProgrammersManual.html

[10] The LLVM Project. “Writing an LLVM pass,” Accessed: Nov. 2, 2025. [Online]. Available: https://llvm.org/docs/WritingAnLLVMNewPMPass.html

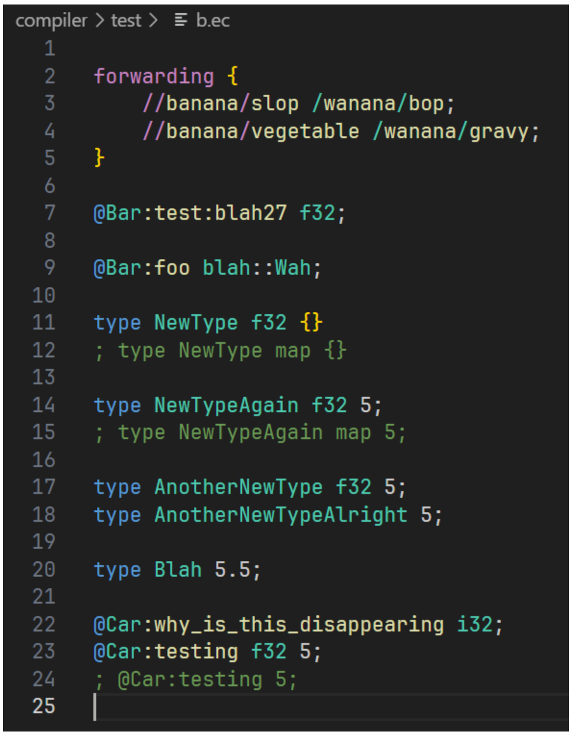
[11] DigiVirus. “Meet T3 chat AI: Your all-in-one, super-fast AI assistant,” Accessed: Nov. 16, 2025. [Online]. Available: https://digivirus.in/meet-t3-chat-ai-your-all-in-one-super-fast-ai-assistant/

A Appendix: Screenshots



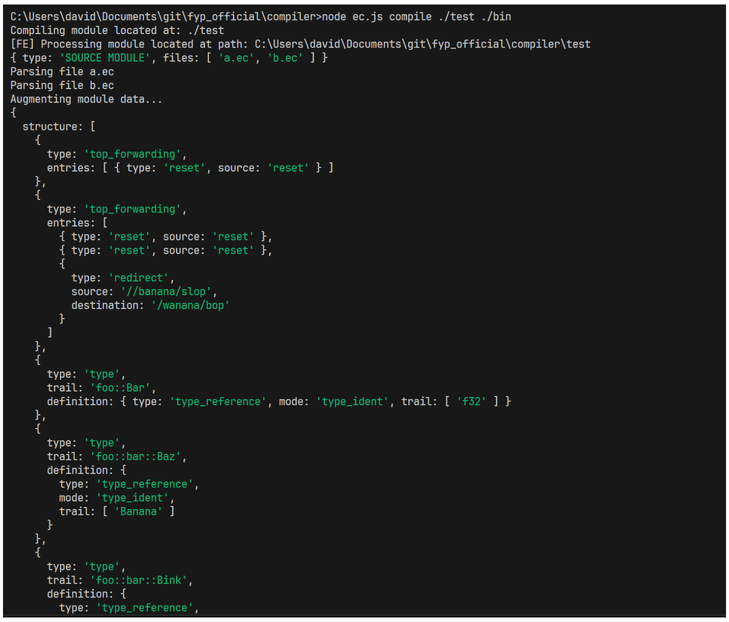
```
forwarding {
    reset //;
}

forwarding {
    reset //;
    reset //foo/bar/baz;
    //banana/slop /wanana/bop;
}

type foo::Bar f32;
type foo::bar::Baz Banana;
type foo::bar::Bink foo::Bar;
type foo::Coconut map;

types egg::plant foo::bar;

types banana import /foo/bar;

types orange import /foo/bar forwarding {
    reset //;
    reset //foo/bar/baz;
    //banana/slop /wanana/bop;
}

mod eggplant import //egg/plant;

mod eggplant import //egg/plant forwarding {
    //pear /pear;
}

; todo why is eggplant2 not working?

```

Figure 1: Syntax highlighting example A



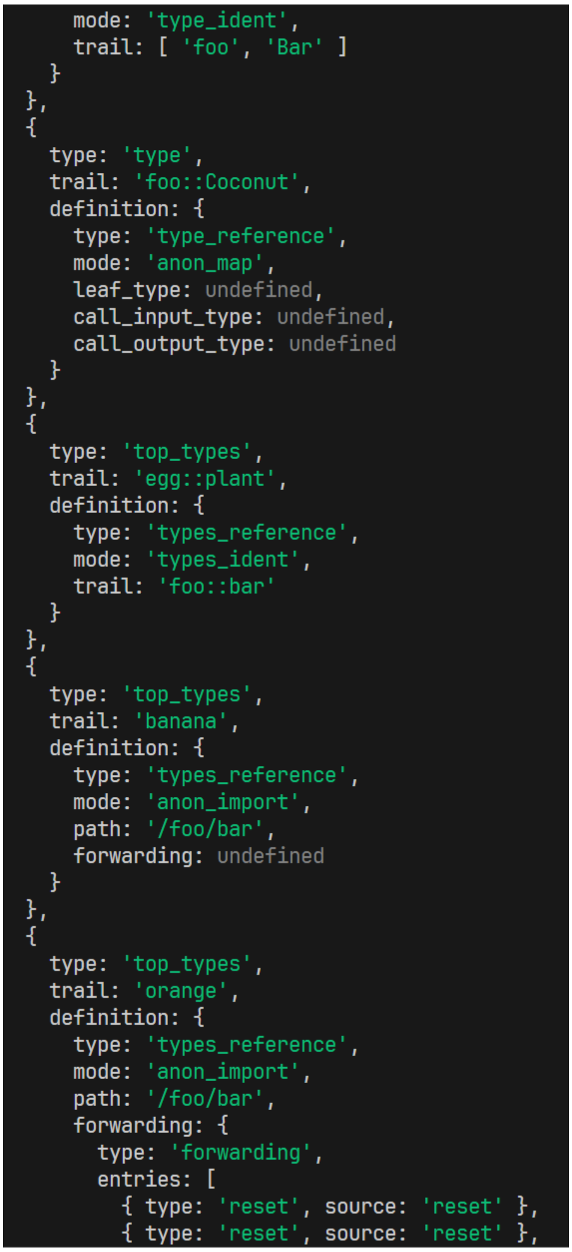
```

forwarding {
    //banana/slop /wanana/bop;
    //banana/vegetable /wanana/gravy;
}

@Bar:test:blah27 f32;

@Bar:foo blah::Wah;

type NewType f32 {}
; type NewType map {}

type NewTypeAgain f32 5;
; type NewTypeAgain map 5;

type AnotherNewType f32 5;
type AnotherNewTypeAlright 5;

type Blah 5.5;

@Car:why_is_this_disappearing i32;
@Car:testing f32 5;
; @Car:testing 5;

```

Figure 2: Syntax highlighting example B



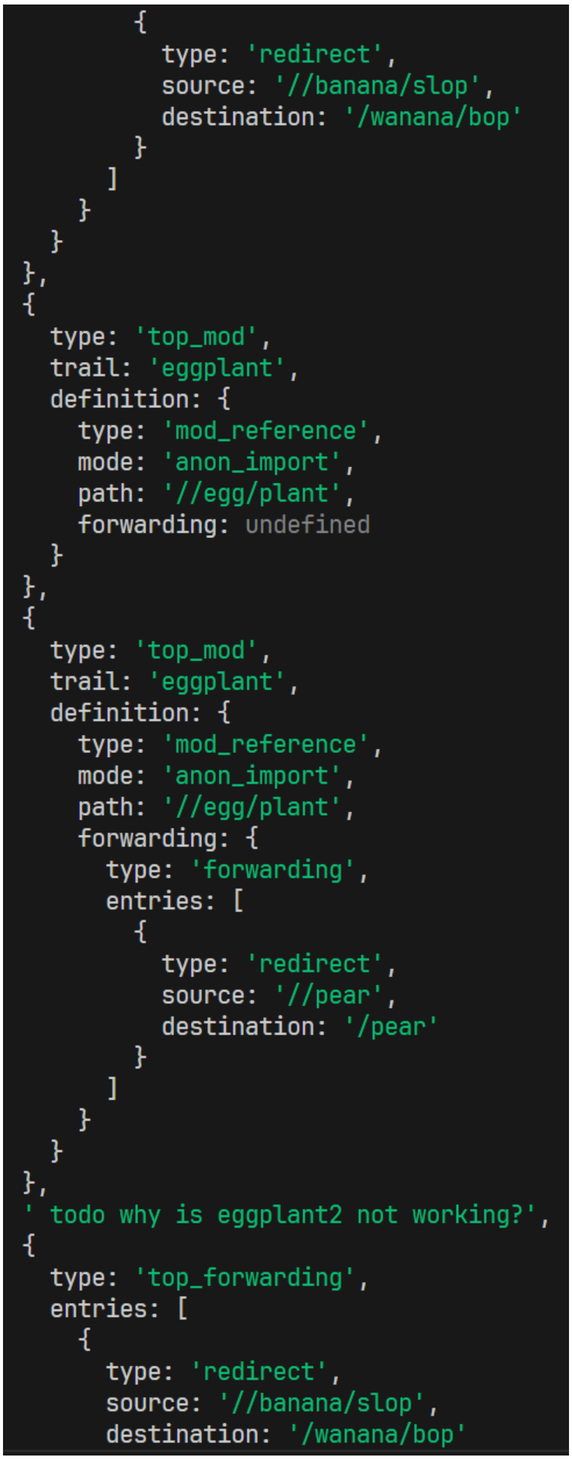
```
C:\Users\david\Documents\git\fyp_official\compiler>node ec.js compile ./test ./bin
Compiling module located at: ./test
[FE] Processing module located at path: C:\Users\david\Documents\git\fyp_official\compiler\test
{ type: 'SOURCE MODULE', files: [ 'a.ec', 'b.ec' ] }
Parsing file a.ec
Parsing file b.ec
Augmenting module data...
{
  structure: [
    {
      type: 'top_forwarding',
      entries: [ { type: 'reset', source: 'reset' } ]
    },
    {
      type: 'top_forwarding',
      entries: [
        { type: 'reset', source: 'reset' },
        { type: 'reset', source: 'reset' },
        {
          type: 'redirect',
          source: '//banana/slop',
          destination: '/wanana/bop'
        }
      ]
    },
    {
      type: 'type',
      trail: 'foo::Bar',
      definition: { type: 'type_reference', mode: 'type_ident', trail: [ 'f32' ] }
    },
    {
      type: 'type',
      trail: 'foo::bar::Baz',
      definition: {
        type: 'type_reference',
        mode: 'type_ident',
        trail: [ 'Banana' ]
      }
    },
    {
      type: 'type',
      trail: 'foo::bar::Bink',
      definition: {
        type: 'type_reference',
```

Figure 3: Syntax tree example A



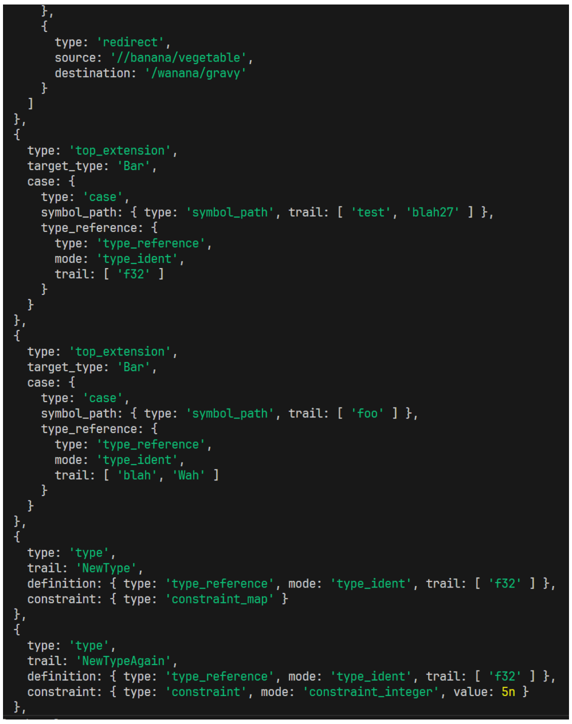
```
    mode: 'type_ident',
    trail: [ 'foo', 'Bar' ]
  }
},
{
  type: 'type',
  trail: 'foo::Coconut',
  definition: {
    type: 'type_reference',
    mode: 'anon_map',
    leaf_type: undefined,
    call_input_type: undefined,
    call_output_type: undefined
  }
},
{
  type: 'top_types',
  trail: 'egg::plant',
  definition: {
    type: 'types_reference',
    mode: 'types_ident',
    trail: 'foo::bar'
  }
},
{
  type: 'top_types',
  trail: 'banana',
  definition: {
    type: 'types_reference',
    mode: 'anon_import',
    path: '/foo/bar',
    forwarding: undefined
  }
},
{
  type: 'top_types',
  trail: 'orange',
  definition: {
    type: 'types_reference',
    mode: 'anon_import',
    path: '/foo/bar',
    forwarding: {
      type: 'forwarding',
      entries: [
        { type: 'reset', source: 'reset' },
        { type: 'reset', source: 'reset' },
```

Figure 4: Syntax tree example B



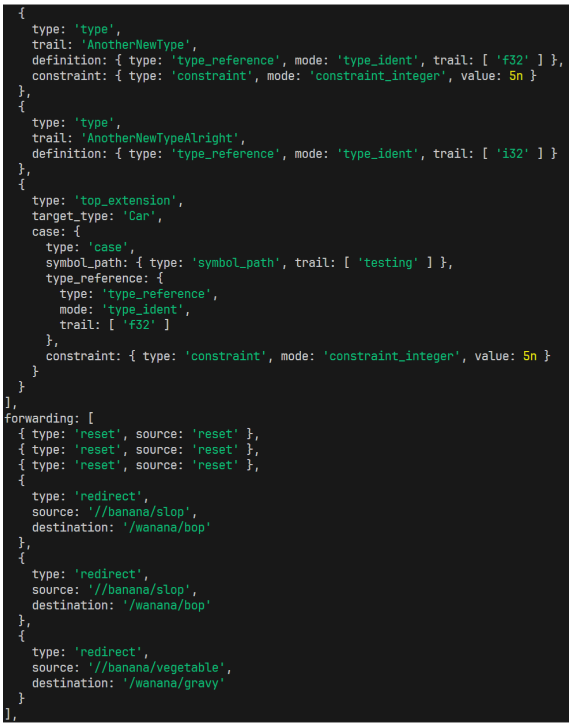
```
          {
            type: 'redirect',
            source: '//banana/slop',
            destination: '/wanana/bop'
          }
        ]
      }
  }
},
{
  type: 'top_mod',
  trail: 'eggplant',
  definition: {
    type: 'mod_reference',
    mode: 'anon_import',
    path: '//egg/plant',
    forwarding: undefined
  }
},
{
  type: 'top_mod',
  trail: 'eggplant',
  definition: {
    type: 'mod_reference',
    mode: 'anon_import',
    path: '//egg/plant',
    forwarding: {
      type: 'forwarding',
      entries: [
        {
          type: 'redirect',
          source: '//pear',
          destination: '/pear'
        }
      ]
    }
  }
},
' todo why is eggplant2 not working?',
{
  type: 'top_forwarding',
  entries: [
    {
      type: 'redirect',
      source: '//banana/slop',
      destination: '/wanana/bop'
```

Figure 5: Syntax tree example C



```
    },
    {
      type: 'redirect',
      source: '//banana/vegetable',
      destination: '/wanana/gravy'
    }
  ]
},
{
  type: 'top_extension',
  target_type: 'Bar',
  case: {
    type: 'case',
    symbol_path: { type: 'symbol_path', trail: [ 'test', 'blah27' ] },
    type_reference: {
      type: 'type_reference',
      mode: 'type_ident',
      trail: [ 'f32' ]
    }
  }
},
{
  type: 'top_extension',
  target_type: 'Bar',
  case: {
    type: 'case',
    symbol_path: { type: 'symbol_path', trail: [ 'foo' ] },
    type_reference: {
      type: 'type_reference',
      mode: 'type_ident',
      trail: [ 'blah', 'Wah' ]
    }
  }
},
{
  type: 'type',
  trail: 'NewType',
  definition: { type: 'type_reference', mode: 'type_ident', trail: [ 'f32' ] },
  constraint: { type: 'constraint_map' }
},
{
  type: 'type',
  trail: 'NewTypeAgain',
  definition: { type: 'type_reference', mode: 'type_ident', trail: [ 'f32' ] },
  constraint: { type: 'constraint', mode: 'constraint_integer', value: 5n }
},
```

Figure 6: Syntax tree example D



```
  {
    type: 'type',
    trail: 'AnotherNewType',
    definition: { type: 'type_reference', mode: 'type_ident', trail: [ 'f32' ] },
    constraint: { type: 'constraint', mode: 'constraint_integer', value: 5n }
  },
  {
    type: 'type',
    trail: 'AnotherNewTypeAlright',
    definition: { type: 'type_reference', mode: 'type_ident', trail: [ 'i32' ] }
  },
  {
    type: 'top_extension',
    target_type: 'Car',
    case: {
      type: 'case',
      symbol_path: { type: 'symbol_path', trail: [ 'testing' ] },
      type_reference: {
        type: 'type_reference',
        mode: 'type_ident',
        trail: [ 'f32' ]
      },
      constraint: { type: 'constraint', mode: 'constraint_integer', value: 5n }
    }
  }
],
forwarding: [
  { type: 'reset', source: 'reset' },
  { type: 'reset', source: 'reset' },
  { type: 'reset', source: 'reset' },
  {
    type: 'redirect',
    source: '//banana/slop',
    destination: '/wanana/bop'
  },
  {
    type: 'redirect',
    source: '//banana/slop',
    destination: '/wanana/bop'
  },
  {
    type: 'redirect',
    source: '//banana/vegetable',
    destination: '/wanana/gravy'
  }
],
```

Figure 7: Syntax tree example E

B Appendix: Source Code

Included in this appendix is a snapshot of the project repository as of the date of submission of this interim report. The full repository, including the entire git history, is available upon request. Please request access to the private repository located at https://github.com/davidcallanan/mu-fyp.

Build artifacts and other uninteresting files have been omitted from this appendix. This includes, but is not limited to, all files mentioned inside .gitignore.

.gitmodules

```
[submodule "compiler/frontend/uoe"]
    path = compiler/frontend/uoe
    url = https://github.com/davidcallanan/uoe
```

2510_syntax_ideas/idea01.ec

```
what i want to do is write my own language, but allow to user to link
   ↪ in with llvm ir should any functionality not be feasible in my
   ↪ language (e.g. specific optimization flags etc).

sources :intf {
    /x/y/z /blah/wah/frah
}

sources :impl {
    /x/y/z /blah/wah/frah
}

deps {
    :print (/print, :static(/x86_64/print, :object_wide:global)); //
   ↪ uses one inter-object implementation
    :print (/print, :static(/x86_64/print, :object_wide:named(/
   ↪ my_lovely_instance)));
    :print (/print, :static(/x86_64/print, :local)); // any instance of
   ↪  this module has its own implementation
    :print (/print :)
}

no because i think the implementation of being static etc actually
   ↪ depends on the bigger picture

deps {
    :print (/print, :concrete(/x86_64/print, :local, :static));
    :print (/print, :concrete(/x86_64/print, :named(/my_implementation)
   ↪ , :static));
    :print (/print, :passed:named(/my_implementation));
}

# this is tough

maybe a distinction for each "module" of what's "internal" to that
   ↪ module and what is "external". anything internal it is the
   ↪ responsibility of the module author to instantiate one with
   ↪ internal aspects prepopulated.

in another module, loading up a module it is assumed that anything
   ↪ internal is pre-populated.

when something is internal it is understood that you cannot create "
   ↪ multiple instances" (not of the module but of the module factory
   ↪ i mean). thus, that module can be configured at a global level.

instantiate modsource ();

:x86_64:print

do i really need a / variant?

print := import_intf:print;

the idea is that / is subject to translation.

any functions that work on the same "realm" will automatically use the
   ↪ designated translator when taking input. [for now, this
   ↪ translator is not going to exist, but the point is it could
   ↪ eventually].
```

2510_syntax_ideas/idea02.ec

```
; this is a comment because why not! i think it would be pretty cool.
; double slash means go to parent root
; triple slash means go to parent root and find its parent root
; there is a cap on triple slash. if you need to go further, you must
   ↪ then start doing mappings within the parent root "sources" entry.
; this ensures we dont go absolutely crazy.

reset-double-root; this resets // to current module
reset-triple-root; this resets /// to current module
```

```
sources {
    /// / ; maybe this is a better way to reset?
    // /
    /// reset; or reset like this
    /deps/foo //deps/foo
    /deps/bar ///deps/bar
    print("hello");
    ; only sources can access // and ///, using this anywhere else
   ↪ would be illegal, so you have to create a local mapping.
    // reset; must come at the end because previous mappings will refer
   ↪  to old version... or not, the user can just have the common
   ↪ sense to know that this only impacts child modules.
    /// resetchildren; this resets only for children.
    ; or preferably
    double-reset;
    triple-reset;
    triple-reset-children;
    double-reset-children;
    ; or maybe
    reset-double;
    reset-triple;
    reset-triple-children;
    reset-double-children;
}
```

2510_syntax_ideas/idea03.ec

```
sources {
    /intf/foo //intf/foo;
    /intf/bar //intf/bar;
    ; if reset /// appears here, it wouldn't be accessible in
   ↪ forwarding blocks
}

forwarded sources {
    reset ///;
    ; if reset /// appears here, it wouldn't be forwarded but would
   ↪ still be accessible in forwarding blocks
}

public namespace foo = import /intf/foo;

namespace bar = import /intf/bar {
    reset //;
    //intf/foo /intf/foo
    //banana/apple //banana/apple
    //banana/grape ///banana/grape
}

type Foo = foo::Foo;

public type Bar = bar::Bar;

; should anything in triple be automatically available in double?
; maybe the other way around? if we access triple, it will fallback to
   ↪ double?
; no it doesn't do justice.
```

2510_syntax_ideas/idea04.ec

```
sources {
    /intf/foo //intf/foo;
    /intf/bar //intf/bar;
    /intf/baz //intf/com.dcallanan/baz;
    ; if reset // appears here, it wouldn't be accessible in forwarding
   ↪  blocks
}

forwarded sources {
    reset //;
    ; if reset // appears here, it wouldn't be forwarded but would
   ↪ still be accessible in forwarding blocks
}

public namespace foo import /intf/foo;

namespace foo_bar import /intf/bar {
    reset //;
    //intf/foo /intf/foo;
    //banana/apple //banana/apple;
}

type Foo foo_bar::FooBarf;

public type Bar foo_bar::Bar;

type Foo map {
    :vfs map (Foo);
    :blah map (*Bar);
    :wah Foo; syntactic sugar for :wah map (Foo)
}
```

```
type Bar enum {
    :my_entry_name;
    :my_other_entry_name (

    );
}

type DeliciousInteger i33;
```

2510_syntax_ideas/idea05.ec

```
sources {
    /intf/foo //intf/foo;
    /intf/bar //intf/bar;
    /intf/baz //intf/com.dcallanan/baz;
    ; if reset // appears here, it wouldn't be accessible in forwarding blocks
}

forwarded sources {
    reset //;
    ; if reset // appears here, it wouldn't be forwarded but would still be accessible in forwarding blocks
}

public namespace foo import /intf/foo;

namespace foo_bar import /intf/bar {
    reset //;
    //intf/foo /intf/foo;
    //banana/apple //banana/apple;
}

type Foo foo_bar::FooBarf;

public type Bar foo_bar::Bar;

type Bar enum {
    :my_entry_name;
    :my_other_entry_name map (
        :vfs Foo;
        :bar i32;
    );
}

type Foo map {
    :vfs Foo; syntactic sugar for :vfs map (: Foo);
    :blah *Bar;
    :point map (i32, i32, i32);
    :distance(:x i32, :y i32) i32;
    (:x i32, :y i32) f64;
    : Foo;
}

type DeliciousInteger i33;
```

2510_syntax_ideas/idea06.ec

```
sources {
    /intf/foo //intf/foo;
    /intf/bar //intf/bar;
    /intf/baz //intf/com.dcallanan/baz {
        ; allows controlling with higher priority what is populated
   ↪  when this is later used
        ; for example if this is later passed down to other modules,
        ; it will base off the "//" at this scope,
        ; not what "//" later happens to be.
        /blah/wah //blah/wah;
        ; / always means the "module root" even though we could be
   ↪  touching a directory that does not reset the module root, recall
   ↪  the difference
        ; between the raw filesystem and the module hierarchy.
        ; thus / is the current module root irrespective of the
   ↪  directory or file mentioned.
        ; and // is the current "outer root" that this module is
   ↪  operating under.
    }
    ; if reset // appears here, it wouldn't be accessible in forwarding
   ↪   blocks
}

forwarded sources {
    reset //;
    ; if reset // appears here, it wouldn't be forwarded but would
   ↪  still be accessible in forwarding blocks
}

public namespace foo import /intf/foo;

namespace foo_bar import /intf/bar {
    reset //;
    //intf/foo /intf/foo;
    //banana/apple //banana/apple;
}

type Foo foo_bar::FooBarf;

public type Bar foo_bar::Bar;

type Bar enum {
    :my_entry_name;
    :my_other_entry_name map (
        :vfs Foo;
        :bar i32;
    );
}

type Foo map {
    :vfs Foo; syntactic sugar for :vfs map (: Foo);
    :blah *Bar;
    :point map (i32, i32, i32);
    :distance(:x i32, :y i32) i32;
    (:x i32, :y i32) f64;
    : Foo;
}

type DeliciousInteger i33;
```

2510_syntax_ideas/idea07.ec

```
sources {
    /intf/foo //intf/foo;
    /intf/bar //intf/bar;
    /intf/baz //intf/com.dcallanan/baz {
        ; allows controlling with higher priority what is populated when this is later used
        ; for example if this is later passed down to other modules,
        ; it will base off the "//" at this scope,
        ; not what "//" later happens to be.
        /blah/wah //blah/wah;
        ; / always means the "module root" even though we could be touching a directory that does not reset the module root, recall the difference
        ; between the raw filesystem and the module hierarchy.
        ; thus / is the current module root irrespective of the directory or file mentioned.
        ; and // is the current "outer root" that this module is operating under.
    }
    ; if reset // appears here, it wouldn't be accessible in forwarding blocks
}

forwarded sources {
    reset //;
    ; if reset // appears here, it wouldn't be forwarded but would still be accessible in forwarding blocks
}

public namespace foo import /intf/foo;

namespace foo_bar import /intf/bar {
    reset //;
    //intf/foo /intf/foo;
    //banana/apple //banana/apple;
}

type Foo foo_bar::FooBarf;

public type Bar foo_bar::Bar;

type Bar enum {
    :my_entry_name;
    :my_other_entry_name map (
        :vfs Foo;
        :bar i32;
    );
}

type Foo map {
    :vfs Foo; syntactic sugar for :vfs map (: Foo);
    :blah *Bar;
    :point map (i32, i32, i32);
    :distance(:x i32, :y i32) i32;
    (:x i32, :y i32) f64;
    : Foo;
}

type DeliciousInteger i33;
```

2510_syntax_ideas/idea08.ec

```
sources {
    /intf/foo //intf/foo;
    /intf/bar //intf/bar;
    /intf/baz //intf/com.dcallanan/baz {
        ; allows controlling with higher priority what is populated when this is later used
        ; for example if this is later passed down to other modules,
        ; it will base off the "//" at this scope,
        ; not what "//" later happens to be.
        /blah/wah //blah/wah;
        ; / always means the "module root" even though we could be touching a directory that does not reset the module root, recall the difference
        ; between the raw filesystem and the module hierarchy.
        ; thus / is the current module root irrespective of the directory or file mentioned.
        ; and // is the current "outer root" that this module is operating under.
    }
    ; if reset // appears here, it wouldn't be accessible in forwarding blocks
}

forwarded sources {
    reset //;
    ; if reset // appears here, it wouldn't be forwarded but would still be accessible in forwarding blocks
}

public namespace foo import /intf/foo;

namespace foo_bar import /intf/bar {
    reset //;
    //intf/foo /intf/foo;
    //banana/apple //banana/apple;
}

type Foo foo_bar::FooBarf;

public type Bar foo_bar::Bar;

type Bar enum {
    :my_entry_name;
    :my_other_entry_name map (
        :vfs Foo;
        :bar i32;
    );
}
```

```
type Foo map {
    :vfs Foo; syntactic sugar for :vfs map (: Foo);
    :blah *Bar;
    :point map (i32, i32, i32);
    :distance map map (:x i32, :y i32) i32;
    :distance map (:x i32, :y i32) -> i32; i think this looks better
    map map (:x i32, :y i32) f64;
    map (:x i32, :y i32) -> f64; i think this is good stuff
    : Foo;
}

type DeliciousInteger i33;
```

2511_syntax_ideas/idea01.ec

```
sources {
    ; probably will scrap a regular sources block because we already
   ↪ have a layer of naming indirection when "importing" should that
   ↪ not be sufficient?
    ; each import then involves a concious choice of using either "/" (
   ↪ i own the module!) or "//" (i am using something external). it
   ↪ doesn't require much effort to change between them if you change
   ↪ your mind.
}

forwarding {
    reset //;
    ; if reset // appears here, it wouldn't be forwarded but would
   ↪ still be accessible in forwarding blocks for individual imports
   ↪ to override if so wished.
    ; the default strategy is not to reset, so that forwarding is
   ↪ automatic.
    ; LHS can only be // here, but RHS can be / or //.
    ; resets must be the very first thing if wanted, otherwise no reset
   ↪ .
    ; every module implicitely resets /.
}

; the key is that a module always owns its types when it comes to the
   ↪ type system (irrespective of whether it owns the codebase when it
   ↪  comes to importing).
; thus it seems rather convenient to be able to export a namespace.
public types foo import /intf/foo;

types foo import /intf/foo;
types foo_bar import /intf/bar forwarding {
    reset //;
    //intf/foo /intf/foo;
    //banana/apple //banana/apple;
    ; like other forwarding blocks, LHS must be //, but RHS can be / or
    ↪  //.
}

type Foo foo_bar::FooBarf;

public type Bar foo_bar::Bar;

type Bar enum {
    :my_entry_name;
    :my_other_entry_name map (
        :vfs Foo;
        :bar i32;
    );
}

type Foo map {
    :vfs Foo; syntactic sugar for :vfs map (: Foo);
    :blah *Bar;
    :point map (i32, i32, i32);
    :distance map map (:x i32, :y i32) i32; i don't want to use this
    ↪ syntax
    :distance map (:x i32, :y i32) -> i32; i think this looks better
    ; the idea is that "->" automatically means a map that takes in non
    ↪ -sym input
    ; of course then in the following line we make the current map
    ↪ itself callable with non-sym.
    ; internally it is taught of as a sort of ":call" symbol, because i
    ↪  like making anything callable.
    map (:x i32, :y i32) -> f64; i think this is good stuff
    : Foo; this is what the type is when not treated as a map. the "
    ↪ leaf value" as i have called it for many years.
}

type DeliciousInteger i33;

; now in source modules, we need to think how we take in dependencies
    ↪ and all that and instantiate a module
; before i think i was thinking that the source module specifies things
    ↪  like how it wants to grab these dependencies (static vs dynamic)
    ↪ , but i think the module user should be able to control that.
; really dependencies is just a map?
```

```
dependencies map {
    :foo Foo; we want an instance of foo
    :bar_factory BarFactory;
}

options map {}

; however i don't want to create special language constructs for this.
   ↪ i wonder is it possible to make module instantiating a regular
   ↪ function. well, it is indeed a comptime function, or is it.
; no it's not, it can be runtime, so we can pass in dependencies.
; but at some stage we can specify that a variable is "static" or
   ↪ something like this, and the compiler has to trace that and
   ↪ remember it.
; the tracing might be difficult.

; i guess first, we should look at how do we import a source module
   ↪ compared to intf, and probably similar.

types foo import /impl/foo;

; i guess ‘foo‘ now has some types in it but then there would also be a
   ↪  way to instantiate it.
; there could be a cool thing i mention in my report "compile-time
   ↪ tracing of the static vs dynamic nature through runtime code".

; maybe we just take stuff "in".

in map {
    :dependencies map {
        :foo Foo; we want an instance of foo
        :bar_factory BarFactory;
    }

    :options map {
        :lru_size i32;
    }
};

; one can repeat "in map" blocks and they get merged.
; we might go ahead and force "in" map to be a non-tuple map to enable
   ↪ this straightforward merge behaviour.

; any instance of a module keeps track of whether it is a runtime or
   ↪ comptime instance.
; some simple rules can then be applied:
; when we create an instance of a module, and pass it a dependency, if
```

```
the dependency is clearly comptime (say at the global scope), the instance remembers this.
; if runtime then also remembers this.
; but the key is that comptime can be reduced to runtime (if any chance of runtime) but runtime can not ever go back up to comptime.
; we can enforce comp-time perhaps through a little keyword (and simply the compiler would choose to bail out if it can't prove it).
; by default when you create an instance of a module, it is made as comptime if the instance itself is comptime (and maybe instead of comptime we are thinking of the word "static").

; im trying to think it through. i guess the root "entry" module we compile ought to be comptime.
; if we access something straight from "input" then this comptime is preserved.
; but let's say we call some function or something to obtain the instance, then comptime is gone.
; wait, but what about when we genuinely instantiate a module instance .... ahhh

; i think it comes back to, do I own the instance or not? (similar to do i own the code?)

; let's suppose we instantiate a module instance...
; i guess a namespace can have attached to it an implementation, allowing instantiation.
; it's like the idea of leaf functions, but here, we are thinking to ourselves that there are no leaf functions anymore, just the one "instantiation" function. further non-leaf instantiation functions can always be created if so desired.

my_module := instantiate foo(
    :dependencies {
        ; blah
    }
    :options {
        ; blah
    }
);

; notice that this instantiation is done at the global scope, thus it can inherit the comptime of the current module instance.
; while properties in maps could also keep track of comptime vs runtime , i feel this might be too complex for a fyp that is due in March .
; so let's just simplify for now and so VARIABLES at module scope keep track of comptime vs runtime.
; variables are easier to deal with.. don't got the complexities of a
   ↪ map with types and all that.

; note though: the "input" is a map, so we do have to keep track
   ↪ through maps ahhhhh

; right how via gonna track via maps? remember we have a few
   ↪ possibilities, we have static, we have thread-local storage or
   ↪ global variable, and we have dynamic.
; however, we can assume tls and gv are static, as a static
   ↪ implementation can wrap the dynamic nature.
; we can use LTO (link time optimization) to easily do this (which
   ↪ would be great to talk about in my report).

; in my report: "technically C does not have to even compile to object
   ↪ files, it could compile each file to an executable file and then
   ↪ use an overkill runtime linkage mechanism, the point is that
   ↪ there is room for alteration in this".

; so all "sym" map instances (don't care about non-sym variants) must
   ↪ keep track of their comptime vs runtime status, collapsing to the
   ↪  worst-case scenario (kind of like how a typesystem collapses its
   ↪  possible values).

; by default, dispatch is "static", but there would be perhaps three
   ↪ options for this demo.

my_module := instantiate foo(

) dispatch {
    static force
};

my_module := instantiate foo(

) dispatch {
    global
};

my_module := instantiate foo(

) dispatch {
    dynamic
};

; force will ensure that the compiler bails out if not possible, but by
   ↪  default, it is just a recommendation.
```

```
; when one instantiates a module in another scope, then dispatch is
   ↪ ignored and it is just treated as dynamic, like instantiating a
   ↪ class.

; however, what if we instantiate two modules that both implement that
   ↪ interface and might pass through?
; i think when we say dynamic, we do in fact mean double dispatch,
   ↪ unfortunately. there is an element of saying the implementation
   ↪ is fixed, but its dependencies etc. is dynamic, so maybe we can
   ↪ consider dyanamic-single vs dynamic-double.
; even some of these things can be put through as considerations in my
   ↪ report, whether i implement or not (because implementing dynamic
   ↪ gonna be hell of a lot easier).

; i think i can make effort to propogate dyanmic-single vs dynamic-
   ↪ double,,, we'll seee..



; im thinking when importing a source module it will instaed be

mod my_module import /impl/my_module;

; we distinguish between "mod" and "ns" because "mod" has instantiation
   ↪  function.
```

2511_syntax_ideas/idea02.ec

```
; i was leaning towards not having special treatment for module
   ↪ instances and that we can have a receiver-like system for any
   ↪ sort of instances.
; but we would force that modules require an instance, in the sense
   ↪ that we have leaf maps that can't do much by themselves.

; it just could get messy if a module has the ability to create a
   ↪ vector etc.
; i might go with implicit module receivers (for anything called
   ↪ create_mod)?
; so instead of this:mod:blah we'd just have mod:blah.
; or maybe module gets special treatement i don't know at this point.

create(:my_dependency Foo, :math Math) {
    this:my_dependency @= my_dependency; // @= says "declare and assign
   ↪ if not already declared", whereas ":= is for variables and can
```

```
    have shadowing, whereas @= we know like there's genuinely only one version".
    this:math @= math;
}

@:create_world() {
    mod:my_dependency:do_something;
}

@:create_world:create_vector(x f64, y f64, z f64) {
    mod:my_dependency:do_something_or_another;

    this:x @= x;
    this:y @= y;
    this:z @= z;
}

@:create_world:create_vector:get_x() -> f64 {
    return this:x;
}

@:create_world:create_vector:get_y() -> f64 {
    return this:y;
}

@:create_world:create_vector:get_z() -> f64 {
    return this:z;
}

@:create_world:create_vector:get_distance(other T:create_world:create_vector) -> f64 {
    return mod:math:sqrt(
        + (this:x - other:get_x) * (this:x - other:get_x)
        + (this:y - other:get_y) * (this:y - other:get_y)
        + (this:z - other:get_z) * (this:z - other:get_z)
    );
}

; i can talk about how forcing no ability for global variables forces the compiler to track everything. preparing us nicely for optimizations.

; need some self-loop of wanting to actually get the exposed version of "this", like "this~" or something.

; ah what now about private functions within my shtuff! now i need a symbol for private too?
; also look now carefully how awful the typing syntax is...
; maybe i do need to have type for returned instances, but damn i didn'
   ↪ t want this.
; also note that i want want different factories for a type, but really
   ↪  the other factories should wrap a main internal factory kind of.

; also create_vector doesn't know any context about create_world unless
   ↪  the information of world is stored in vector's this... so it's
   ↪ kinda like not really appropriate to even have this create_world
   ↪ mentioned...

; note also that maybe if we do have like a Vector declaration that we
   ↪ need only include mutable things in here (to prevent the need for
   ↪  @=) and anything constant can be extended in the same way methods
   ↪  are.
```

2511_syntax_ideas/idea03.ec

```
; let's have another go at idea02.

type Mod map {
    :_my_dependency Foo;
    :_math Math;
}

create(:my_dependency Foo, :math Math) {
    ; if constructor finds something not assigned, compile-time error.
    mod:_my_dependency = my_dependency;
    mod:_math = math;
    ; i'm thinking of _ here but in practice i think i want to follow
   ↪ the public approach where there is no such thing as module-level
   ↪ private.
}

type World map {
}

; the idea is that one should be able to immediately start doing stuff
   ↪ with @Mod without even using "create" or "type Mod map" unless
   ↪ they actually need it.

@Mod:create_world() -> World { ; we want () here to allow later
   ↪ extensibility.. this is just a convention that programmers would
   ↪ likely follow.
    mod:_my_dependency:do_something;
}

type Vector map {
    :_x f64;
    :_y f64;
    :_z f64;
}

@World:create_vector(x f64, y f64, z f64) -> Vector {
    mod:_my_dependency:do_something_or_another;

    this:_x = x;
    this:_y = y;
    this:_z = z;
}

@Vector:get_x() -> f64 {
    return this:_x;
}

@Vector:get_y() -> f64 {
    return this:_y;
}

@Vector:get_z() -> f64 {
    return this:_z;
}

@Vector:get_distance(other T:Vector) -> f64 {
    return mod:_math:sqrt(
        + (this:_x - other:get_x) * (this:_x - other:get_x)
        + (this:_y - other:get_y) * (this:_y - other:get_y)
        + (this:_z - other:get_z) * (this:_z - other:get_z)
    );
}
```

2511_syntax_ideas/idea04.ec

```
; the goal of this iteration is to determine how public private should
  ↪ work
; and to resolve the notations of maps (RHS) of @ extensions.
; i'd also like to determine how to make things mutable.

; anything mutable must be defined in the type. how about we suppose
  ↪ that this is not necessary for sake of exploration.
```

```
@Mod:my_dependency Foo;
@Mod:math Math;

create(:my_dependency Foo, :math Math) {
    mod:my_dependency = my_dependency;
    mod:math = math;
}

type World map;

pub @Mod:create_world map () -> World . { ; . means implicit? this
   ↪ distinguishes between type details and actual body.
    mod:my_dependency:do_something;
}

type Vector map;

mut @Vector:x f64; mut kinda means non-deterministic
mut @Vector:y f64; these should mutable because maybe we have a method
   ↪ to mutate them
mut @Vector:z f64;

pub @World:create_vector map (x f64, y f64, z f64) -> Vector . {
    mod:my_dependency:do_something_or_another;

    ; this is problematic. this could be referring to world, or it
   ↪ could be referring to vector instance.

    this:x = x;
    this:y = y;
    this:z = z;
}

pub @Vector:get_x map . {
    return this:x;
}

pub @Vector:get_y map . {
    return this:y;
}

pub @Vector:get_z map . {
    return this:z;
}

pub @Vector:get_distance map (other Vector) -> f64 . {
    return mod:math:sqrt(
        + (this:x - other:get_x) * (this:x - other:get_x)
        + (this:y - other:get_y) * (this:y - other:get_y)
        + (this:z - other:get_z) * (this:z - other:get_z)
    );
}

; on the next iteration i need to resolve some things:
; - this does not work in factories very nicely... ahhh!!!
; - . for implicit is weird... i need to think that through more.
```

2511_syntax_ideas/idea05.ec

```
; i think we already decided prior that the whole point of prefixing maps with "map" at the type level is so that we can take advantage of () and {} notation for instances.

; so i wonder can we get going with that immediately.

@Mod:my_dependency Foo;
@Mod:math Math;

create(:my_dependency Foo, :math Math) {
    :my_dependency my_dependency;
    :math math;
}

type World map;

pub @Mod:create_world() -> World { ; here World is kinda optional, and we can make everything optional because we know the difference between type and instance at the lexer level almost.
    mod:my_dependency:do_something;
}

type Vector map;

mut @Vector:x f64;
mut @Vector:y f64;
mut @Vector:z f64;

pub @World:create_vector(:x f64, :y f64, :z f64) -> Vector {
    mod:my_dependency:do_something_or_another;

    ; normally we would return things like this:

    :x x;
    :y y;
    :z z;

    ; but remember! these are private!
    ; seems dodgy.
    ; then again, the notion of private is a module-level thing (Go ftw).
    ; there's not supposed to be the ability to do private.
    ; if you need truly private and temporary, then use a local variable
    ; otherwise you just gotta use a field.
    ; you can internally decide on a _x convention if wanted, but my thinking is that this is unnecessary
    ; since i myself have accessed _x things outside where i thought i would in javascript many times, that's the whole point of having a module level of privacy.
}

pub @Vector:get_x f64 {
    return this:x;
}

pub @Vector:get_y f64 {
    return this:y;
}

pub @Vector:get_z f64 {
    return this:z;
}

pub @Vector:get_distance(other Vector) -> f64 {
    return mod:math:sqrt(
        + (this:x - other:get_x) * (this:x - other:get_x)
        + (this:y - other:get_y) * (this:y - other:get_y)
        + (this:z - other:get_z) * (this:z - other:get_z)
    );
}

; this actually looks quite good.
```

2511_syntax_ideas/idea06.ec

```
@Mod:my_dependency Foo;
@Mod:math Math;

create(:my_dependency Foo, :math Math) {
    :my_dependency my_dependency;
    :math math;
}

type World map;

pub @Mod:create_world() -> World {
    mod:my_dependency:do_something;
}

type Vector map;

mut @Vector:x f64;
mut @Vector:y f64;
mut @Vector:z f64;

pub @World:create_vector(:x f64, :y f64, :z f64) -> Vector {
    mod:my_dependency:do_something_or_another;

    :x x;
    :y y;
    :z z;
}

pub @Vector:get_x f64 {
    return this:x;
}

pub @Vector:get_y f64 {
    return this:y;
}

pub @Vector:get_z f64 {
    return this:z;
}

pub @Vector:get_distance(other Vector) -> f64 {
    return mod:math:sqrt(
        + (this:x - other:get_x) * (this:x - other:get_x)
        + (this:y - other:get_y) * (this:y - other:get_y)
        + (this:z - other:get_z) * (this:z - other:get_z)
    );
}
```

2511_syntax_ideas/idea07.ec

```
types foo import //intf/foo;

mod bar import /impl/bar forwarding {
    reset //; todo: i need to update my code to allow resetting any "
  ↪ folder"?
    reset //foo/bar/baz;
    //intf/foo //intf/foo;
}

type Foo foo::Foo;
type Math bar::Math;

@Mod:my_dependency Foo;
@Mod:math Math;

create(:my_dependency Foo, :math Math) -> {
    :my_dependency my_dependency;
    :math math;
}

type World map;

pub @Mod:create_world() -> World {
    mod:my_dependency:do_something;
}

type Vector map;

mut @Vector:x f64;
mut @Vector:y f64;
mut @Vector:z f64;

pub @World:create_vector(:x f64, :y f64, :z f64) -> Vector {
    mod:my_dependency:do_something_or_another;

    :x x;
    :y y;
    :z z;
}

pub @Vector:get_x f64 {
    return this:x;
}

pub @Vector:get_y f64 {
    return this:y;
}
```

```
pub @Vector:get_z f64 {
    return this:z;
}

pub @Vector:get_distance(other Vector) -> f64 {
    first_part := (this:x - other:get_x) * (this:x - other:get_x)
    second_part := (this:y - other:get_y) * (this:y - other:get_y)
    third_part := (this:z - other:get_z) * (this:z - other:get_z)

    : mod:math:sqrt(
        + first_part
        + second_part
        + third_part
    );
}
```

README.md

```
# Essence C (Maynooth University - Final Year Project)

This respository acts as the definitive record of work for my final
   ↪ year project. I will try to keep as much of my work as possible
   ↪ within this repo - including code, writeups, notes, random
   ↪ thoughts.

The goal of this project is to develop a custom programming language
   ↪ known as "Essence C".

The language should include bespoke mechanism(s) to facilitate a
   ↪ flexible module dispatch mechanism - primarily achieved by fusing
   ↪  the notions of classes and modules that are traditionally
   ↪ separate in existing programming languages.

The end goal of this is to enhance the compiler's potential to perform
   ↪ link-time optimization on the resulting artifacts, resulting in
   ↪ small performance optimizations relative to other low-level
   ↪ languages (if things work out smoothly).

Additional research (if time permits) will be to study how this
   ↪ facilitates the continuum of both static and dynamic modules in
   ↪ kernel development. This may (if time permits) involve writing
   ↪ some simple kernel code in my programming language and analyzing
   ↪ performance gains relative to C code. The goal will be to get the
   ↪  project working before focusing on these aspects though.
```

```
One core aspect of the project will be attempting to maintain
    ↪ interconnectivity with existing low-level infrastructure, as the
    ↪ language must be suitable for kernel development. It is also
    ↪ pretty much guaranteed that LLVM IR will be the target (and I
    ↪ want to look at Clang's link-time optimizations when it uses LLVM
    ↪  IR). If I am using LLVM IR, the easiest approach will be to
    ↪ support interconnectivity between my language and LLVM IR, rather
    ↪  than offering direct integration with any other programming
    ↪ languages. LLVM IR also supports inline assembly really
    ↪ effectively (and this should be adequate for kernel development
    ↪ although I would have to look into this). My compiler might have
    ↪ to do some analysis of outputted LLVM IR files and even do some
    ↪ minor mutations to them to facilitate my performance
    ↪ optimizations (I think this is likely given that we may link
    ↪ foreign LLVM IR code and we may need to touch this code to get
    ↪ the best out of our optimizations across module boundaries, which
    ↪  is what I would want to consider this an absolute success).

Of course, this final year project is really just initial research and
    ↪ development of something that could take years of work. So I must
    ↪  manage my expectations and remember that I am not creating a
    ↪ fully-functional programming language that is ready for
    ↪ production use, or anything close to it.

## Project Structure

- ‘compiler‘ houses the actual compilation process which takes in
    ↪ language code and spits out a final executable.
- ‘extension‘ houses the VS Code extension that offers syntax
    ↪ highlighting to the programmer.
- ‘XXXX_syntax_ideas‘ contains some early syntax ideas ideas which have
    ↪  been significantly deviated from.

## Git Submodules

When cloning this repo, make sure git "submodules" are enabled and are
    ↪ recursively cloned.
```

appendix_generator/source_code/README.md

```
# Appendix Generator - Source Code

This quick and dirty tool takes all files in the codebase (except some
    ↪ explicitly ignored patterns) and dumps their contents into a
    ↪ latex file for easy inclusion in the interim and final report.
```

```
Proper formatting and escape logic was implemented to get this to work
    ↪ nicely.

Run ‘node app.js‘ from this directory and the result will be outputted
    ↪ to ‘appendix_source_code.tex‘.
```

appendix_generator/source_code/app.js

```
import fs from "fs/promises";
import path from "path";
import { fileURLToPath } from "url";

const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

const root_path = path.resolve(__dirname, "../../");
const output_path = path.join(__dirname, "appendix_source_code.tex");

const ignored_paths = new Set([
    "uoe",
    "pnpm-lock.yaml",
    ".gitignore",
    ".vscodeignore",
]);

const ignored_extensions = new Set([
    ".pdf",
    ".png",
    ".tex",
]);

const escape_latex = (text) => {
    return (text
        .replace(/\\/g, "\\textbackslash{}")
        .replace(/\^/g, "\\textasciicircum{}")
        .replace(/~/g, "\\textasciitilde{}")
        .replace(/[{}&%$#_]/g, "\\$&")
    );
};

const escape_verbatim = (text) => {
    // don’t ask me how many hours it took to figure this out

    return (text
        .replace(/@/g, "@\\char\"40@")
        .replace(/\\end\{lstlisting\}/g, "@\\textbackslash{}end\{
```

```
   ↪ lstlisting\}@")
        // .replace(/\\/g, "(*@\\textbackslash{}@*)")
        // .replace(/%/g, "(*@\\%@*)");
    );
};

const parse_gitignore = async (path) => {
    const patterns = [];

    try {
        var content = await fs.readFile(path, "utf8");
    } catch (err) {
        console.log("No .gitignore found at:", path);
        return [];
    }

    const lines = content.split("\n");

    for (let line of lines) {
        line = line.trim();

        if ((false
            || line === ""
            || line.startsWith("#")
        )) {
            continue;
        }

        const is_negated = line.startsWith("!");

        if (is_negated) {
            line = line.substring(1);
        }

        patterns.push({ pattern: line, is_negated });
    }

    return patterns;
};

const pattern_to_regex = (pattern) => {
    const is_root = pattern.startsWith("/");

    if (is_root) {
        pattern = pattern.substring(1);
    }
```

```
const is_directory_only = pattern.endsWith("/");

if (is_directory_only) {
    pattern = pattern.substring(0, pattern.length - 1);
}

let regex_str = "";

for (let i = 0; i < pattern.length; i++) {
    const char = pattern[i];

    if (char === "*") {
        if ((true
            && i + 1 < pattern.length
            && pattern[i + 1] === "*"
        )) {
            regex_str += ".*";
            i++;
            if ((true
                && i + 1 < pattern.length
                && pattern[i + 1] === "/"
            )) {
                i++;
            }
        } else {
            regex_str += "[^/]*";
        }
    } else if (char === "?") {
        regex_str += "[^/]";
    } else if ((false
        || ".*+?^${}()|[]\\".includes(char)
    )) {
        regex_str += "\\" + char;
    } else {
        regex_str += char;
    }
}

if (is_root) {
    regex_str = "^" + regex_str;
} else {
    regex_str = "(^|/)" + regex_str;
}

if (is_directory_only) {
    regex_str += "(/|$)";
} else {
        regex_str += "(/.*|$)";
    }

    return new RegExp(regex_str);
};

const load_gitignore_rules = async (base_dir) => {
    const rules_by_dir = new Map();

    const analyze_directory = async (dir, rel_path) => {
        rel_path ??= "";

        const gitignore_path = path.join(dir, ".gitignore");
        const patterns = await parse_gitignore(gitignore_path);

        if (patterns.length > 0) {
            const normalized_rel_path = rel_path.replace(/\\/g, "/");
            rules_by_dir.set(normalized_rel_path, patterns);
        }

        const entries = await fs.readdir(dir, { withFileTypes: true });

        for (const entry of entries) {
            if ((true
                && entry.isDirectory()
                && entry.name !== ".git"
            )) {
                const new_rel_path = rel_path ? path.join(rel_path,
   ↪ entry.name) : entry.name;
                await analyze_directory(path.join(dir, entry.name),
   ↪ new_rel_path);
            }
        }
    };

    await analyze_directory(base_dir);

    return rules_by_dir;
};

const is_ignored = (file_path, _is_directory, rules_by_dir) => {
    const normalized = file_path.replace(/\\/g, "/");
    const segments = normalized.split("/");

    let is_ignored = false;

    for (let i = 0; i <= segments.length; i++) {
        const dir_path = segments.slice(0, i).join("/");
        const remaining = segments.slice(i).join("/");

        const rules = rules_by_dir.get(dir_path === "" ? "" : dir_path)
   ↪  ?? [];

        for (const { pattern, is_negated } of rules) {
            const regex = pattern_to_regex(pattern);

            if (regex.test(remaining)) {
                is_ignored = !is_negated;
            }
        }
    }

    if ((false
        || normalized === ".git"
        || normalized.startsWith(".git/")
    )) {
        is_ignored = true;
    }

    for (const part of segments) {
        if (ignored_paths.has(part)) {
            is_ignored = true;
            break;
        }
    }

    for (const ext of ignored_extensions) {
        if (normalized.endsWith(ext)) {
            is_ignored = true;
            break;
        }
    }

    return is_ignored;
};

const collect_files = async (dir, base_path, rules_by_dir) => {
    const results = [];
    const entries = await fs.readdir(dir, { withFileTypes: true });

    for (const entry of entries) {
        const full_path = path.join(dir, entry.name);
        const rel_path = base_path ? path.join(base_path, entry.name) :
   ↪  entry.name;
```

```
        if (is_ignored(rel_path, entry.isDirectory(), rules_by_dir)) {
            continue;
        }

        if (entry.isDirectory()) {
            results.push(...await collect_files(full_path, rel_path,
   ↪ rules_by_dir));
        } else if (entry.isFile()) {
            results.push(rel_path);
        }
    }

    return results;
};

const generate_latex = async () => {
    console.log("Collecting all files from:", root_path);
    console.log("Loading .gitignore rules...");

    const rules_by_dir = await load_gitignore_rules(root_path);
    console.log(‘Loaded .gitignore rules from ${rules_by_dir.size}
   ↪ directories.‘);

    const files = await collect_files(root_path, "", rules_by_dir);
    files.sort();

    console.log(‘Found ${files.length} files to include in this report’
   ↪ s appendix.‘);

    let latex = "";
    latex += "% This is an auto-generated appendix using my appendix
   ↪ generator script\n";
    latex += "% Date of generation: " + new Date().toISOString() + "\n\
   ↪ n";
    latex += "\\lstset{\n";
    latex += "  breaklines=true,\n";
    latex += "  breakatwhitespace=false,\n";
    latex += "  postbreak={\\mbox{\\textcolor{red}{$\\hookrightarrow$
   ↪ }\\space}},\n";
    latex += "  columns=fixed,\n";
    latex += "  keepspaces=true,\n";
    latex += "  showstringspaces=false,\n";
    latex += "  tabsize=4,\n";
    latex += "}\n\n";

    let i = 0;
    for (const file_path of files) {
        console.log(‘Processing: ${file_path}‘);

        const full_path = path.join(root_path, file_path);

        const content = await fs.readFile(full_path, "utf8");

        const normalized_path = file_path.replace(/\\/g, "/");
        const escaped_path = escape_latex(normalized_path);
        const escaped_content = escape_verbatim(content);

        latex += "\\vspace{0.5cm}\n";
        latex += "\\noindent\\colorbox{lightgray}{\\parbox{\\dimexpr\\
   ↪ textwidth-2\\fboxsep}{\\small\\textbf{" + escaped_path + "}}}\n\n
   ↪ ";
        latex += "\\vspace{0.2cm}\n\n";
        latex += "\\begin{lstlisting}[language={}, escapechar=@,
   ↪ basicstyle=\\ttfamily\\footnotesize, breaklines=true,
   ↪ breakatwhitespace=false]\n";

        // change to limit how many files are outputted for debugging
   ↪ purposes.
        if (i <= 999) { // 17
            latex += escaped_content;
        }

        if (!escaped_content.endsWith("\n")) {
            latex += "\n";
        }

        latex += "\\endlstlisting\n\n";

        i++;
    }

    await fs.writeFile(output_path, latex, "utf8");
    console.log(‘\nLatex appendix has been succesfully outputted to: ${
   ↪ output_path}‘);
    console.log(‘Total number of files: ${files.length}‘);
};

generate_latex();
```

appendix_generator/source_code/package.json

```
{
    "type": "module"
}
```

compiler/README.md

```
# Compiler

This directory contains the main entrypoint to the compiler.

The compilation process is separated into two phases:

- ‘frontend‘ (JavaScript): Parsing + initial analysis and
   ↪ transformations.
- ‘backend‘ (C++): Remaining analysis and transformations + codegen
   ↪ using LLVM.

While I intended to write the ‘backend‘ in JavaScript, the primary
   ↪ interface for LLVM is ‘C++‘-based, so I’m writing this phase in ‘
   ↪ C++‘ for simplicity.

## Setup

- Install ‘pnpm ^9.15.1‘.
```

compiler/backend/Dockerfile

```
FROM ubuntu:24.04

RUN apt-get update

RUN apt-get install -y \
    make \
    # LLVM toolchain
    llvm-17 \
    llvm-17-dev \
    clang-17 \
    lld-17 \
    # One can add cross-compilation dependencies here in future
    && rm -rf /var/lib/apt/lists/*

RUN ln -sf /usr/bin/clang-17 /usr/bin/clang && \
    ln -sf /usr/bin/clang++-17 /usr/bin/clang++

ENV LLVM_CONFIG=llvm-config-17
```

```
WORKDIR /app

COPY . .

RUN mkdir -p /app/out /volume/out

RUN make -j$(nproc) all

RUN chmod +x live.sh

VOLUME ["/volume/out"]
VOLUME ["/volume/in"]

# Final steps that are dynamic (using live volumes) are delegated to
   ↪ this dedicated shell script.
CMD ["bash", "./live.sh"]
```

compiler/backend/Makefile

```
CXX                 := clang++
AR                  := ar

CXXFLAGS            := -std=c++20 -O2 -Wall -Wextra -Wpedantic
LDFLAGS             :=

LLVM_COMPONENTS     := core support irreader

LLVM_CXXFLAGS       := $(shell $(LLVM_CONFIG) --cxxflags)
LLVM_LDFLAGS        := $(shell $(LLVM_CONFIG) --ldflags)
LLVM_LIBS           := $(shell $(LLVM_CONFIG) --libs $(LLVM_COMPONENTS))
SYSTEM_LIBS         := $(shell $(LLVM_CONFIG) --system-libs)

BIN                 := bin/backend

src_files := $(wildcard src/*.cpp)
obj_files := $(patsubst src/%.cpp,build/%.o,$(src_files))

build/%.o: src/%.cpp | build
    $(CXX) $(CXXFLAGS) $(LLVM_CXXFLAGS) -c $< -o $@

$(BIN): $(obj_files) | bin
    $(CXX) -o $@ $^ $(LDFLAGS) $(LLVM_LDFLAGS) $(LLVM_LIBS) $(
   ↪ SYSTEM_LIBS)

build:
	mkdir -p build

bin:
	mkdir -p bin

.PHONY: all
all: $(BIN)
```

compiler/backend/README.md

```
# Compiler Backend

Language: C++.

## Requirements

- Docker
- Linux or WSL (Windows)

## Build & Run

**Linux or WSL**:

```
chmod +x ./host/run.sh
./host/run.sh
```
```

compiler/backend/host/build.sh

```
docker build -t davidcallanan--mu-fyp--compiler-backend .
```

compiler/backend/host/buildrun.sh

```
./host/build.sh && \
./host/run.sh
```

compiler/backend/host/run.sh

```
docker run -it -v "$(pwd)/in:/volume/in" -v "$(pwd)/out:/volume/out"
   ↪ davidcallanan--mu-fyp--compiler-backend
```

compiler/backend/live.sh

```
shopt -s nullglob

/app/bin/backend

files=(/app/out/*)
if (( ${#files[@]} )); then
  cp -r "${files[@]}" /volume/out/
fi
```

compiler/backend/src/main.cpp

```
#include <fstream>
#include <iostream>

int main() {
    std::string output_path = "/app/out/test.txt";

    std::ofstream out_file(output_path);

    if (!out_file) {
        std::cerr << "error" << std::endl;
        return 1;
    }

    out_file << "Hello, world!" << std::endl;

    out_file.close();

    std::cout << "done" << std::endl;

    return 0;
}
```

compiler/ec.js

```
import { Command } from "commander";
import { process as frontend_process } from "./frontend/process.js";

const program = new Command();

const backend_process = () => { };

const compile_module = async (config) => {
    console.log("Compiling module located at:", config.src);
    const result = await frontend_process(config);
    console.dir(result, { depth: null });
    await backend_process(result);
};

program
    .name("ec")
    .description("Essence C compiler")
    .version("1.0.0");

program
    .command("compile <src> <dest>")
    .description("Compile a module located at <src> and output build
   ↪ artifacts to <dest>")
    .action(async (src, dest, _options) => {
        await compile_module({
            src,
            dest,
        });
    });

program.parse();
```

compiler/frontend/README.md

```
# Compiler Frontend

Language: JavaScript.
```

compiler/frontend/create_fs_source_scope.js

```
import { create_unsuspended_factory } from "./uoe/
   ↪ create_unsuspended_factory.js";

class FsSourceScope {
    _init(dependencies, base_path) {
        this._dependencies = dependencies;
        this._base_path = base_path;
    }

    async resolve(path) {
        // todo: add lru cache of like 1000 entries here.
        const directory = this._dependencies.path.join(this._base_path,
   ↪  path);

        try {
            var files = await this._dependencies.fs.readdir(directory);
        } catch (e) {
            console.warn(e);
            return undefined;
        }

        if (files.some(file => file.endsWith(".ec"))) {
            return {
                type: "SOURCE MODULE",
                files: files.filter(file => file.endsWith(".ec")),
            };
        } else if (files.some(file => file.endsWith(".eh"))) {
            return {
                type: "INTERFACE MODULE",
                files: files.filter(file => file.endsWith(".eh")),
            };
        }

        return undefined;
    }
}

export const create_fs_source_scope = create_unsuspended_factory(
   ↪ FsSourceScope);
```

compiler/frontend/create_source_scope.js

```
import { create_unsuspended_factory } from "./uoe/
   ↪ create_unsuspended_factory.js";

const validate_redirect = (redirect) => {
    if (false
        || !Array.isArray(redirect)
        || typeof redirect[0] !== "string"
        || (true
            && redirect[1] !== undefined
            && typeof redirect[1] !== "function"
            && typeof redirect[1] !== "object"
        )
        || (true
            && redirect[2] !== undefined
            && typeof redirect[2] !== "string"
        )
    ) {
        return false;
    }
    return true;
};

const validate_redirects = (redirects) => {
    if (false
        || !Array.isArray(redirects)
        || redirects.some(redirect => !validate_redirect(redirect))
    ) {
        throw new Error("Validation error.");
    }
};

const validate_path = (path) => {
    if (false
        || typeof path !== "string"
        || !/^[a-z0-9_\/.]*$/.test(path)
        || path.includes("..") // prevent user from escaping sandbox.
    ) {
        throw new Error("Validation error.");
    }
};

class SourceScope {
    _init(redirects) {
        validate_redirects(redirects);

        // the goal is to get the longer prefixes first to ensure
   ↪  correct precedence.
        this._redirects = redirects.sort((a, b) => b[0].length - a[0].
   ↪  length);
    }

    resolve(path) {
        validate_path(path);

        for (const [prefix, parent_scope, replacement] of this.
   ↪  _redirects) {
            if (path.startsWith(prefix)) {
                if (false
                    || parent_scope === undefined
                    || replacement === undefined
                ) {
                    return undefined;
                }

                const remainder = path.slice(prefix.length);
                const new_path = replacement + remainder;

                return parent_scope.resolve(new_path);
            }
        }

        return undefined;
    }
}

export const create_source_scope = create_unsuspended_factory(
   ↪ SourceScope);
```

compiler/frontend/process.js

```
import { mapData, join, opt, multi, opt_multi, or, declare } from "./
   ↪ uoe/ec/blurp.js";

import * as fs from "fs/promises";
import * as node__path from "path";
import { create_fs_source_scope } from "./create_fs_source_scope.js";
import { create_source_scope } from "./create_source_scope.js";

// LEXICAL TOKENS

const WHITESPACE = /^\s+/;
const SINGLELINE_COMMENT = mapData(/^\s*;(.*?)(\n|$)\s*/, data => data.
   ↪ groups[0]);
const SKIPPERS = opt(multi(or(SINGLELINE_COMMENT, WHITESPACE)));
const CORE_SKIPPERS = opt(multi(WHITESPACE));

const withSkippers = (p) => mapData(join(SKIPPERS, p, SKIPPERS), data
   ↪ => data[1]);
const withCarefulSkippers = (p) => mapData(join(SKIPPERS, p,
   ↪ CORE_SKIPPERS), data => data[1]);
const withLeftSkippers = (p) => mapData(join(SKIPPERS, p), data => data
   ↪ [1]);
const withRightSkippers = (p) => mapData(join(p, SKIPPERS), data =>
   ↪ data[0]);

const KW_FORWARDING = withCarefulSkippers("forwarding");
const KW_RESET = withCarefulSkippers("reset");
const KW_TYPE = withCarefulSkippers("type");
const KW_TYPES = withCarefulSkippers("types");
const KW_MAP = withCarefulSkippers("map");
const KW_IMPORT = withCarefulSkippers("import");
const KW_MOD = withCarefulSkippers("mod");
const LBRACE = withCarefulSkippers("{");
const RBRACE = withCarefulSkippers("}");
const PATH_OUTER_ROOT = withCarefulSkippers(mapData(/^\/\/[a-z0-9_
   ↪ \/\.]*/, data => data.groups.all));
const PATH_MODULE_ROOT = withCarefulSkippers(mapData(/^\/[a-z0-9_
   ↪ \/\.]*/, data => data.groups.all));
const PATH = or(PATH_OUTER_ROOT, PATH_MODULE_ROOT);
const OUTER_ROOT = withCarefulSkippers("//");
const SEMI = withRightSkippers(SINGLELINE_COMMENT);
// i need to go through all these again. why did I put "_" outside the
   ↪ []. why is underscore and numbers not matched in the first part?
const TYPE_IDENT = withCarefulSkippers(mapData(/^(?:([a-z]+(?:_[a-z0
   ↪ -9]+)*::)*[A-Z][a-zA-Z0-9]*)|i[1-9][0-9]{0,4}|u[1-9][0-9]{0,4}|
   ↪ f16|f32|f64|f128/, data => data.groups.all));
const TYPES_IDENT = withCarefulSkippers(mapData(/^(?:([a-z]+(?:_[a-z0
   ↪ -9]+)*::)*(?:[a-z]+(?:_[a-z0-9]+)*))/, data => data.groups.all));
const MOD_IDENT = withCarefulSkippers(mapData(/^(?:([a-z]+(?:_[a-z0
   ↪ -9]+)*::)*(?:[a-z]+(?:_[a-z0-9]+)*))/, data => data.groups.all));
   ↪  // identical to TYPES_IDENT for now.
const SYMBOL = withCarefulSkippers(mapData(/^:([a-zA-Z_][a-zA-Z0-9_]*)
   ↪ /, data => data.groups.all));
const EXTAT = withLeftSkippers("@");
const INTEGER = withCarefulSkippers(mapData(/^[0-9]+/, data => BigInt(
   ↪ data.groups.all)));
const FLOAT = withCarefulSkippers(mapData(/^[0-9]+\.[0-9]+/, data =>
   ↪ data.groups.all));

// PARSER RULES

const symbol_path = mapData(
    multi(
        SYMBOL,
    ),
    (data) => ({
        type: "symbol_path",
        trail: data.map(entry => entry.substring(1)),
    }),
);

const constraint_map = mapData(
    join(
        LBRACE,
        RBRACE,
    ),
    (data) => ({
        type: "constraint_map",
    }),
);

const constraint_integer = mapData(
    INTEGER,
    (data) => ({
        type: "constraint",
        mode: "constraint_integer",
        value: data,
    }),
);

const constraint_float = mapData(
    FLOAT,
    (data) => ({
        type: "constraint",
        mode: "constraint_float",
        value: data,
    }),
);

const constraint_semiless = or(
    constraint_map,
);

const constraint_semiful = or(
    constraint_float,
    constraint_integer,
);

const constraint_maybesemi = or(
    constraint_semiless,
    mapData(
        join(constraint_semiful, SEMI),
        (data) => data[0],
    ),
);

const forwarding = mapData(
    join(
        KW_FORWARDING,
        LBRACE,
        opt_multi(
            or(
                mapData(
                    join(KW_RESET, or(PATH_OUTER_ROOT, OUTER_ROOT),
   ↪ SEMI),
                    (data) => ({ type: "reset", source: data[0] }),
                ),
                mapData(
                    join(PATH_OUTER_ROOT, PATH, SEMI),
                    (data) => ({ type: "redirect", source: data[0],
   ↪ destination: data[1] })
                ),
            )
        ),
        RBRACE,
    ),
    (data) => ({
        type: "forwarding",
        entries: data[2],
    }),
);

const top_forwarding = mapData(
    forwarding,
    (data) => ({
        ...data,
        type: "top_forwarding",
    }),
);

const type_reference = or(
    mapData(
        TYPE_IDENT,
        (data) => ({
            type: "type_reference",
            mode: "type_ident",
            trail: data.split("::"),
        }),
    ),
    mapData(
        KW_MAP,
        (_data) => ({
            type: "type_reference",
            mode: "anon_map",
            leaf_type: undefined,
            call_input_type: undefined,
            call_output_type: undefined,
        }),
    ),
);

const top_type = or(
    mapData(
        join(
            KW_TYPE,
            TYPE_IDENT,
            type_reference,
            SEMI,
        ),
        (data) => ({
            type: "type",
            trail: data[1],
            definition: data[2],
        }),
    ),
    mapData(
        join(
            KW_TYPE,
            TYPE_IDENT,
            type_reference,
            constraint_maybesemi,
        ),
        (data) => ({
            type: "type",
            trail: data[1],
            definition: data[2],
            constraint: data[3],
        }),
    ),
    mapData(
        join(
            KW_TYPE,
            TYPE_IDENT,
            constraint_maybesemi,
        ),
        (data) => ({
            type: "type",
            trail: data[1],
            constraint: data[2],
        }),
    ),
);

// a value is just a stricter version of a type, but it's still a type
   ↪ from the compiler's perspective. (value constraint perhaps i'll
   ↪ call it constraint instead of value).

const top_types = mapData(
    join(
        KW_TYPES,
        TYPES_IDENT,
        or(
            mapData(
                join(TYPES_IDENT, SEMI),
                (data) => ({
                    type: "types_reference",
                    mode: "types_ident",
                    trail: data[0],
                }),
            ),
            mapData(
                join(KW_IMPORT, PATH, SEMI),
                (data) => ({
                    type: "types_reference",
                    mode: "anon_import",
                    path: data[1],
                    forwarding: undefined,
                }),
            ),
            mapData(
                join(KW_IMPORT, PATH, forwarding),
                (data) => ({
                    type: "types_reference",
                    mode: "anon_import",
                    path: data[1],
                    forwarding: data[2],
                }),
            ),
        ),
    ),
    (data) => ({
        type: "top_types",
        trail: data[1],
        definition: data[2],
    }),
);

const top_mod = mapData(
    join(
        KW_MOD,
        MOD_IDENT,
        or(
            mapData(
                join(KW_IMPORT, PATH, SEMI),
                (data) => ({
                    type: "mod_reference",
                    mode: "anon_import",
                    path: data[1],
                    forwarding: undefined,
                }),
            ),
            mapData(
                join(KW_IMPORT, PATH, forwarding),
                (data) => ({
                    type: "mod_reference",
                    mode: "anon_import",
                    path: data[1],
                    forwarding: data[2],
                }),
            ),
        ),
    ),
    (data) => ({
        type: "top_mod",
        trail: data[1],
        definition: data[2],
    }),
);

const case_ = or(
    mapData(
        join(
            symbol_path,
            type_reference,
            SEMI,
        ),
        (data) => ({
            type: "case",
            symbol_path: data[0],
            type_reference: data[1],
        }),
    ),
    mapData(
        join(
            symbol_path,
            type_reference,
            constraint_maybesemi,
        ),
        (data) => ({
            type: "case",
            symbol_path: data[0],
            type_reference: data[1],
            constraint: data[2],
        }),
    ),
);

const top_extension = mapData(
    join(
        EXTAT,
        TYPE_IDENT,
        case_,
    ),
    (data) => ({
        type: "top_extension",
        target_type: data[1],
        case: data[2],
    })
);

const top_entry = or(
    top_forwarding,
    top_type,
    top_types,
    top_mod,
    top_extension,
    SEMI,
);

// i wonder if values and maps could be combined?
// but for now, value_map will just be optional after type.

const file_root = opt_multi(top_entry);

// SEMANTIC ANALYSIS

// COMPILER

// OUTER INTERFACE

// class ModuleLoader {
//  constructor() {

//  }

//  obtain_module_files(path) {

//  }
// }

// create_sources_scope([
//  ["//"], // reset
//  ["/", parent_scope, "/"],
//  ["/banana", parent_scope, "/prefix/on/parent/scope/"],
//  ["//octopus", parent_scope, "//prefix/on/parent/scope/"],

// ]);

const make_structured = (entries) => {
    const structured = {
        structure: entries,
        forwarding: [],
    };

    for (const entry of entries) {
        if (entry.type === "top_forwarding") {
            structured.forwarding = [
                ...structured.forwarding,
                ...entry.entries,
            ];
        }
    }

    return structured;
};

const augment = (data) => {
    const module_source_scope = create_source_scope([
        ["//", data.source_scopes.external, "//"],
        ["/", data.source_scopes.internal, "/"],
    ]);

    const forwarding_source_scope = create_source_scope(data.forwarding
   ↪ .map(entry => {
        if (entry.type === "reset") {
            return [entry.source];
        } else if (entry.type === "redirect") {
            return [entry.source, module_source_scope, entry.
   ↪ destination];
        }

        console.warn("Entry:", entry);
        throw new Error("Unhandled case.");
    }));
    return {
        ...data,
        source_scopes: {
            ...data.source_scopes ?? [],
            module: module_source_scope,
            forwarding: forwarding_source_scope,
        },
    };
};

export const process = async (config) => {
    const path = config.src;
    const actual_path = node__path.resolve(path);
    console.log("[FE] Processing module located at path:", actual_path)
   ↪ ;

    const file_system_source_scope = create_fs_source_scope({
        fs,
        path: node__path
    }, actual_path);

    const internal_source_scope = create_source_scope([
        ["//"],
        ["/", file_system_source_scope, "./"],
    ]);

    const result = await internal_source_scope.resolve("/");

    console.log(result);

    const parse_results = [];

    for (const file of result.files) {
        console.log("Parsing file", file);

        const text = (await fs.readFile(node__path.join(actual_path,
   ↪ file), "utf-8")).replaceAll("\r\n", "\n");
        const result = file_root(text);

        if (false
            || result.success === false
            || result.input !== ""
        ) {
            console.error(result);
            throw new Error(‘Failed to parse.‘);
        }
        parse_results.push(result.data);
    }

    console.log("Augmenting module data...");

    const combined = parse_results.flat();

    const structured = make_structured(combined);

    structured.module_path = actual_path;

    structured.source_scopes = {
        internal: internal_source_scope,
    };

    const final = augment(structured);

    return final;
};
```

compiler/package.json

```
{
    "type": "module",
    "dependencies": {
        "commander": "^14.0.2"
    }
}
```

compiler/test/a.ec

```
forwarding {
    reset //;
}

forwarding {
    reset //;
    reset //foo/bar/baz;
    //banana/slop /wanana/bop;
}

type foo::Bar f32;
type foo::bar::Baz Banana;
type foo::bar::Bink foo::Bar;
type foo::Coconut map;
```

```
types egg::plant foo::bar;

types banana import /foo/bar;

types orange import /foo/bar forwarding {
    reset //;
    reset //foo/bar/baz;
    //banana/slop /wanana/bop;
}

mod eggplant import //egg/plant;

mod eggplant import //egg/plant forwarding {
    //pear /pear;
}

; todo why is eggplant2 not working?
```

compiler/test/b.ec

```
forwarding {
    //banana/slop /wanana/bop;
    //banana/vegetable /wanana/gravy;
}

@Bar:test:blah27 f32;

@Bar:foo blah::Wah;

type NewType f32 {}
; type NewType map {}

type NewTypeAgain f32 5;
; type NewTypeAgain map 5;

type AnotherNewType f32 5;
type AnotherNewTypeAlright 5;

type Blah 5.5;

@Car:why_is_this_disappearing i32;
@Car:testing f32 5;
; @Car:testing 5;
```

extension/LICENSE.txt

```
no license given
```

extension/README.md

```
# Extension

This directory houses the VS Code extension for my language, which
   ↪ provides syntax highlighting to the programmer.

It is unlikely that any additional extension features would be worked
   ↪ on, as that would be out of the scope of this project.

## Setup

- Ensure ‘vsce‘ command is set up on your system.

## Build

- ‘vsce package‘

## Install

- Within VS Code, right click the generated ‘.vsix‘ file in the
   ↪ explorer tree.
- Select "Install Extension VSIX".
- You may need to reload your window to see the changes take effect.

## Structure

- The ‘language-configuration.json‘ file contains some fundamental
   ↪ syntax rules that VS Code needs to know about.
- The ‘syntaxes/essencec.tmLanguage.json‘ file contains the meat up the
   ↪  syntax and also assigns appropriate colors to the syntax.

## Autogenerated Files

The ‘yo code‘ command was used to generate the boilerplate for this
   ↪ directory.

Thus, some files were auto-generated by the CLI.

Some other files were scaffolded by the CLI but have since been
   ↪ modified significantly.
```

extension/language-configuration.json

```
{
    "comments": {
        // symbol used for single line comment. Remove this entry if your language does not support line comments
        "lineComment": ";",
    },
    // symbols used as brackets
    "brackets": [
        [
            "{",
            "}"
        ],
        [
            "[",
            "]"
        ],
        [
            "(",
            ")"
        ]
    ],
    // symbols that are auto closed when typing
    "autoClosingPairs": [
        [
            "{",
            "}"
        ],
        [
            "[",
            "]"
        ],
        [
            "(",
            ")"
        ],
        [
            "\"",
            "\""
        ],
        [
            "'",
            "'"
        ]
    ],
    // symbols that can be used to surround a selection
    "surroundingPairs": [
        [
            "{",
            "}"
        ],
        [
            "[",
            "]"
        ],
        [
            "(",
            ")"
        ],
        [
            "\"",
            "\""
        ],
        [
            "’",
            "’"
        ]
    ]
}
```

extension/package.json

```
{
  "publisher": "blabidy",
  "name": "essence-c",
  "displayName": "essence-c",
  "description": "",
  "version": "0.0.1",
  "engines": {
    "vscode": "^1.96.2"
  },
  "categories": [
    "Programming Languages"
  ],
  "contributes": {
    "languages": [
      {
        "id": "essencec",
        "aliases": [
          "Essence C",
          "essencec"
        ],
        "extensions": [
          ".ec",
          ".eh"
        ],
        "configuration": "./language-configuration.json"
      }
    ],
    "grammars": [
      {
        "language": "essencec",
        "scopeName": "source.essencec",
        "path": "./syntaxes/essencec.tmLanguage.json"
      }
    ]
  },
  "repository": "https://google.com/"
}
```

extension/syntaxes/essencec.tmLanguage.json

```
{
    "$schema": "https://raw.githubusercontent.com/martinring/tmlanguage
  ↪ /master/tmlanguage.json",
    "name": "Essence C",
    "patterns": [
        {
            "include": "#keywords"
        },
        {
            "include": "#strings"
        },
        {
            "name": "comment.line.content.essencec",
            "match": "^\\s*;(.*)"
        },
        {
            "match": "(;)(.*)",
            "captures": {
                "1": {
                    "name": "punctuation.separator.delimiter.essencec"
                },
                "2": {
                    "name": "comment.line.content.essencec"
                }
            }
        },
        {
            "name": "entity.name.qualified.custom",
            "begin": "\\b([a-z_][a-z0-9_]*)(::)",
            "beginCaptures": {
                "1": {
                    "name": "entity.name.namespace.component.custom"
                },
                "2": {
                    "name": "punctuation.separator.namespace.custom"
                }
            },
            "end": "(?!::[A-Za-z_][a-zA-Z0-9_]*)",
            "patterns": [
                {
                    "match": "[a-z][a-zA-Z0-9_]*",
                    "name": "entity.name.namespace.component.custom"
                },
                {
                    "match": "[A-Z][a-zA-Z0-9_]*",
                    "name": "entity.name.namespace.component.custom"
                },
                {
                    "match": "::",
                    "name": "punctuation.separator.namespace.custom"
                }
            ]
        },
        {
            "name": "storage.type.object.array.java",
            "match": "[A-Z]([a-z0-9_])*"
        },
        {
            "name": "storage.type.object.array.java",
            "match": "\\b(i[1-9][0-9]{0,4}|u[1-9][0-9]{0,4}|f16|f32|f64|f128)\\b"
        },
        {
            "name": "entity.name.tag",
            "match": "\\b(this|mod)\\b"
        },
        {
            "name": "entity.name.function",
            "match": ":[a-z0-9_]+"
        },
        {
            "name": "entity.name.function",
            "match": ":\\s"
        },
        {
            "begin": "(\\/\\/)(([a-z0-9_\\.])+)",
            "beginCaptures": {
                "1": {
                    "name": "keyword.control"
                },
                "2": {
                    "name": "entity.name.function"
                }
            },
            "end": "(?![a-z0-9_\\/\\.])",
            "patterns": [
                {
                    "name": "entity.name.function",
                    "match": "[a-zA-Z0-9_\\.]+"
                },
                {
                    "name": "keyword.control",
                    "match": "\\/"
                }
            ]
        },
        {
            "begin": "(\\/)(([a-z0-9_\\.])+)",
            "beginCaptures": {
                "1": {
                    "name": "storage.type.object.array.java"
                },
                "2": {
                    "name": "entity.name.function"
                }
            },
            "end": "(?![a-z0-9_\\/\\.])",
            "patterns": [
                {
                    "name": "entity.name.function",
                    "match": "[a-zA-Z0-9_\\.]+"
                },
                {
                    "name": "storage.type.object.array.java",
                    "match": "\\/"
                }
            ]
        },
        {
            "name": "keyword.control",
            "match": "\\/\\/"
        },
        {
            "name": "entity.name.tag",
            "match": "@"
        }
    ],
    "repository": {
        "keywords": {
            "patterns": [
                {
                    "name": "keyword.control.essencec",
                    "match": "\\b(in|if|for|return|import|forwarding|
  ↪ create)\\b"
                },
                {
                    "name": "keyword.modifier.essencec",
                    "match": "\\b(instantiate|reset|types|type|pub|map|
  ↪ enum|mut)\\b"
                }
            ]
        },
        "strings": {
            "name": "string.quoted.double.essencec",
            "begin": "\"",
            "end": "\"",
            "patterns": [
                {
                    "name": "constant.character.escape.essencec",
                    "match": "\\\\."
                }
            ]
        }
    },
    "scopeName": "source.essencec"
}
```

C Appendix: LLM Prompts

Thread (42 messages)
ID: 7add1779-4337-44f1-a533-c37c4eb71efe **2025/06/14 23:01**

USER (Message 1) [gemini-2.5-flash] 2025/06/14 23:01

```
does clang go through llvm? how on earth can assembly be inlined in
   ↪ this case? does llvm support an assembly inlining mechanism?
```

AGENT (Message 2) [gemini-2.5-flash] 2025/06/14 23:01

```
Yes, Clang uses LLVM as its backend. When you compile C, C++, or
   ↪ Objective-C code with Clang, it first parses your code and
   ↪ converts it into an Abstract Syntax Tree (AST). This AST is then
   ↪ lowered into LLVM Intermediate Representation (IR). LLVM IR is a
   ↪ low-level, assembly-like language that is independent of any
   ↪ specific CPU architecture.

From this LLVM IR, LLVM's optimization passes analyze and transform the
   ↪  code to improve its performance. Finally, the LLVM backend for
   ↪ your target architecture (e.g., x86, ARM) takes the optimized
   ↪ LLVM IR and generates machine code (assembly).

### Assembly Inlining

You're right to ask how assembly can be inlined when the compilation
   ↪ process goes through an intermediate representation like LLVM IR.
   ↪  LLVM does indeed have a mechanism to support assembly inlining.
   ↪ Here's how it generally works:

When you use inline assembly in your C/C++ code (using ‘asm‘ or ‘
   ↪ __asm__‘ keywords), Clang recognizes this and, instead of trying
   ↪ to translate the assembly instructions into LLVM IR, it
   ↪ effectively passes them through to the LLVM backend.

The LLVM IR itself doesn't directly represent every assembly
   ↪ instruction. Instead, it has an intrinsic called ‘llvm.
   ↪ experimental.asm‘ or similar, which acts as a placeholder for the
   ↪  inline assembly block. This intrinsic carries the actual
   ↪ assembly string, along with information about its constraints (
   ↪ inputs, outputs, clobbered registers).

When the LLVM backend generates machine code from the LLVM IR, it
   ↪ encounters these assembly intrinsics. At this point, it takes the
   ↪  assembly string provided by the intrinsic and inserts it
```

```
  ↪ directly into the generated assembly output.

**Example (simplified conceptual flow):**

1.  **C/C++ Code:**

    ‘‘‘c++
    int add_one(int x) {
        int result;
        asm("addl $1, %0" : "=r"(result) : "0"(x));
        return result;
    }
    ‘‘‘

2.  **Clang's Handling:** Clang recognizes the ‘asm‘ block. It
  ↪ translates the surrounding C++ code into LLVM IR, but for the ‘
  ↪ asm‘ block, it generates an ‘llvm.experimental.asm‘ intrinsic.

    *(This is a highly simplified representation of the intrinsic, the
  ↪ actual LLVM IR is more complex with operand bundles and
  ↪ constraints)*

    ‘‘‘llvm
    ; ... other LLVM IR instructions ...
    %result = call i32 @llvm.experimental.asm(metadata !"addl $1, $0",
  ↪ metadata !"=r,0"(i32 %x))
    ; ...
    ‘‘‘

3.  **LLVM Backend's Handling:** When the x86 backend processes this
  ↪ LLVM IR, it sees the ‘llvm.experimental.asm‘ intrinsic. It
  ↪ extracts the assembly string ‘"addl $1, %0"‘ and the constraint
  ↪ information. It then inserts the ‘addl‘ instruction directly into
  ↪  the generated x86 assembly.

    ‘‘‘assembly
    ; ... other x86 assembly instructions ...
    addl $1, %eax   ; This comes from the inline asm
    ; ...
    ‘‘‘

### LLVM's Assembly Inlining Mechanism

LLVM's design for handling inline assembly is quite sophisticated:

*   **Constraints:** You provide constraints (like ‘=r‘ for an output
  ↪ register, ‘0‘ for input reuse) to tell the compiler how the
```

```
    ↪ assembly interacts with C/C++ variables. LLVM uses this
    ↪ information to correctly map registers and memory locations.
*   **Volatile Assembly:** The ‘volatile‘ keyword in inline assembly
    ↪ prevents the compiler from reordering or optimizing away the
    ↪ assembly instruction, ensuring it executes exactly where
    ↪ specified.
*   **Clobber List:** You specify a clobber list to inform the compiler
    ↪  which registers or memory locations your inline assembly
    ↪ modifies that aren’t explicitly listed as outputs. This prevents
    ↪ the compiler from making incorrect assumptions about the state of
    ↪  those resources.
*   **Target-Specific Assembly:** The beauty of this approach is that
    ↪ you write assembly specific to your target architecture. LLVM’s
    ↪ role is to ensure that this target-specific assembly is correctly
    ↪  integrated into the generated machine code without trying to "
    ↪ understand" or optimize the assembly instructions themselves.

In essence, LLVM provides a well-defined interface for "punching
    ↪ through" its IR and directly inserting architecture-specific
    ↪ assembly instructions, while still managing the data flow and
    ↪ register allocation around those inlined blocks.
```

USER (Message 3) [gemini-2.5-flash] 2025/06/14 23:04

```
i am creating a custom language, and i actually want to have a similar
    ↪ asm feature but instead it would be inserting the raw assembly
    ↪ bytes, does llvm support giving it the raw assembly with no fancy
    ↪  mechanisms for passing in data or anything into it. like it has
    ↪ to find the data at its own will via known registers and stuff
    ↪ based on the cpu
```

AGENT (Message 4) [gemini-2.5-flash] 2025/06/14 23:04

```
That’s an interesting approach for a custom language! You’re
    ↪ essentially asking for a way to embed raw machine code bytes
    ↪ directly into the generated executable, bypassing the assembly
    ↪ language layer and LLVM’s register allocation and constraint
    ↪ system for inline assembly.

While LLVM’s primary inline assembly mechanism (‘llvm.experimental.asm‘
    ↪  or similar) is designed to handle assembly *strings* with
    ↪ constraints, which then get assembled by the backend, directly
    ↪ injecting *raw bytes* is not a first-class, widely exposed
    ↪ feature in the same way. The reason is that LLVM generally works
    ↪ at a higher level of abstraction (IR) and then relies on its
    ↪ backends to generate the specific machine code.
```

```
However, there are a few ways you *might* achieve something similar or
   ↪ work around this, though they come with significant caveats and
   ↪ are not typical LLVM usage:

### 1. Using an ‘llvm.experimental.asm‘ Intrinsic with Byte-Generating
   ↪ Assembly

This is still using the ‘llvm.experimental.asm‘ intrinsic, but instead
   ↪ of providing human-readable assembly instructions, you would
   ↪ provide assembly directives that encode raw bytes. This is highly
   ↪  dependent on the specific assembler used by the LLVM backend for
   ↪  your target.

For example, in GNU Assembler syntax (often used by LLVM's backends for
   ↪  x86/ARM), you can use ‘.byte‘ directives.

**Conceptual LLVM IR (very simplified):**

‘‘‘llvm
; This is an illustrative example, not actual working IR.
; The actual intrinsic would still require a constraint string,
; but the 'assembly' part would contain byte directives.

call void @llvm.experimental.asm(
  metadata !"\\0x90\\0x90\\0x90\\0x90", ; raw NOP bytes
  metadata !""                        ; no constraints/operands
)
‘‘‘

**Challenges with this approach:**

*   **Assembler Dependence:** You're reliant on the target assembler's
   ↪ syntax for embedding raw bytes (e.g., ‘.byte‘, ‘.long‘, ‘.word‘,
   ↪ ‘.inst‘). This makes your "raw byte" feature less portable across
   ↪  different LLVM backends if they use different internal
   ↪ assemblers or syntaxes.
*   **No Register/Memory Management:** As you noted, there are no fancy
   ↪  mechanisms for passing data. You'd have to manually manage
   ↪ registers and memory within your raw bytes, assuming you know the
   ↪  CPU state.
*   **Optimization Barriers:** The LLVM optimizer will treat this as an
   ↪  opaque block. It won't understand the semantics of the bytes and
   ↪  might reorder it unless you use appropriate ‘volatile‘ markers (
   ↪ which you'd have to ensure are correctly passed to the ‘llvm.
   ↪ experimental.asm‘ intrinsic).
*   **Target-Specific:** You are essentially generating target-specific
   ↪  machine code directly in your custom language's front-end, then
```

```
    ↪ encoding it as an assembly string. This ties your language
    ↪ directly to the target architecture at a very low level.

### 2. Using Global Data with Section Attributes and Function Pointers

This approach doesn’t involve ‘llvm.experimental.asm‘ directly but
    ↪ rather leverages LLVM’s ability to define global data and then
    ↪ potentially execute it. You would define a global array of bytes,
    ↪  mark it with appropriate section attributes to make it
    ↪ executable, and then cast a pointer to it as a function pointer.

**Conceptual LLVM IR:**

‘‘‘llvm
; Define a global array of bytes
@my_raw_code = private constant [4 x i8] c"\90\90\90\90", align 16

; Tell the linker to put this in an executable section
; This often involves target-specific section names or attributes
; (e.g., ".text", ".data..code")
; In LLVM IR, you might use ’section’ attribute on the global:
; @my_raw_code = private constant [4 x i8] c"\90\90\90\90", section ".
    ↪ text"

define void @execute_raw_code() {
  %ptr = getelementptr inbounds [4 x i8], ptr @my_raw_code, i64 0, i64 0
  %func_ptr = bitcast ptr %ptr to void ()*
  call void %func_ptr()
  ret void
}
‘‘‘

**Challenges with this approach:**

*   **Executable Data Section:** Getting the linker to mark data as
    ↪ executable is crucial and can be tricky. It often involves target
    ↪ -specific section names or special linker flags. You might need
    ↪ to add attributes like ‘section ".text"‘ to your global variable
    ↪ definition in LLVM IR, or even create a custom linker script.
*   **Permissions:** At runtime, the memory page containing this data
    ↪ must have executable permissions.
*   **Security:** Executing data directly is a common security
    ↪ vulnerability (e.g., buffer overflows leading to arbitrary code
    ↪ execution). If your language aims for safety, this feature would
    ↪ be a significant security risk.
*   **Relocation:** Your raw bytes cannot contain any relative
    ↪ addresses or references to symbols unless you manually handle
    ↪ relocations. This means self-modifying code or position-
    ↪ independent code for your raw bytes would be incredibly complex
    ↪ to manage.
*   **No Compiler Optimizations:** The compiler will treat this as
    ↪ opaque data and opaque function calls. It cannot optimize the
    ↪ code within your raw bytes.
*   **Function Calls:** If your raw bytes need to call into other
    ↪ functions, you'll need to manually calculate their addresses or
    ↪ use position-independent code techniques.

### 3. Using LLVM's JIT (Just-In-Time) API (Advanced)

If your custom language has a runtime component or interpreter, and you
    ↪ 're using LLVM's JIT capabilities, you could potentially inject
    ↪ raw bytes into memory and execute them through the JIT's memory
    ↪ manager. This is far more complex as it bypasses the standard
    ↪ compilation flow.

**Challenges:**

*   **Extremely Advanced:** This requires deep knowledge of LLVM's ORC
    ↪ JIT API or MCJIT.
*   **Memory Management:** You'd be responsible for allocating
    ↪ executable memory, copying your bytes, and managing their
    ↪ lifetime.
*   **No Compiler Support:** The LLVM compiler itself isn't generating
    ↪ these bytes; your runtime would be doing it.

### Recommendation

Given your goal of "inserting the raw assembly bytes" where your code
    ↪ has to "find the data at its own will via known registers and
    ↪ stuff based on the cpu," the first option (using `llvm.
    ↪ experimental.asm` with byte-generating assembly directives) is
    ↪ probably the *least difficult* way to achieve direct byte
    ↪ injection *through LLVM's compilation pipeline*.

However, it's critical to understand the implications:

*   **Your custom language's "asm" feature would be highly target-
    ↪ specific.** The raw bytes you input would only work on the CPU
    ↪ architecture for which they were designed.
*   **You lose all the benefits of LLVM's optimization passes** for
    ↪ these raw byte blocks.
*   **You are responsible for every single detail** of the machine code
    ↪ , including register usage, stack frame setup, function calls,
    ↪ and data access.
```

This is a very powerful but dangerous feature. Most high-level languages that offer an "assembly" feature still compile human-readable assembly strings (like C's ‘asm‘) and let the toolchain handle the final byte generation and integration, precisely because of the complexity of dealing with raw bytes directly.

If you proceed, thoroughly research the assembler syntax for embedding raw bytes for your target architecture (e.g., ‘.byte‘, ‘.incbin‘) and how LLVM's backend handles ‘llvm.experimental.asm‘ with such directives.

USER (Message 5) [gemini-2.5-flash] 2025/06/14 23:06

this is great. however, its not truly the raw bytes i want i dont think ? does dynamic addresses and stuff require an ever so slightly different mechanism to be considered? i.e. what if want to jump to a function... mmmm maybe i do need to pass data

AGENT (Message 6) [gemini-2.5-flash] 2025/06/14 23:06

You've hit on the core reason why most "inline assembly" mechanisms *don't* just take raw bytes without any "fancy mechanisms for passing in data." The moment you need to interact with the broader program's context like jumping to a function, accessing variables, or calling external libraries you run into the problem of **addresses and relocations**.

Let's break down why "truly raw bytes" are problematic for dynamic addresses and what mechanisms help.

### The Problem with "Truly Raw Bytes" and Dynamic Addresses

Imagine you have a piece of raw machine code bytes like ‘\xCD\x21‘ (an old DOS ‘int 21h‘ interrupt call). If you simply inject these bytes, they will execute exactly as they are. But what if you want to call a C function named ‘my_c_function‘?

Your raw bytes might try to include an instruction like ‘call my_c_function‘. On an x86-64 system, a ‘call‘ instruction might look like ‘E8 [offset]‘ where ‘[offset]‘ is a 32-bit signed offset relative to the instruction pointer.

The problem is:

1. **Where is ‘my_c_function‘?** Its address isn't known until the program is linked, and possibly even until it's loaded into

```
   ↪ memory (if it’s a dynamically linked library).
2.  **What’s the offset?** You can’t hardcode this offset into your raw
   ↪  bytes because you don’t know the final address of your ‘call‘
   ↪ instruction or ‘my_c_function‘ at the time you generate the raw
   ↪ bytes.

This is precisely what **relocation** is for. When you compile and link
   ↪  a program, the linker’s job is to fill in these unknown
   ↪ addresses and offsets.

### How Relocation Works (and why "fancy mechanisms" exist)

When you write assembly like ‘call my_c_function‘, the assembler doesn’
   ↪ t directly insert the final offset. Instead, it inserts a
   ↪ placeholder (often zeros or a dummy value) and generates **
   ↪ relocation entries** in the object file. A relocation entry tells
   ↪  the linker: "At this specific byte offset within this section,
   ↪ there’s a ‘call‘ instruction that needs to be updated with the
   ↪ address of ‘my_c_function‘ (or an offset to it)."

The "fancy mechanisms" in ‘asm‘ blocks (like constraints, output
   ↪ operands, input operands) are precisely how you communicate to
   ↪ the compiler and linker about these dynamic needs:

*   **Passing data in/out:** ‘asm("addl %1, %0" : "=r"(result) : "r"(x)
   ↪ );‘
    *   ‘%0‘ and ‘%1‘ are placeholders that LLVM’s backend will replace
   ↪  with actual register names or memory locations. LLVM handles the
   ↪  register allocation for ‘result‘ and ‘x‘.
    *   The ‘=‘ and ‘r‘ constraints tell LLVM: "Allocate a general-
   ↪ purpose register for ‘result‘ (output), and ‘x‘ (input)."

*   **Referencing labels/functions:**
    *   If you have a local label in assembly, ‘jmp .mylabel‘, the
   ↪ assembler calculates the offset for you.
    *   If you call an external function, ‘call some_function‘, the
   ↪ assembler generates a relocation entry, and the linker resolves
   ↪ it.

### Your Goal: "Finding the data at its own will via known registers
   ↪ and stuff based on the CPU"

This is *Position-Independent Code (PIC)* or making assumptions about
   ↪ the calling convention.

*   **Position-Independent Code (PIC):** Modern code (especially shared
   ↪  libraries) is often compiled as PIC. This means it doesn’t rely
```

~~~
  ↪ on absolute addresses. Instead, it uses relative jumps/calls or
  ↪ accesses data via pointers stored in special tables (like the
  ↪ Global Offset Table - GOT - or Procedure Linkage Table - PLT).
  ↪ Writing PIC machine code by hand (without the assembler/compiler'
  ↪ s help) is incredibly complex.

*   **Known Registers / Calling Convention:** If your raw bytes are
  ↪ meant to be a function, they must adhere to the **Application
  ↪ Binary Interface (ABI)** or calling convention of your target
  ↪ system. This specifies:
    *   Which registers are used for function arguments.
    *   Which registers hold the return value.
    *   Which registers a function can clobber (modify) and which it
  ↪ must preserve.
    *   How the stack is used for local variables and saving registers.

    If your raw bytes *assume* `arg1` is in `rdi` (x86-64 Linux/System
  ↪ V ABI), then you're hardcoding for that ABI. If you want your raw
  ↪  bytes to access a global variable, you'd need its absolute
  ↪ address (which implies relocation) or a PIC mechanism to find it.

### What You Likely Need (and how LLVM helps)

If you want to jump to a function or access dynamic addresses from your
  ↪  custom language's "raw assembly" feature, you *do* need some "
  ↪ fancy mechanism" to handle relocations.

**The `llvm.experimental.asm` intrinsic is still your best bet for this
  ↪ **, but you'll use it to leverage the assembler's features for
  ↪ addressing symbols, not just raw bytes.

Instead of writing:

```c
// Hypothetical raw byte for a call to address 0x12345678
// This is not portable or relocatable!
asm(".byte 0xE8, 0x78, 0x56, 0x34, 0x12");
~~~

You would write:

```c
// This is standard GNU AS syntax, which LLVM backends understand
asm("call my_c_function");
```

How `llvm.experimental.asm` handles this:
```
```

```
When you provide an assembly string like ‘"call my_c_function"‘ to ‘
    ↪ llvm.experimental.asm‘, LLVM’s backend, when generating the final
    ↪  machine code:

1.  **Recognizes ‘my_c_function‘ as a symbol.**
2.  **Inserts a placeholder** (e.g., zeros) into the machine code where
    ↪  the target address/offset of ‘my_c_function‘ should go.
3.  **Generates a relocation entry** in the object file. This entry
    ↪ tells the linker: "At this spot in the ‘.text‘ section, update it
    ↪  with the address of ‘my_c_function‘."

**This is the crucial step that "raw bytes" on their own cannot do.**
    ↪ Raw bytes are static; they don’t carry relocation information.

### Summary of What You’d Pass to ‘llvm.experimental.asm‘

If your custom language supports something like:

‘‘‘
asm {
    mov rax, 1
    call my_runtime_func
    add rbx, rcx
    ret
}
‘‘‘

Your compiler for the custom language would transform this into an LLVM
    ↪  IR ‘call‘ to ‘llvm.experimental.asm‘, providing the entire
    ↪ string:

‘‘‘llvm
call void @llvm.experimental.asm(
  metadata !"mov rax, 1\ncall my_runtime_func\nadd rbx, rcx\nret",
  metadata !""  ; Assuming no C/custom-language variables are directly
    ↪ used via constraints
)
‘‘‘

**Key Takeaways:**

*   **Truly "raw bytes" implies *no* dynamic addresses.** They are
    ↪ hardcoded values.
*   **To handle dynamic addresses (like function calls, global variable
    ↪  access), you *need* the help of the assembler and linker’s
    ↪ relocation mechanisms.**
```

~~~~
*   **‘llvm.experimental.asm‘ with assembly *strings* is the intended
   ↪ way to achieve this**, as it allows you to use the target
   ↪ assembler’s syntax, which in turn generates the necessary
   ↪ relocation entries.
*   You’ll still have to deal with **calling conventions (ABIs)** for
   ↪ your "raw" assembly functions to interact correctly with
   ↪ functions generated by LLVM. You’ll need to know which registers
   ↪ store arguments, return values, etc., for your target CPU and OS.

So, while you might not directly *pass data* in the C-style ‘asm‘
   ↪ constraint sense, you *are* passing symbolic names (like ‘
   ↪ my_c_function‘) to the assembler, and that’s its mechanism for "
   ↪ passing" or resolving dynamic address information.
~~~~

USER (Message 7) [gemini-2.5-flash] 2025/06/14 23:07

~~~~
what does inlining some assembly code look like in llvm ir?
~~~~

AGENT (Message 8) [gemini-2.5-flash] 2025/06/14 23:07

~~~~
Inlining assembly code in LLVM IR typically looks like a call to an
   ↪ intrinsic function that represents the inline assembly block. The
   ↪  specific intrinsic used is ‘llvm.experimental.asm‘.

Here’s a breakdown of what that looks like and the key components:

### The ‘llvm.experimental.asm‘ Intrinsic

The ‘llvm.experimental.asm‘ intrinsic is declared as:

‘‘‘llvm
declare <return type> @llvm.experimental.asm(<argument types>...)
‘‘‘

However, it’s not a regular function call. It’s a special intrinsic
   ↪ that tells LLVM’s backend to insert the provided assembly string
   ↪ directly into the generated machine code. The arguments to this
   ↪ intrinsic convey the assembly code, constraints, and potentially
   ↪ Clobbered registers.

Let’s look at an example using a common inline assembly snippet from C/
   ↪ C++:

**Original C Code:**

‘‘‘c
// Assuming x86-64 architecture
~~~~

```
int add_and_get_eax(int x) {
    int result;
    // Add 5 to x, then put it in EAX, then move EAX to result
    // (This is a simplified and somewhat redundant example for
   ↪ demonstration)
    asm volatile (
        "addl $5, %1\n\t"    // Add 5 to input %1 (which is x)
        "movl %%eax, %0"     // Move EAX to output %0 (result)
        : "=r"(result)       // Output: result in a general-purpose
   ↪ register
        : "r"(x)             // Input: x in a general-purpose register
        : "eax"              // Clobber list: EAX register is modified
    );
    return result;
}
‘‘‘

**Corresponding LLVM IR (simplified and illustrative):**

The LLVM IR generated for the ‘asm‘ block might look something like
   ↪ this. Note that the actual IR can be more complex due to operand
   ↪ bundles, more precise type handling, and specific target features
   ↪ .

‘‘‘llvm
; Function signature
define dso_local i32 @add_and_get_eax(i32 %x) {
entry:
  ; %x is already an i32 value in an LLVM register
  ; The inline assembly is represented by a call to llvm.experimental.
   ↪ asm
  ;
  ; Arguments:
  ; 1. The assembly string itself. Newlines (\n) and tabs (\t) are
   ↪ important.
  ; 2. The constraint string. This specifies how operands are handled.
  ;    - "=r": output, general-purpose register
  ;    - "r": input, general-purpose register
  ;    - "~{eax}": clobbers eax. Note that in IR, clobbers are part of
   ↪ the constraint string.
  ; 3. The actual operands (input/output/etc.) as LLVM values.
  ;    - %result is returned by the intrinsic, so it's the result of
   ↪ the call.
  ;    - %x is passed as an argument.

  %result = call i32 @llvm.experimental.asm(
    metadata !"addl $5, %1\n\tmovl %%eax, %0",  ; Assembly string
```

```
      metadata !"=r,r,~{eax}",                              ; Constraint string
      i32 %x                                               ; Operands (only inputs
    ↪  appear here as arguments)
      ; Output operand is the return value of the intrinsic
    ) #0

    ret i32 %result
  }

  ; Declaration for the intrinsic (often implicitly available or
    ↪ generated by clang)
  ; The actual signature of the intrinsic depends on the return type and
    ↪ number/types of operands.
  declare i32 @llvm.experimental.asm(metadata, metadata, ...)
  ‘‘‘

  ### Key Components of ‘llvm.experimental.asm‘ in LLVM IR:

  1.  **Assembly String (‘metadata !"‘...)**:
      *   This is the literal string of assembly instructions you
    ↪ provided in your source code.
      *   It often contains placeholders (like ‘%0‘, ‘%1‘, etc., or ‘$0‘,
    ↪  ‘$1‘ depending on the target and assembler flavor) which
    ↪ correspond to the operands specified in the constraints.
      *   It can include standard assembly directives supported by the
    ↪ target assembler.
      *   Newlines (‘\n‘) and tabs (‘\t‘) within the string help format
    ↪ it for the final assembler.

  2.  **Constraint String (‘metadata !"‘...)**:
      *   This is a highly compact and important string that describes
    ↪ how the assembly block interacts with LLVM IR values (inputs,
    ↪ outputs) and modifies the machine state (clobbered registers/
    ↪ memory).
      *   **Output constraints:** Start with ‘=‘. E.g., ‘=r‘ (output in
    ↪ general-purpose register), ‘=m‘ (output to memory).
      *   **Input constraints:** E.g., ‘r‘ (input in general-purpose
    ↪ register), ‘i‘ (immediate integer), ‘m‘ (memory operand).
      *   **Matching constraints:** Numbers (e.g., ‘0‘, ‘1‘) indicate
    ↪ that an operand uses the *same register/memory location* as a
    ↪ previous operand.
      *   **Clobber list:** These describe registers or memory that the
    ↪ assembly modifies but are not explicitly listed as outputs. They
    ↪ are indicated by ‘~{register_name}‘ (e.g., ‘~{eax}‘, ‘~{memory}‘)
    ↪ . This tells the optimizer that these resources might be changed,
    ↪  so it cannot rely on their prior values.
      *   The constraints are comma-separated. The order typically
    ↪ matches the order of the actual LLVM IR operands and the
    ↪ placeholders in the assembly string.

3.  **Operands (LLVM IR values)**:
    *   These are the actual LLVM IR values (e.g., ‘%x‘ in the example)
    ↪  that correspond to the inputs and outputs defined by the
    ↪ constraints.
    *   Input operands appear as arguments to the ‘call‘ instruction.
    *   The output operand is typically the return value of the ‘llvm.
    ↪ experimental.asm‘ call.

4.  **‘volatile‘ Flag (often inferred or handled by context)**:
    *   In C/C++, the ‘volatile‘ keyword on ‘asm‘ prevents the compiler
    ↪  from optimizing away or reordering the assembly block.
    *   In LLVM IR, the ‘llvm.experimental.asm‘ intrinsic itself
    ↪ carries this semantic. It’s considered an opaque side-effecting
    ↪ operation that cannot be optimized away or reordered with respect
    ↪  to other memory operations, effectively making it "volatile" by
    ↪ default in terms of observable side effects.

### How it Works During Code Generation

When LLVM’s code generator (the backend) processes the LLVM IR:

1.  It encounters the ‘llvm.experimental.asm‘ intrinsic.
2.  It parses the constraint string to understand the register/memory
    ↪ requirements for the operands.
3.  It allocates physical registers or stack slots for the LLVM IR
    ↪ values involved, based on the constraints.
4.  It substitutes the specific register/memory names (e.g., ‘%eax‘,
    ↪ ‘(%rbp)‘) into the assembly string where the placeholders (‘%0‘,
    ↪ ‘%1‘) are.
5.  It injects the resulting, concrete assembly instructions directly
    ↪ into the output ‘.s‘ (assembly) file.
6.  It ensures that the register usage and clobbers are correctly
    ↪ communicated to the register allocator and instruction scheduler.

This mechanism allows LLVM to remain largely unaware of the specific
    ↪ semantics of the inlined assembly while still correctly
    ↪ integrating it into the surrounding generated code, handling data
    ↪  flow, and respecting register usage.
```

USER (Message 9) [gemini-2.5-flash] 2025/06/14 23:11

```
how is llvm ir code structured into maybe modules or some similar kind
    ↪ of terminology, like units or something?
```

AGENT (Message 10) [gemini-2.5-flash] 2025/06/14 23:11

```
In LLVM, the primary unit of code organization is the **Module**.

Think of an LLVM **Module** as analogous to:

*   A single ‘.c‘ or ‘.cpp‘ source file (after preprocessing).
*   A single object file (‘.o‘ or ‘.obj‘).
*   A compilation unit.

A **Module** contains the entire program or a self-contained part of it
   ↪ . It’s the top-level container for all the LLVM IR that belongs
   ↪ together and will eventually be compiled into a single object
   ↪ file or executable component.

Here’s a breakdown of what an LLVM Module typically contains:

1.  **Global Variables:**
    *   Variables declared at the top level (outside any function).
    *   They can be mutable or constant.
    *   Examples: ‘int global_var;‘, ‘const char* string_literal;‘

2.  **Functions:**
    *   The core of executable code.
    *   Each function consists of:
        *   A name (e.g., ‘@main‘, ‘@my_function‘).
        *   A return type.
        *   A list of arguments with their types.
        *   **Basic Blocks:** A function’s body is composed of one or
   ↪ more basic blocks. A basic block is a sequence of instructions
   ↪ where control flow enters at the beginning and leaves at the end
   ↪ without any branches in the middle. The last instruction in a
   ↪ basic block is always a terminator instruction (e.g., ‘ret‘, ‘br
   ↪ ‘, ‘switch‘).
        *   **Instructions:** Within basic blocks, individual LLVM IR
   ↪ instructions perform operations (e.g., ‘add‘, ‘load‘, ‘store‘, ‘
   ↪ call‘).

3.  **Aliases:**
    *   Provide alternative names for global values (functions or
   ↪ global variables).
    *   Useful for providing weaker linkage or different visibility.

4.  **Global Aliases:** (Deprecated in favor of ‘alias‘ and ‘ifunc‘)
    *   Similar to aliases but for functions that can be dynamically
   ↪ resolved at runtime.
```

```
5.  **Named Metadata:**
    *   Collections of metadata nodes. Metadata in LLVM is side-band
   ↪ information that doesn’t affect the execution of the program but
   ↪ provides hints to optimizers, debuggers, or other tools.
    *   Examples: debug information, optimization hints, source
   ↪ location information.

6.  **Data Layout String:**
    *   A string that describes the target machine’s data layout (e.g.,
   ↪  endianness, size and alignment of types, pointer size). This is
   ↪ crucial for correctly interpreting data and generating machine
   ↪ code.

7.  **Target Triple:**
    *   A string that identifies the target architecture, operating
   ↪ system, and environment (e.g., ‘x86_64-pc-linux-gnu‘, ‘armv7-
   ↪ apple-darwin‘). This helps LLVM select the correct backend and
   ↪ ABI rules.

8.  **Source Filename:**
    *   The name of the original source file from which the module was
   ↪ generated. Useful for debugging and error reporting.

### Module Structure in a Nutshell:

‘‘‘
; Target-specific information
target triple = "x86_64-pc-linux-gnu"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
source_filename = "example.c"

; Global variable definition
@my_global_var = dso_local global i32 10, align 4

; String literal (often implicitly a constant global array)
@.str = private unnamed_addr constant [13 x i8] c"Hello, LLVM!\00",
   ↪ align 1

; Function definition
define dso_local i32 @main() {
entry:
  ; Basic Block
  %1 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  %2 = load i32, ptr @my_global_var, align 4
  %call = call i32 (ptr, ...) @printf(ptr noundef getelementptr inbounds
   ↪  ([13 x i8], ptr @.str, i64 0, i64 0))
  ret i32 0
}

; External function declaration (e.g., from a standard library)
declare i32 @printf(ptr noundef, ...)

; Named metadata
!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"clang", i32 17}
!2 = !{!"clang version 17.0.6"}
‘‘‘

### Why Modules?

*   **Compilation Units:** Modules naturally align with how compilers
   ↪ process source files into object files. Each ‘.o‘ file contains
   ↪ the compiled code for one ‘.c‘ file, and an LLVM Module is the IR
   ↪  representation of that ‘.o‘ file before it’s converted to native
   ↪  machine code.
*   **Encapsulation:** They provide a clear boundary for optimization
   ↪ passes. Many passes operate on a "module-level" scope, meaning
   ↪ they analyze and transform the entire module.
*   **Linkage:** Modules contain information about global symbols (
   ↪ functions and variables) and their linkage types (e.g., ‘external
   ↪ ‘, ‘private‘, ‘internal‘, ‘linkonce‘). This information is
   ↪ crucial for the linker to combine multiple modules into an
   ↪ executable.

While you might conceptually think of smaller "units" like functions or
   ↪  basic blocks, the **Module** is the fundamental structural unit
   ↪ that LLVM deals with for compilation, optimization, and code
   ↪ generation. When you use tools like ‘opt‘ or ‘llc‘, you’re
   ↪ typically feeding them an LLVM Module.
```

USER (Message 11) [gemini-2.5-flash] 2025/06/14 23:12

```
does llvm ir have a freestanding triple?
```

AGENT (Message 12) [gemini-2.5-flash] 2025/06/14 23:12

```
Yes, LLVM IR very much has a concept of a "freestanding triple," though
   ↪  it’s more commonly referred to as a **bare-metal target triple**
   ↪  or a triple that doesn’t specify an operating system (OS) or
   ↪ environment.
```

```
### The Target Triple

As we discussed, the **target triple** is a critical piece of
    ↪ information contained within an LLVM Module. It typically follows
    ↪  the format:

‘architecture-vendor-os-environment‘

Examples:

*   ‘x86_64-pc-linux-gnu‘ (x86-64, PC vendor, Linux OS, GNU environment
    ↪ /ABI)
*   ‘armv7-apple-darwin‘ (ARM v7, Apple vendor, Darwin OS - macOS/iOS,
    ↪ default environment)
*   ‘wasm32-unknown-wasi‘ (WebAssembly 32-bit, unknown vendor, WASI OS
    ↪ - WebAssembly System Interface)

### Freestanding Triples

A "freestanding" or bare-metal triple is one where the ‘os‘ and ‘
    ↪ environment‘ components are set to ‘none‘ or ‘unknown‘. This
    ↪ indicates that there’s no standard operating system or C runtime
    ↪ environment that the compiled code can assume.

Common examples of freestanding triples:

*   ‘armv7m-none-eabi‘ (ARM Cortex-M7, no vendor, no OS, Embedded ABI)
*   ‘arm-none-eabi‘ (Generic ARM, no vendor, no OS, Embedded ABI)
*   ‘aarch64-none-elf‘ (AArch64, no vendor, no OS, ELF object format)
*   ‘riscv32-unknown-elf‘ (RISC-V 32-bit, unknown vendor, no OS, ELF
    ↪ object format)

### Implications of a Freestanding Triple

When LLVM encounters a freestanding triple, it makes several important
    ↪ assumptions and adjustments:

1.  **No C Runtime (CRT) Initialization:** The generated code will *not
    ↪ * include typical C runtime setup (like calling ‘_start‘ or
    ↪ initializing global constructors). You are responsible for
    ↪ providing the entry point and all necessary low-level
    ↪ initialization.
2.  **No OS System Calls:** You cannot directly call POSIX or Windows
    ↪ API functions (e.g., ‘printf‘, ‘malloc‘, ‘open‘). If you need
    ↪ such functionality, you must provide your own bare-metal
    ↪ implementations or link against a bare-metal compatible library (
    ↪ like Newlib or your own custom routines).
3.  **No Dynamic Linking:** Freestanding executables are typically
    ↪ statically linked. There's no expectation of shared libraries or
    ↪ dynamic loaders.
4.  **No Standard Library Assumption:** The compiler won't implicitly
    ↪ link against ‘libc‘ or other standard libraries. All external
    ↪ symbols must either be provided by your code or a bare-metal
    ↪ library you explicitly link.
5.  **ABI (Application Binary Interface):** Even without an OS, an ABI
    ↪ is crucial. Freestanding targets often specify an "Embedded ABI"
    ↪ (EABI), which defines calling conventions, register usage, and
    ↪ data layout. This is essential for your bare-metal code to
    ↪ correctly interact with any compiled C/C++ functions.
6.  **Linker Scripts:** You'll almost certainly need a custom linker
    ↪ script to precisely control memory layout (where code, data,
    ↪ stack, and heap go in the target's memory map) since there's no
    ↪ OS to manage this for you.
7.  **No Paging/Virtual Memory:** The concept of virtual memory, paging
    ↪ , and memory protection is typically absent. Code runs directly
    ↪ on physical addresses.

### Use Cases for Freestanding Triples

*   **Embedded Systems/Microcontrollers:** This is the most common use
    ↪ case. When programming a small chip that runs without an OS (like
    ↪  an Arduino or an STM32), you use a bare-metal toolchain.
*   **Operating System Kernels:** Developing an OS kernel itself
    ↪ requires compiling code in a freestanding environment before the
    ↪ OS is fully up and running.
*   **Bootloaders:** Similar to kernels, bootloaders are the first
    ↪ pieces of code to run on a system and operate in a bare-metal
    ↪ context.
*   **Firmware:** Code for specialized hardware that doesn't have a
    ↪ general-purpose OS.
*   **Hypervisors:** Low-level software that runs directly on hardware
    ↪ to manage virtual machines.

In essence, when you select a freestanding triple for LLVM, you're
    ↪ telling the compiler: "Generate the absolute minimum machine code
    ↪  required for this architecture, without assuming any surrounding
    ↪  OS or runtime environment. I will handle everything else."
```

USER (Message 13) [gemini-2.5-flash] 2025/06/14 23:13

```
why is it called a triple when there are four segments?
```

AGENT (Message 14) [gemini-2.5-flash] 2025/06/14 23:13

```
That’s an excellent and very common question! You’re absolutely right
   ↪ to notice that ‘architecture-vendor-os-environment‘ has four
   ↪ parts, not three.

The reason it’s called a **triple** is historical and somewhat reflects
   ↪  its common, simpler form and evolution:

1.  **Historical Context:**
    *   The concept originated with GNU Autoconf/Autotools, where the
   ↪ canonical form for specifying a target system for cross-
   ↪ compilation was initially simpler, often just ‘cpu-company-system
   ↪ ‘.
    *   For many early or simpler targets, the "environment" part wasn’
   ↪ t always explicitly needed or was implicitly derived from the OS.
   ↪  So, ‘architecture-vendor-os‘ was a very common, *three-part*
   ↪ representation.

2.  **The "Environment" Segment is Often Optional or Implicit:**
    *   For many common targets, the environment part is either omitted
   ↪ , or it’s implicitly derived from the OS.
        *   ‘x86_64-pc-linux-gnu‘: Here, ‘gnu‘ explicitly refers to the
   ↪  GNU C library ABI. It could arguably be omitted if ‘linux‘
   ↪ always implies ‘gnu‘ for a given architecture, but it’s explicit
   ↪ for clarity and to differentiate from, say, musl libc.
        *   ‘x86_64-apple-darwin‘: The environment is usually omitted
   ↪ because ‘darwin‘ implies the standard Apple ABI.
        *   ‘wasm32-unknown-wasi‘: Here ‘wasi‘ acts as an OS *and*
   ↪ environment.
    *   For bare-metal targets, ‘none‘ or ‘elf‘ might appear in the
   ↪ environment slot, or it might just be ‘armv7m-none-eabi‘, where ‘
   ↪ eabi‘ is technically an environment/ABI but the "OS" slot is ‘
   ↪ none‘.

3.  **Core Indentifiers:** The most fundamental, almost always present,
   ↪  and distinct identifiers are:
    *   **Architecture:** What CPU instruction set are we targeting? (e
   ↪ .g., ‘x86_64‘, ‘arm‘, ‘riscv32‘)
    *   **Vendor:** Who made the system or whose ABI are we following?
   ↪ (e.g., ‘pc‘, ‘apple‘, ‘unknown‘, ‘none‘)
    *   **OS/System:** What operating system or execution environment
   ↪ are we on? (e.g., ‘linux‘, ‘darwin‘, ‘windows‘, ‘none‘, ‘cuda‘)

    The "environment" or "ABI" segment often feels like a refinement or
   ↪  sub-category of the "OS" or even an extension to the basic "
   ↪ architecture-vendor-os" description.

So, while ‘architecture-vendor-os-environment‘ is the full, modern, and
```

```
    ↪  most precise form in LLVM, the term "triple" persists due to its
    ↪  historical roots and the fact that for many common cases, the
    ↪ environment part is less distinct or even omitted, leaving three
    ↪ truly distinct, core components. It’s a linguistic legacy more
    ↪ than a strict numerical definition in all cases.
```

USER (Message 15) [gemini-2.5-flash] 2025/06/14 23:14

```
is each llvm ir module a single file? also, how does the ir reference
    ↪ some external reference?
```

AGENT (Message 16) [gemini-2.5-flash] 2025/06/14 23:14

```
### Is Each LLVM IR Module a Single File?

Conceptually and typically, **yes, each LLVM IR module is represented
    ↪ as a single file** when it’s serialized to disk.

*   When Clang (or any other frontend) compiles a single source file (e
    ↪ .g., ‘my_program.c‘), it generates a single LLVM IR module, which
    ↪  it can then write out to a ‘.ll‘ file (human-readable text) or a
    ↪  ‘.bc‘ file (binary bitcode).
*   Similarly, when you compile a library or an object file, that
    ↪ compilation unit corresponds to one LLVM IR module.

You can combine multiple LLVM IR modules into a single larger module
    ↪ using tools like ‘llvm-link‘, but the starting point for
    ↪ compilation units is usually one source file generating one
    ↪ module.

### How Does LLVM IR Reference External References?

LLVM IR references external entities (like functions defined in other
    ↪ modules/libraries, or global variables defined elsewhere) using
    ↪ **declarations** and by specifying their **linkage type**.

When a function or global variable is defined within an LLVM IR module,
    ↪  it has a definition (a body for functions, an initial value for
    ↪ globals). When it’s merely referenced but defined elsewhere, it
    ↪ has only a **declaration**.

Let’s look at examples:

#### 1. Referencing External Functions

If your code calls a function like ‘printf‘ (which is typically defined
    ↪  in the C standard library, external to your module), its LLVM IR
    ↪  representation will be a **declaration** for ‘printf‘.

**C Code:**

‘‘‘c
#include <stdio.h>

void my_func() {
    printf("Hello from my_func!\n");
}
‘‘‘

**Corresponding LLVM IR (for ‘my_func‘):**

‘‘‘llvm
; This is a declaration for the printf function.
; It specifies its return type (i32), arguments (ptr, ... for varargs),
; and attributes (noundef here).
; It does NOT have a function body (the ’{ ... }’ part).
declare i32 @printf(ptr noundef, ...)

; This is the definition of my_func, which calls the declared printf.
define dso_local void @my_func() {
entry:
  %call = call i32 (ptr, ...) @printf(ptr noundef getelementptr inbounds
    ↪  ([21 x i8], ptr @.str, i64 0, i64 0))
  ret void
}

; The string literal for "Hello from my_func!\n" would be a global
    ↪ constant
@.str = private unnamed_addr constant [21 x i8] c"Hello from my_func
    ↪ !\00"
‘‘‘

**Key Points:**

*   The ‘declare‘ keyword signifies that the function’s definition (its
    ↪  actual code) resides elsewhere.
*   The ‘@‘ prefix indicates a global value (function or global variable
    ↪ ).
*   The function signature (return type, argument types, attributes)
    ↪ must match the actual definition in the external library for
    ↪ correct linking and calling conventions.
*   The **linker** is ultimately responsible for resolving these
    ↪ declared symbols to their actual definitions in other object
    ↪ files or shared libraries.
```

```
#### 2. Referencing External Global Variables

Similarly, if your code references a global variable defined in another compilation unit:

**C Code (in ‘file1.c‘):**

‘‘‘c
int external_global_var; // Defined in file2.c
extern int external_global_var; // Declaration in file1.c

void use_external_var() {
    external_global_var = 123;
}
‘‘‘

**Corresponding LLVM IR (for ‘use_external_var‘ in ‘file1.ll‘):**

‘‘‘llvm
; Declaration for the external global variable.
; It has a type (i32) but no initializer and no definition of where its memory is.
; The ’external global’ linkage indicates it’s defined elsewhere.
@external_global_var = external global i32

define dso_local void @use_external_var() {
entry:
  store i32 123, ptr @external_global_var, align 4
  ret void
}
‘‘‘

**Key Points:**

*   The ‘external global‘ linkage type (which is the default for declarations if not specified) explicitly tells LLVM and the linker that this symbol is defined outside the current module.
*   There’s no initializer here; the memory for this variable is allocated and initialized by the module where it’s defined.

### The Role of the Linker

LLVM itself (specifically, the ‘llc‘ tool that generates object files from IR) doesn’t resolve external references. It generates an object file (e.g., ‘my_program.o‘) where all these ‘declare‘d symbols remain **unresolved symbols**.
```

```
It’s the **system linker** (like ‘ld‘ on Linux/macOS or ‘link.exe‘ on
    ↪ Windows) that takes all these object files (‘.o‘, ‘.obj‘) and
    ↪ library archives (‘.a‘, ‘.lib‘) and shared libraries (‘.so‘, ‘.
    ↪ dylib‘, ‘.dll‘) and performs the final symbol resolution. The
    ↪ linker matches the declarations in one object file to the
    ↪ definitions in another, stitches everything together, and
    ↪ resolves all the addresses and offsets to create the final
    ↪ executable or library.
```

USER (Message 17) [gemini-2.5-flash] 2025/06/14 23:16

```
is there anything like an llvm ir header file?
```

AGENT (Message 18) [gemini-2.5-flash] 2025/06/14 23:16

```
That’s a great question, and it points to a fundamental difference in
    ↪ how LLVM IR works compared to C/C++ source code.

**No, there isn’t a direct equivalent to a C/C++ header file (‘.h‘) for
    ↪  LLVM IR.**

Here’s why and what serves a similar purpose in the LLVM ecosystem:

### Why No IR Header File?

1.  **Declarations are Self-Contained:** In LLVM IR, the ‘declare‘
    ↪ keyword is all you need to define an external function or global
    ↪ variable. It contains the full signature (return type, argument
    ↪ types, attributes, linkage). There’s no separate "type definition
    ↪ " mechanism that needs to be imported or included.

    *   In C, a function prototype in a header (‘void foo(int);‘)
    ↪ defines the signature, but the actual function body is in a ‘.c‘
    ↪ file.
    *   In LLVM IR, the ‘declare void @foo(i32)‘ line itself serves both
    ↪  as the prototype *and* the mechanism to make the symbol known
    ↪ for linking.

2.  **Modules are the Compilation Unit:** As discussed, the Module is
    ↪ the unit. When an LLVM frontend compiles a ‘.c‘ file, it produces
    ↪  a complete Module with all necessary declarations and
    ↪ definitions. If your ‘.c‘ file includes a ‘.h‘ file, the
    ↪ information from that header is *inlined* into the compilation
    ↪ unit’s LLVM IR module. There’s no separate ‘.ll‘ or ‘.bc‘ file
    ↪ generated *just* for the header.
```

```
3.  **Bitcode Linker (‘llvm-link‘):** When you’re working directly with
  ↪  LLVM IR (e.g., if you have multiple ‘.ll‘ or ‘.bc‘ files that
  ↪ need to be combined before native code generation), you use the ‘
  ↪ llvm-link‘ tool. ‘llvm-link‘ takes multiple LLVM IR modules and
  ↪ merges them into a single, larger module. It handles duplicate
  ↪ declarations gracefully. This merging process is conceptually
  ↪ similar to what the C preprocessor does with ‘#include‘
  ↪ directives, but it operates on already-parsed IR.

### What Serves a Similar Purpose?

While there’s no dedicated header file, these mechanisms fulfill
  ↪ similar roles:

1.  **‘declare‘ Instructions:** This is the most direct equivalent.
  ↪ When you define an external function in LLVM IR (as seen
  ↪ previously: ‘declare i32 @printf(ptr noundef, ...)‘), this
  ↪ declaration acts exactly like a C function prototype in a header
  ↪ file  it tells the current module how to call a function whose
  ↪ definition will be provided by another module at link time.

2.  **‘llvm-link‘ for Combining Modules:**
    If you’re building a larger project by combining independently
  ↪ compiled LLVM IR files, ‘llvm-link‘ is your "inclusion" mechanism
  ↪ .
    ‘‘‘bash
    # Compile file1.c and file2.c to LLVM bitcode
    clang -emit-llvm -c file1.c -o file1.bc
    clang -emit-llvm -c file2.c -o file2.bc

    # Link the bitcode files into a single module.bc
    llvm-link file1.bc file2.bc -o combined.bc

    # Now you can optimize and compile combined.bc to a native object
  ↪ file
    opt -O3 combined.bc -o optimized.bc
    llc optimized.bc -o program.o
    ‘‘‘
    This ‘llvm-link‘ step effectively brings all the declarations and
  ↪ definitions from ‘file1.bc‘ and ‘file2.bc‘ into a single
  ↪ compilation unit, resolving internal references between them (
  ↪ though not yet to external system libraries).

3.  **C/C++ Header Files (at the source level):** From the perspective
  ↪ of a C/C++ developer using Clang, ‘*.h‘ files still very much
  ↪ exist and are used exactly as you’d expect. It’s just that the *
  ↪ intermediate representation* doesn’t have a separate concept for
```

```
   ↪ them. The information from headers is absorbed into the ‘.ll‘ or
   ↪ ‘.bc‘ file that Clang outputs for the corresponding ‘.c‘ or ‘.cpp
   ↪ ‘ file.

In summary, LLVM IR’s design of self-contained declarations and module-
   ↪ as-compilation-unit eliminates the need for separate "IR header
   ↪ files." The necessary information for linking is either directly
   ↪ in the ‘declare‘ instruction or is implicitly handled when ‘llvm-
   ↪ link‘ merges multiple modules.
```

USER (Message 19) [gemini-2.5-flash] 2025/06/14 23:17

```
i take it you cant import anything then in an llvm ir file?
```

AGENT (Message 20) [gemini-2.5-flash] 2025/06/14 23:17

```
You **cannot use an explicit ‘import‘ directive** *within* an LLVM IR
   ↪ (‘.ll‘ or ‘.bc‘) file itself, in the way you might in Python,
   ↪ Java, or JavaScript, or even ‘#include‘ in C/C++.

LLVM IR files are designed as self-contained compilation units (Modules
   ↪ ), not as scripts or components that directly reference other
   ↪ files at the text/bitcode level for inclusion.

Here’s a deeper look at why and how the equivalent functionality is
   ↪ achieved:

### Why No ‘import‘ Directive in LLVM IR Files?

1.  **Nature of LLVM IR:** LLVM IR is a *representation* of code, not a
   ↪  scripting or programming language meant for direct linking or
   ↪ modularization at its own level. It’s the output of a frontend
   ↪ and the input to a backend.
2.  **Compilation Units:** Each ‘.ll‘ or ‘.bc‘ file typically
   ↪ represents a complete LLVM **Module**. A module is already
   ↪ intended to be a single, self-contained compilation unit.
3.  **Link-Time Resolution:** The primary mechanism for combining
   ↪ different pieces of code (different modules) is **linking**, not
   ↪ importing. The frontend (e.g., Clang) generates an LLVM Module,
   ↪ making sure all external dependencies are represented by ‘declare
   ↪ ‘ instructions. When you compile these modules to native object
   ↪ files (‘.o‘), the job of finding the actual definitions for those
   ↪  ‘declare‘d symbols falls to the system linker.

### How is the "Import" Functionality Achieved?

The functionality that you associate with "importing" or "including"
```

```
   ↪ other code is handled at different stages in the LLVM toolchain:

1.  **Frontend (e.g., Clang) handles includes:**
    *   If you’re compiling C/C++ code, the ‘#include‘ directives are
   ↪ processed by the C preprocessor and compiler *before* LLVM IR is
   ↪ generated. The contents of included header files are effectively
   ↪ "inlined" into the source file being compiled. The LLVM IR module
   ↪  generated for that source file will then contain all the
   ↪ necessary declarations and definitions that resulted from that
   ↪ inclusion.

2.  **‘llvm-link‘ merges LLVM Modules:**
    *   If you have multiple ‘.ll‘ or ‘.bc‘ files that were generated
   ↪ independently (e.g., ‘a.ll‘ from ‘a.c‘ and ‘b.ll‘ from ‘b.c‘),
   ↪ and ‘a.ll‘ calls a function defined in ‘b.ll‘, ‘a.ll‘ will
   ↪ contain a ‘declare‘ for that function.
    *   To combine them into a single larger LLVM IR module (which is
   ↪ often done before optimization), you use the ‘llvm-link‘ tool:
        ‘‘‘bash
        llvm-link a.ll b.ll -o combined.ll
        ‘‘‘
    *   This process effectively merges all the contents of ‘a.ll‘ and
   ↪ ‘b.ll‘ into ‘combined.ll‘. ‘llvm-link‘ correctly handles ‘declare
   ↪ ‘s that now have corresponding definitions within the combined
   ↪ module. This is the closest equivalent to "importing" or "
   ↪ including" other LLVM IR.

3.  **System Linker resolves external symbols:**
    *   Once you’ve compiled your LLVM IR module(s) down to native
   ↪ object files (‘.o‘), the operating system’s linker takes over.
   ↪ The linker’s job is to take all the ‘.o‘ files and any specified
   ↪ libraries (‘.a‘, ‘.so‘, ‘.dylib‘, ‘.dll‘) and resolve all the
   ↪ remaining unresolved symbols (the ‘declare‘s that didn’t find a
   ↪ definition within your own ‘llvm-link‘ed modules). This is how
   ↪ your code links against standard libraries like ‘libc‘ or other
   ↪ user-defined libraries.

So, while you won’t find an ‘import‘ statement in an ‘.ll‘ file, the
   ↪ overall toolchain provides the necessary mechanisms to bring
   ↪ together code from multiple source files or pre-compiled units to
   ↪  form a complete program.
```

USER (Message 21) [gemini-2.5-flash] 2025/06/14 23:30

```
what options are available in an llvm ir file when mentioning that we
   ↪ are depending on some externally defined function or variable. is
   ↪  there much configuratoin options or is it pretty blatant?
```

AGENT (Message 22) [gemini-2.5-flash] 2025/06/14 23:30

~~~
When mentioning an externally defined function or variable in an LLVM IR file, the options for configuration are relatively **blatant** and focused primarily on ensuring correct linkage and ABI compatibility. There isn't a vast array of "configuration options" in the way you might configure a library import in a higher-level language or build system.

The core mechanism is the **declaration** of the global value (function or variable), and the key "options" are its **type**, **name**, and **linkage type**, along with certain **attributes** for functions.

### 1. External Function Declaration

For an externally defined function, you use the `declare` keyword.

```llvm
declare [return type] @[function name]([argument types]...) [function attributes]
```

**Key components and their "options":**

*   **`declare`**: Keyword indicating it's a declaration, not a definition.
*   **`[return type]`**: The type of the value returned by the function (e.g., `i32`, `void`, `ptr`). This **must** match the external definition's return type.
*   **`@[function name]`**: The mangled or unmangled name of the function. This **must** match the name of the function as it will be seen by the linker (e.g., `_Z3fooii` for a C++ mangled function, or `printf` for a C function).
*   **`([argument types]...)`**: A comma-separated list of the types of arguments the function expects. For variadic functions (like `printf`), you use `...` after the last fixed argument. This **must** match the external definition's argument types and order.
*   **`[function attributes]`**: These provide hints to the optimizer and code generator about the function's behavior. While they don't change the *declaration* itself, they are critical for correct and optimal code generation when calling the external function. These are listed after the argument list and before the closing parenthesis.

    Common attributes for external functions include:
    *   **`nounwind`**: Function does not unwind the stack (i.e., doesn
~~~

~~~~
    ↪ ’t throw exceptions).
     *   **‘noreturn‘**: Function does not return to its caller (e.g., ‘
    ↪ exit‘, ‘abort‘).
     *   **‘readonly‘**: Function only reads memory pointed to by its
    ↪ arguments, doesn’t write to any global memory (unless through its
    ↪  own arguments, if those arguments point to memory it can read).
     *   **‘readnone‘**: Function neither reads nor writes memory.
     *   **‘noalias‘**: The return value pointer does not alias any
    ↪ arguments or other accessible memory.
     *   **‘noundef‘**: Arguments passed to this parameter are
    ↪ guaranteed to not be undefined.
     *   **‘alignstack(N)‘**: Specifies required stack alignment on
    ↪ entry.
     *   **‘allocsize(N)‘**: Hints about memory allocation.
     *   **‘cold‘**: Hints that this function is unlikely to be executed
    ↪ .
     *   **‘hot‘**: Hints that this function is likely to be executed
    ↪ frequently.
     *   **‘optnone‘**: Disable optimization on this function (usually
    ↪ for debugging).

     These attributes are crucial for **ABI compatibility** and **
    ↪ optimization effectiveness**. For instance, if you declare ‘
    ↪ printf‘ as ‘noreturn‘ but it actually returns, you’ll have issues
    ↪ . If you mark a memory-modifying function as ‘readonly‘, the
    ↪ optimizer might incorrectly reorder calls or eliminate loads.

#### Example: External ‘strlen‘

‘‘‘llvm
declare i64 @strlen(ptr noundef) #0 ; Assuming C standard library strlen
                                     ; #0 refers to an attribute group
    ↪ defined elsewhere
                                     ; e.g., attributes #0 = { nounwind
    ↪ readonly }
‘‘‘

### 2. External Global Variable Declaration

For an externally defined global variable, you use ‘@‘ followed by its
    ↪ name and type.

‘‘‘llvm
@[variable name] = [linkage type] global [type]
‘‘‘

**Key components and their "options":**
~~~~

```
*   **‘@[variable name]‘**: The name of the global variable. This **must
  ↪ ** match the name used by the linker.
*   **‘[linkage type]‘**: This is the most significant "option" for
  ↪ external globals. It dictates how the linker should handle the
  ↪ symbol.
    *   **‘external‘**: (Default if omitted for a declaration) The most
  ↪  common type. Means the variable is defined in another module and
  ↪  will be resolved by the linker.
    *   **‘extern_weak‘**: The symbol may or may not be present at link
  ↪  time. If it’s not found, its address becomes null. Useful for
  ↪ optional weak dependencies.
    *   Other linkage types (‘private‘, ‘internal‘, ‘common‘, ‘weak‘, ‘
  ↪ available_externally‘, ‘linkonce‘, ‘appending‘) are typically
  ↪ used for definitions, not external declarations, or for very
  ↪ specific C/C++ language features (e.g., ‘common‘ for C tentative
  ↪ definitions).
*   **‘global‘**: Keyword indicating it’s a global variable.
*   **‘[type]‘**: The type of the global variable (e.g., ‘i32‘, ‘[10 x
  ↪ i8]‘, ‘ptr‘). This **must** match the external definition’s type
  ↪ and size.

#### Example: External ‘errno‘ (conceptually)

‘‘‘llvm
; Assuming errno is an external global integer
@errno = external global i32
‘‘‘

### Summary of "Configuration" for External References:

It’s "blatant" in that there are no complex nested configurations. You
  ↪ essentially just provide:

1.  **The Name:** How the linker knows to find it.
2.  **The Type/Signature:** What data it holds or how it’s called (
  ↪ number/types of arguments, return type). This is critical for ABI
  ↪  compatibility.
3.  **Linkage Type:** (Mainly for variables) How the linker should
  ↪ treat the symbol’s resolution (e.g., must be present, optional).
4.  **Attributes:** (Mainly for functions) Non-functional properties
  ↪ that aid optimization and correct code generation (e.g., does it
  ↪ write to memory? does it return?).

These elements, while seemingly simple, are precisely what the compiler
  ↪  and linker need to correctly generate calls to and interact with
  ↪  external code. Any mismatch in these aspects between the
```

```
   ↪ declaration and the actual definition will lead to linking errors
   ↪  or subtle runtime bugs.
```

USER (Message 23) [gemini-2.5-flash] 2025/06/14 23:31

```
could you casually use "..." for any function if you are too lazy to
   ↪ know what exact input it can take?
```

AGENT (Message 24) [gemini-2.5-flash] 2025/06/14 23:31

```
You’re asking if you can just casually use ‘...‘ (varargs, or variable
   ↪ arguments) in an LLVM IR function declaration, even if the actual
   ↪  function isn’t variadic, just because you’re "too lazy" to
   ↪ specify the exact arguments.

The answer is **no, you absolutely cannot** do that without running
   ↪ into serious issues.

### Why You Can’t Casually Use ‘...‘ in LLVM IR Declarations:

1.  **ABI Mismatch:**
    *   The most critical reason. The "Application Binary Interface" (
   ↪ ABI) dictates how functions pass arguments (which registers,
   ↪ which stack slots) and return values.
    *   A function declared with ‘...‘ (variadic) follows a *different*
   ↪  calling convention than a function with fixed arguments. For
   ↪ example, on x86-64 System V ABI:
        *   Fixed arguments: Integers go in ‘rdi‘, ‘rsi‘, ‘rdx‘, ‘rcx‘,
   ↪  ‘r8‘, ‘r9‘. Floating-point arguments go in ‘xmm0‘ to ‘xmm7‘.
        *   Variadic arguments: *All* fixed arguments (integer and
   ↪ floating-point) are promoted and passed in both integer registers
   ↪  and XMM registers. There’s also usually a count of floating-
   ↪ point registers passed.
    *   If you declare a fixed-argument function as variadic, your
   ↪ caller will prepare arguments according to the variadic
   ↪ convention, but the callee (the actual function) will expect them
   ↪  according to the fixed-argument convention. This will lead to:
        *   **Incorrect values:** The callee will read garbage from
   ↪ registers or stack slots where it expects arguments.
        *   **Crashes:** The callee might try to use values that aren’t
   ↪  there, leading to segmentation faults or other crashes.
        *   **Stack corruption:** Incorrect stack cleanup (caller vs.
   ↪ callee responsibility) can corrupt the stack frame.

2.  **Optimizer Confusion:**
    *   LLVM’s optimizer relies heavily on knowing the exact types and
   ↪ number of arguments a function takes.
```

```
    *   If you use `...`, the optimizer loses information about what
   ↪ values are truly being passed or what the function needs. This
   ↪ can prevent many optimizations, as the function becomes an "
   ↪ opaque" black box.

3.  **Type Safety (even in IR):**
    *   While LLVM IR is lower-level than C, it still enforces strong
   ↪ typing. `...` means "any number of arguments of any type," but it
   ↪  doesn't mean "I don't know the types of the fixed arguments."
   ↪ The types *before* the `...` must still be accurate. If a
   ↪ function is `void foo(int a, float b)`, you could declare it as `
   ↪ declare void @foo(i32, float, ...)`, but *not* `declare void @foo
   ↪ (...)`.
    *   Even if you correctly declare the fixed arguments and *then*
   ↪ use `...`, the ABI issue mentioned above still applies if the
   ↪ actual function is not variadic.

### When `...` IS Appropriate:

The `...` (variadic) syntax in LLVM IR is **only appropriate** when:

*   The function is genuinely variadic (e.g., `printf`, `scanf`, `execl
   ↪ `).
*   The function's actual definition follows a variadic calling
   ↪ convention for your target ABI.

### Conclusion

Using `...` as a shortcut for "I'm lazy" will lead to incorrect code
   ↪ generation, runtime errors, and potentially very difficult-to-
   ↪ debug crashes. The exact signature of external functions (return
   ↪ type, argument types, and variadic status) is absolutely crucial
   ↪ and must match the actual function definition. The compiler and
   ↪ linker rely on this precise information to ensure your program
   ↪ calls external code correctly.
```

USER (Message 25) [gemini-2.5-flash] 2025/06/14 23:33

```
is there any way in llvm ir of not specifying the arguments?
```

AGENT (Message 26) [gemini-2.5-flash] 2025/06/14 23:33

```
No, **there is no way in LLVM IR to declare a function without
   ↪ specifying its arguments and return type**. The arguments (and
   ↪ return type) are a fundamental part of a function's type
   ↪ signature in LLVM IR, and they are always explicitly defined.
```

```
Think of it this way:

*   **Type Safety:** LLVM IR is a strongly typed intermediate
  ↪ representation. Every value has a type, and every operation (like
  ↪  a function call) operates on values of specific types. Not
  ↪ knowing the argument types would break this fundamental principle
  ↪ .
*   **ABI Compatibility:** As discussed, the Application Binary
  ↪ Interface (ABI) depends heavily on knowing the exact types and
  ↪ order of arguments to correctly manage registers and stack frames
  ↪  during function calls. Without this information, it’s impossible
  ↪  to generate correct machine code for a function call.
*   **Optimization:** Optimizers need to know the types to perform many
  ↪  transformations. For example, if it knows a function takes an ‘
  ↪ i32*‘ and returns ‘void‘, it can make assumptions about memory
  ↪ access. If it doesn’t know the types, its optimization
  ↪ capabilities are severely limited.
*   **Variadic Functions:** Even for variadic functions (which you were
  ↪  asking about previously), you *must* specify the fixed arguments
  ↪  that come before the ellipsis. The ‘...‘ itself signifies that
  ↪ there *might* be more arguments, but it doesn’t mean "I don’t
  ↪ know any of the arguments."

    *   ‘declare i32 @printf(ptr noundef, ...)‘: Here, ‘ptr noundef‘ is
  ↪ the first (fixed) argument type. The ‘...‘ means any subsequent
  ↪ arguments.
    *   ‘declare void @exit(i32)‘: Here, ‘i32‘ is the only fixed
  ↪ argument. There’s no ‘...‘
    *   ‘declare void @my_no_arg_func()‘: Even a function with no
  ↪ arguments needs empty parentheses.

### What if I Truly Don’t Know the Signature?

If you are in a situation where you genuinely do not know the signature
  ↪  of an external function you need to call (perhaps you’re dealing
  ↪  with an extremely dynamic, unconventional runtime, or you’re
  ↪ doing something very low-level and risky like calling code at an
  ↪ arbitrary memory address), then you are operating outside the
  ↪ normal, type-safe boundaries of LLVM IR.

In such a highly unusual scenario, you would typically:

1.  **Use a Generic Function Pointer and Bitcast:**
    *   Declare a very generic function pointer type (e.g., ‘ptr‘, or a
  ↪  function type that takes no arguments and returns nothing).
    *   ‘bitcast‘ the memory address you intend to call (e.g., from a
  ↪ global variable holding an address) to this generic function
```

~~~
   ↪ pointer type.
    *   Then, call through this generic function pointer.

    **Example (Highly dangerous and not recommended for typical use):**

    ```llvm
    ; Declare a very generic function pointer type
    ; void()* - a function that takes no args and returns void
    ; This is arbitrary, you might pick a different generic type
    %generic_func_ptr_type = type void ()*

    ; Assume @external_address is a global variable holding the address
    ; of the function you want to call, populated at runtime or by
   ↪ linker.
    ; This is simplified; you'd typically load this from somewhere.
    @external_address_holder = global ptr null

    define void @call_unknown_func() {
      entry:
        ; Load the address from the global
        %func_addr = load ptr, ptr @external_address_holder, align 8

        ; Bitcast the raw address to your chosen generic function
   ↪ pointer type
        %callable_ptr = bitcast ptr %func_addr to %
   ↪ generic_func_ptr_type

        ; Call the function through the generic pointer.
        ; You are responsible for ensuring the stack, registers,
        ; and arguments are correct for the target architecture's ABI.
        ; This call will likely crash if the actual function needs
   ↪ arguments
        ; or returns a value, as the ABI will be mismatched.
        call void %callable_ptr()

        ret void
    }
    ~~~
    This approach means *you* (the code generating the IR) are
   ↪ completely responsible for setting up the correct registers and
   ↪ stack for the target's ABI *before* the `call` instruction, and
   ↪ interpreting the return value. LLVM's generated `call`
   ↪ instruction for `void %callable_ptr()` will only set up for a
   ↪ call to a `void()` function.

**In summary: No, LLVM IR requires explicit declaration of argument
   ↪ types and return types for functions. Any attempt to bypass this
```
```

```
   ↪ with "I don’t know" will lead to undefined behavior, ABI
   ↪ mismatches, and likely crashes at runtime.** The "laziness" you
   ↪ describe is precisely what compilers and ABIs exist to prevent.
```

USER (Message 27) [gemini-2.5-flash] 2025/06/14 23:43

```
basically a raw function at this piont then is something that you are
   ↪ essentially just jumping to. it ends up just being a bare bones
   ↪ memory address?
```

AGENT (Message 28) [gemini-2.5-flash] 2025/06/14 23:43

```
Yes, you’ve precisely captured the essence of what a "raw function" or
   ↪ "bare bones memory address" becomes when you try to circumvent
   ↪ the standard function calling mechanisms and ABI.

### A "Raw Function" as a Bare Bones Memory Address

When you declare a function in LLVM IR with a specific signature (e.g.,
   ↪  ‘declare i32 @my_func(i32, i32)‘), you are asking the compiler to
   ↪  generate machine code that adheres to the **Application Binary
   ↪ Interface (ABI)** for your target system. The ABI defines:

*   How arguments are passed (e.g., in ‘RDI‘, ‘RSI‘ for first two
   ↪ integers on x86-64 Linux).
*   How return values are passed (e.g., in ‘RAX‘ for integers).
*   Which registers are "caller-saved" vs. "callee-saved."
*   Stack alignment requirements.
*   Stack frame layout.

The ‘call‘ instruction in LLVM IR then compiles down to a native
   ↪ machine ‘call‘ instruction, and the compiler handles all the ABI-
   ↪ mandated setup and teardown based on the declared signature.

However, when you resort to something like:

1.  **Loading a raw memory address into a register.**
2.  **Bitcasting it to a generic function pointer type.**
3.  **Calling through that pointer.**

You are effectively telling LLVM: "I have a memory address, treat it as
   ↪  a function entry point. Don’t worry about the ABI, I’ll handle
   ↪ that."

At this point, the "function" you’re calling is, from the perspective
   ↪ of LLVM’s code generation, just a **bare memory address** that
   ↪ the program jumps to.
```

```
### Implications of Treating a Function as a Raw Address:

*   **No ABI Compliance by LLVM:** LLVM will generate a machine `call`
   ↪ instruction to that address. However, it will *not* automatically
   ↪  set up arguments in registers or on the stack according to any
   ↪ specific ABI based on a specific function signature, because you
   ↪ haven't provided one. It only sets up for the *generic* signature
   ↪  you gave it during the `bitcast`.
*   **Your Sole Responsibility for ABI:** You are entirely responsible
   ↪ for:
    *   Placing the arguments in the correct registers and/or stack
   ↪ locations *before* the `call` instruction is executed.
    *   Saving and restoring any registers that the "raw function"
   ↪ might clobber, according to the target's ABI.
    *   Reading the return value from the correct register or memory
   ↪ location *after* the `call`.
    *   Ensuring stack alignment and cleanup.
*   **No Optimization Insights:** Because LLVM doesn't know the
   ↪ function's signature or behavior, it cannot perform any
   ↪ meaningful optimizations related to its inputs, outputs, or side
   ↪ effects. It treats it as an opaque jump.
*   **Risk of Crashes:** Any mismatch between how your code sets up the
   ↪  call and what the "raw function" expects will lead to undefined
   ↪ behavior, which often manifests as crashes (segmentation faults,
   ↪ invalid instructions, data corruption).

### Use Cases for this "Raw Address" Concept:

This kind of "raw function as a bare memory address" interaction is
   ↪ typically reserved for very low-level scenarios:

*   **JIT (Just-In-Time) Compilation:** When a JIT compiler generates
   ↪ machine code in memory and then executes it.
*   **Self-Modifying Code:** Though rare and generally discouraged, if
   ↪ code dynamically modifies itself or generates new functions at
   ↪ runtime.
*   **Calling into a specific known firmware routine/trap:** Where the
   ↪ exact entry point and ABI are documented outside the standard
   ↪ compiler toolchain.
*   **Exploits/Security Research:** Bypassing normal program flow by
   ↪ jumping to arbitrary code.

For any standard application programming, sticking to declared
   ↪ functions with proper signatures is essential for safety,
   ↪ portability, and optimization. The "raw address" approach strips
   ↪ away all the benefits of the compiler's ABI knowledge and type
```

```
    ↪ checking.
```

USER (Message 29) [gemini-2.5-flash] 2025/06/14 23:44

```
is there term used for such an executable memory address, i.e. an
    ↪ address that can be called?
```

AGENT (Message 30) [gemini-2.5-flash] 2025/06/14 23:44

```
Yes, there are several terms used for such an executable memory address
    ↪ , depending on the context and the level of abstraction:

1.  **Entry Point (Address):** This is a very common and general term.
    ↪ It refers to the memory address where execution of a piece of
    ↪ code (like a function, a program, or a routine) begins. When you
    ↪ "call" or "jump" to something, you are typically targeting its
    ↪ entry point.

2.  **Function Pointer / Code Pointer:** This term is used when you
    ↪ have a *pointer variable* that holds the memory address of a
    ↪ function. In C/C++, `void (*func_ptr)();` declares a function
    ↪ pointer. The value stored in `func_ptr` is the executable memory
    ↪ address.

3.  **Instruction Pointer (IP) / Program Counter (PC):** While these
    ↪ refer to the *CPU's register* that holds the address of the next
    ↪ instruction to be executed, when you "call" an address, you are
    ↪ effectively loading that address into the IP/PC. So, an
    ↪ executable memory address is one that can be loaded into the IP/
    ↪ PC to begin execution.

4.  **Code Segment Address (or offset):** In segmented memory
    ↪ architectures (like older x86 real mode), code resided in a "code
    ↪  segment," and functions were referenced by an offset within that
    ↪  segment. So, the executable address would be a segment:offset
    ↪ pair. In modern flat memory models, this distinction is less
    ↪ common, and it's usually just a linear address.

5.  **Relocatable Address:** Before linking, the address of a function
    ↪ might just be an offset relative to the start of its object file'
    ↪ s code section. It becomes a fixed "executable memory address"
    ↪ only after the linker has resolved all relocations and assigned
    ↪ it a final position in the executable's virtual address space.

6.  **Trampoline (Address):** In some advanced scenarios (e.g., dynamic
    ↪  dispatch, hot-patching, or when dealing with certain thunks), a
    ↪ "trampoline" is a small piece of executable code at a specific
    ↪ address that then jumps to the real target. So, you might call a
    ↪ trampoline address which then redirects to the actual function.

7.  **Routine Address / Subroutine Address:** A more general term for
    ↪ the starting address of a block of code designed to perform a
    ↪ specific task.

8.  **Direct Address / Absolute Address:** When the memory address is
    ↪ fixed and known (e.g., a hardcoded address in a bootloader for a
    ↪ specific hardware routine). This contrasts with relative
    ↪ addresses or addresses that need relocation.

In the context of your discussion about jumping to "raw functions" or
    ↪ bypassing LLVM's ABI handling, you are most accurately referring
    ↪ to a **bare-bones memory address** that serves as an **entry
    ↪ point** for execution, effectively becoming a **code pointer**
    ↪ that you directly manipulate.
```

USER (Message 31) [gemini-2.5-flash] 2025/06/14 23:49

```
how to do comments in llvm ir?
```

AGENT (Message 32) [gemini-2.5-flash] 2025/06/14 23:49

```
In LLVM IR (both the human-readable ‘.ll‘ format and the internal
    ↪ representation), comments are straightforward:

Use a semicolon (‘;‘) at the beginning of a line. Everything from the
    ↪ semicolon to the end of the line is treated as a comment and is
    ↪ ignored by LLVM tools.

**Example:**

‘‘‘llvm
; This is a single-line comment.
; All comments in LLVM IR start with a semicolon.

define i32 @main() {
entry:
  ; Allocate space for an integer on the stack
  %1 = alloca i32, align 4

  ; Store the value 0 into the allocated memory
  store i32 0, ptr %1, align 4

  ; Load a global string for printf
  %2 = load ptr, ptr @.str.1, align 8 ; Load the address of the string
  ; Call the printf function
  ; First argument is the format string
  ; Second argument (variadic) is the integer value
  %call = call i32 (ptr, ...) @printf(ptr noundef %2, i32 42)

  ; Return 0 from main
  ret i32 0
}

; Declare the printf function, which is external
declare i32 @printf(ptr noundef, ...)

; Define a global string constant (used by printf)
@.str.1 = private unnamed_addr constant [15 x i8] c"The answer is %d\0A
   ↪ \00", align 1
‘‘‘

**Key Points about LLVM IR Comments:**

*   **Single-line only:** There’s no multi-line comment syntax like ‘/*
   ↪  ... */‘ in C. Each line you want commented must start with a
   ↪ semicolon.
*   **No nested comments:** Semicolons within a line *after* the first
   ↪ one are just part of the comment text.
*   **Ignored by tools:** The ‘llc‘, ‘opt‘, ‘llvm-as‘, ‘llvm-dis‘, and
   ↪ other LLVM utilities completely ignore comment lines. They are
   ↪ purely for human readability.
*   **Useful for debugging and understanding:** Adding comments to LLVM
   ↪  IR can be invaluable when you’re manually inspecting or
   ↪ transforming it, especially for complex optimization passes or
   ↪ when trying to understand compiler output.
```

USER (Message 33) [gemini-2.5-flash] 2025/06/14 23:59

```
how to assign to a register in llvm ir
```

AGENT (Message 34) [gemini-2.5-flash] 2025/06/14 23:59

```
You don’t directly "assign to a register" in LLVM IR in the way you
   ↪ might in assembly language (e.g., ‘mov eax, 5‘). LLVM IR operates
   ↪  on an **infinite set of virtual registers** (also called SSA
   ↪ registers or temporaries).

Instead of specific physical registers (like ‘eax‘, ‘rbx‘), you work
   ↪ with named values that act as symbolic registers. These virtual
   ↪ registers are prefixed with a percent sign (‘%‘).
```

~~~
### How to "Assign" to a Virtual Register

You "assign" a value to a virtual register by having an **instruction
   ↪ produce a result value**, and then assigning that result to a new
   ↪  virtual register name. Each instruction that produces a value
   ↪ effectively "writes" to a virtual register.

**Syntax:**

```llvm
%result_register = <instruction> [operands]
```

**Examples:**

1.  **Arithmetic Operations:**

    ```llvm
    define i32 @add_example(i32 %a, i32 %b) {
    entry:
      ; Add %a and %b, store the result in a new virtual register named
   ↪  %sum
      %sum = add i32 %a, %b

      ; Multiply %sum by 10, store in %product
      %product = mul i32 %sum, 10

      ret i32 %product
    }
    ```

2.  **Memory Operations (Loading):**
    When you load a value from memory, the loaded value is placed into
   ↪ a virtual register.

    ```llvm
    define i32 @load_example(ptr %ptr) {
    entry:
      ; Load the i32 value pointed to by %ptr, store it in %loaded_val
      %loaded_val = load i32, ptr %ptr, align 4
      ret i32 %loaded_val
    }
    ```

3.  **Function Calls:**
    If a function returns a value, that value is placed into a virtual
~~~

```
↪ register.

  ‘‘‘llvm
  declare i32 @get_value()

  define i32 @call_example() {
  entry:
    ; Call @get_value(), store its return value in %returned_val
    %returned_val = call i32 @get_value()
    ret i32 %returned_val
  }
  ‘‘‘

4.  **‘alloca‘ Instruction:**
  The ‘alloca‘ instruction allocates space on the stack and returns a
↪  *pointer* to that allocated memory. This pointer is stored in a
↪ virtual register.

  ‘‘‘llvm
  define void @stack_alloc_example() {
  entry:
    ; Allocate space for an i32 on the stack, store the pointer in %
↪ stack_ptr
    %stack_ptr = alloca i32, align 4

    ; Now you can use %stack_ptr to store/load values to/from that
↪ memory.
    store i32 10, ptr %stack_ptr, align 4
    ret void
  }
  ‘‘‘

### Virtual Registers vs. Physical Registers

*   **Virtual Registers (‘%name‘):** These are the abstract registers
↪ you work with in LLVM IR. They are in **Static Single Assignment
↪ (SSA)** form, meaning each virtual register is assigned a value
↪ exactly once. This property is extremely beneficial for
↪ optimization.
*   **Physical Registers (‘eax‘, ‘rbx‘, etc.):** These are the actual
↪ hardware registers on the target CPU. During the code generation
↪ phase (by ‘llc‘), LLVM’s register allocator maps the virtually
↪ infinite set of LLVM IR virtual registers to the finite set of
↪ available physical registers. This mapping is a complex
↪ optimization problem.

So, while you might think of ‘%sum‘ as "register ‘sum‘," it’s truly a
```

```
   ↪ symbolic temporary. The LLVM backend will eventually decide which
   ↪  physical register (or stack slot) that value will reside in
   ↪ during actual execution. You, as the IR writer, don't make that
   ↪ decision directly.
```

USER (Message 35) [gemini-2.5-flash] 2025/06/15 00:04

```
what if we want an external variable but we dont know its type, we just
   ↪  know theres some data there
```

AGENT (Message 36) [gemini-2.5-flash] 2025/06/15 00:04

```
If you genuinely don't know the type of an external variable and you
   ↪ just know "there's some data there," you are operating outside
   ↪ the normal, type-safe boundaries of LLVM IR and most compilers.

LLVM IR is **strongly typed**. Every value, including global variables,
   ↪  *must* have a defined type. You cannot declare a global variable
   ↪  with an unknown or generic type like `void*` for data in the
   ↪ same way you might for a function pointer that points to unknown
   ↪ code.

### Why You Can't Have an "Unknown Type" Global:

1.  **Memory Allocation and Access:** The type of a global variable
   ↪ dictates:
    *   **Size:** How much memory should be reserved for it.
    *   **Alignment:** How it needs to be aligned in memory.
    *   **Load/Store Instructions:** The machine code generated for `
   ↪ load` and `store` instructions directly depends on the size and
   ↪ type of the data being accessed. Loading a `float` is different
   ↪ from loading an `i32`.
2.  **Type Safety:** The entire LLVM ecosystem (optimizers, backends)
   ↪ relies on knowing the types to perform correct and safe
   ↪ operations.
3.  **Linker's Role:** While the linker resolves addresses, it also
   ↪ needs to know the size and alignment of symbols, often derived
   ↪ from their type.

### What You Have to Do (Even if You Don't "Know" It):

Even if *you* don't conceptually know the type, you *must* make a
   ↪ decision and provide a type in LLVM IR. The most common way to
   ↪ represent "raw data" when its specific structure isn't known to
   ↪ the current module is to treat it as a **pointer to an opaque or
   ↪ generic type**, or as an array of bytes.
```

~~~
Here are the practical approaches, ordered from more common to less
   ↪ ideal for truly unknown data:

1.  **Declare as `ptr` (Pointer to an Opaque Type):**
    This is the closest you can get to "I know it's a memory address,
   ↪ but I don't know what's there." You declare the external variable
   ↪  as a pointer (`ptr`). When you want to access the "data," you'd
   ↪ then `bitcast` this pointer to a more specific type before
   ↪ loading or storing.

    ```llvm
    ; Declares an external global variable named @unknown_data_ptr
    ; which is itself a pointer. The linker will resolve its address.
    ; This is common for things like global pointers to tables,
   ↪ structures, etc.
    @unknown_data_ptr = external global ptr

    ; --- How you'd use it: ---
    define i32 @access_unknown_data_as_i32() {
    entry:
      ; Load the pointer *value* from @unknown_data_ptr (which is a
   ↪ pointer to the data)
      %data_address = load ptr, ptr @unknown_data_ptr, align 8

      ; Now, bitcast this *address* to a pointer to the type you want
   ↪ to interpret it as
      %i32_ptr = bitcast ptr %data_address to ptr

      ; Load the i32 value from that interpreted address
      %loaded_val = load i32, ptr %i32_ptr, align 4
      ret i32 %loaded_val
    }
    ~~~
    This is very common for `void*` in C. The `ptr` type is LLVM's
   ↪ abstract pointer type, and it doesn't carry explicit pointee type
   ↪ information itself, making it effectively opaque for the
   ↪ variable *holding the pointer*.

2. **Declare as a Specific, Generic Base Type (e.g., `i8` or `[N x i8
   ↪]`):**
    If the external variable is truly just a block of raw bytes whose
   ↪ contents are not structured (e.g., a buffer), you can declare it
   ↪ as an `i8` (a single byte) or an array of bytes `[N x i8]`.

    ```llvm
    ; Declare an external global variable named @raw_byte_data
    ; as if it's a single byte.
```

```
    @raw_byte_data = external global i8

    ; --- How you'd use it to access data: ---
    define i8 @get_first_byte() {
    entry:
      %byte_val = load i8, ptr @raw_byte_data, align 1
      ret i8 %byte_val
    }

    ; To access later bytes, you'd use getelementptr:
    define i8 @get_third_byte() {
    entry:
      ; Get a pointer to the 3rd byte (index 2)
      %third_byte_ptr = getelementptr i8, ptr @raw_byte_data, i64 2
      %third_byte_val = load i8, ptr %third_byte_ptr, align 1
      ret i8 %third_byte_val
    }
    ‘‘‘
    This is useful if you are literally treating the data as a byte stream. If you then want to interpret a sequence of bytes as an ‘i32‘, you'd ‘bitcast‘ the ‘getelementptr‘ result to ‘ptr‘ before loading/storing.

### The Bottom Line:

Even for "unknown" external data, **you must give it a concrete type** in its LLVM IR declaration. The ‘ptr‘ type (if it's a pointer to the data) or ‘i8‘/‘[N x i8]‘ (if it's the data itself) are your closest options for maximum flexibility. You then rely on ‘bitcast‘ and ‘getelementptr‘ to interpret and navigate that raw memory when you access it.

This explicitly contrasts with the ‘...‘ for functions, which just indicates variadic arguments *of known types* that follow, not "unknown signature entirely." For data, the *size and structure* are baked into the type, and LLVM must know them.
```

USER (Message 37) [gemini-2.5-flash] 2025/06/15 00:05

```
does this add another dereferencing step, or is the compilation to assembly gonna insert this pointer directly into the compiled instructions?
```

AGENT (Message 38) [gemini-2.5-flash] 2025/06/15 00:05

```
You're asking a crucial question about the translation from LLVM IR to native assembly, specifically regarding the ‘ptr‘ type and ‘
```

```
    ↪ bitcast‘. Let’s clarify the dereferencing.

### LLVM IR’s ‘ptr‘ Type and Its Translation

In LLVM IR:

*   The ‘ptr‘ type is a **generic pointer type**. It represents a
    ↪ memory address. It *does not* inherently carry information about
    ↪ the type of data it points to.
*   Instructions like ‘load‘ and ‘store‘ *always* require a ‘ptr‘ as
    ↪ their address operand, but they also require an explicit data
    ↪ type to specify *what kind* of data is being loaded or stored (e.
    ↪ g., ‘load i32, ptr %addr‘).
*   ‘bitcast‘ is a **type-casting instruction**. It changes the *type*
    ↪ of a value *without changing its bit pattern*. It’s a no-op at
    ↪ runtime (it doesn’t generate any machine code instruction itself)
    ↪ . It purely exists for type-system purposes in the IR.
*   ‘getelementptr‘ (GEP) calculates an address. It’s also a no-op at
    ↪ runtime in terms of memory access; it just does pointer
    ↪ arithmetic.

### Analysis of the "Unknown Data" Example:

Let’s re-examine the two primary ways we discussed declaring "unknown"
    ↪ external data:

#### Scenario 1: Declaring the External as ‘external global ptr‘

‘‘‘llvm
; In file where variable is DEFINED (e.g., in C: int* unknown_data_ptr
    ↪ = &my_data_here;)
@unknown_data_ptr = external global ptr @some_actual_data_address ; or
    ↪ some constant

; In your LLVM IR module (where it’s DECLARED and USED)
@unknown_data_ptr = external global ptr ; Declaration

define i32 @access_unknown_data_as_i32() {
entry:
  ; 1. Load the pointer *value* from @unknown_data_ptr
  ;    This is the first **dereference** (reading the pointer itself
    ↪ from memory).
  %data_address = load ptr, ptr @unknown_data_ptr, align 8

  ; 2. Bitcast the *address* (which is a ptr type) to a pointer to i32.
  ;    This is a compile-time type conversion, **no runtime instruction
    ↪ **.
  %i32_ptr = bitcast ptr %data_address to ptr

  ; 3. Load the i32 value from the address %i32_ptr points to.
  ;    This is the second **dereference** (reading the i32 data from
   ↪ the address).
  %loaded_val = load i32, ptr %i32_ptr, align 4
  ret i32 %loaded_val
}
‘‘‘

**Assembly Translation:**

In this scenario, you will indeed have **two dereferencing steps (
   ↪ memory loads)** at the machine code level:

1.  A memory load to retrieve the *value* of the ‘unknown_data_ptr‘
   ↪ global variable. That value *is* a memory address.
2.  A second memory load from the address that was just retrieved, to
   ↪ get the actual ‘i32‘ data.

So, if ‘@unknown_data_ptr‘ points to another pointer, then yes, that’s
   ↪ two levels of indirection and two memory accesses.

#### Scenario 2: Declaring the External as ‘external global i8‘ (or ‘[N
   ↪  x i8]‘)

‘‘‘llvm
; In your LLVM IR module (where it’s DECLARED and USED)
@raw_byte_data = external global i8 ; Declaration, meaning
   ↪ @raw_byte_data *is* the data itself.

define i32 @get_third_byte_as_i32_via_bitcast() {
entry:
  ; 1. Calculate the address of the 3rd byte.
  ;    GEP only does pointer arithmetic; it’s a **calculation**, not a
   ↪ dereference.
  %third_byte_ptr = getelementptr i8, ptr @raw_byte_data, i64 2

  ; 2. Bitcast this calculated pointer to a pointer to i32.
  ;    This is a compile-time type conversion, **no runtime instruction
   ↪ **.
  %i32_ptr_from_gep = bitcast ptr %third_byte_ptr to ptr

  ; 3. Load the i32 value from that interpreted address.
  ;    This is the **first and only dereference** in this function.
  %loaded_val = load i32, ptr %i32_ptr_from_gep, align 4
  ret i32 %loaded_val
```

```
}
‘‘‘

**Assembly Translation:**

In this scenario, you will typically have **one dereferencing step (one
    ↪  memory load)** at the machine code level.

1.  The ‘getelementptr‘ will translate to an address calculation (e.g.,
    ↪  ‘LEA‘ instruction on x86, or adding an offset to a base register
    ↪ ). This is not a memory access.
2.  The ‘bitcast‘ will disappear completely.
3.  The ‘load‘ instruction will perform a single memory access from the
    ↪  calculated address.

### Conclusion

*   The ‘bitcast‘ instruction in LLVM IR itself **never generates a
    ↪ runtime instruction** that accesses memory. It’s purely a type-
    ↪ system instruction for the compiler.
*   The number of dereferencing steps at runtime depends entirely on
    ↪ the **‘load‘ and ‘store‘ instructions** you use, and the **type
    ↪ of the pointer** being loaded.
*   If your external global is declared as a ‘ptr‘ that *points to* the
    ↪  data you want, then yes, loading that pointer first constitutes
    ↪ one dereference, and then loading the data it points to is a
    ↪ second.
*   If your external global *is* the data itself (e.g., ‘i8‘, ‘i32‘, ‘[
    ↪ N x T]‘), then accessing it might only involve one dereference (
    ↪ unless you’re chaining ‘load‘s of pointers *from* that data).

This highlights the importance of correctly typing your external
    ↪ variables in LLVM IR to match how they are *actually* structured
    ↪ and used in memory, to avoid unnecessary indirection or incorrect
    ↪  accesses.
```

USER (Message 39) [gemini-2.5-flash] 2025/06/15 00:07

```
BUT I WANT TO THE ADDRESS TO BE KNOWN AT LINK TIME. it doesnt make
    ↪ sense to have a pointer here. how do i handle an arbitrarily
    ↪ sized unknown data without adding another dereferecning step. the
    ↪  plan is to use some dedicated assembly inline that will deal
    ↪ with this.
```

AGENT (Message 40) [gemini-2.5-flash] 2025/06/15 00:07

```
Okay, this clarifies your goal significantly! You’re saying:
```

```
1.  The "unknown data" is directly at a **known symbol address** that
    ↪ will be resolved by the linker.
2.  You want to access this data without an extra indirection (no ‘load
    ↪  ptr, ptr @unknown_data_ptr‘).
3.  You're willing to handle the actual data access within **inline
    ↪ assembly**.

This is achievable, but it still requires LLVM IR to understand the *
    ↪ pointer to* your data. The trick is how you declare the **type**
    ↪ of the global in LLVM IR when it's just a raw block of bytes.

### The Correct Way: Declare as ‘[0 x i8]‘ or ‘i8‘ at the Global Scope

The most appropriate LLVM IR declaration for an external block of raw,
    ↪ arbitrarily-sized data, whose address is directly known at link
    ↪ time, is to declare it as either:

1.  ‘[0 x i8]‘ (an empty array of bytes)
2.  ‘i8‘ (a single byte)

This signifies to LLVM that:

*   It's a global entity.
*   Its address will be resolved by the linker.
*   It consists of bytes.
*   It doesn't define any specific size for LLVM's memory management,
    ↪ just a starting point.

**Syntax:**

‘‘‘llvm
@[variable name] = external global [0 x i8]
; OR
@[variable name] = external global i8
‘‘‘

The ‘[0 x i8]‘ is often preferred as it cleanly indicates it's an array
    ↪  of bytes, even if of unknown size to *this* module. ‘i8‘ can
    ↪ imply a single byte, but ‘getelementptr‘ can still access beyond
    ↪ it.

### How to Access it with Inline Assembly (and No Extra Dereference)

Now, when you use this in your ‘asm‘ block, you'll need to pass the *
    ↪ address* of this global symbol to the inline assembly. This
    ↪ address will be resolved by the linker.
```

```
**Example LLVM IR:**

‘‘‘llvm
; Declare the external data block as an empty array of bytes.
; This tells LLVM it’s a global symbol whose address the linker knows,
; and it’s a pointer to bytes.
@my_raw_data_block = external global [0 x i8]

; Assuming some function where you want to access this data.
define void @process_raw_data() {
entry:
  ; Get a pointer to the start of the data block.
  ; GEP on a global symbol with no indices (or zero indices)
  ; effectively gives you the address of the global itself.
  ; This instruction *calculates* an address, it does not dereference.
  %data_ptr = getelementptr [0 x i8], ptr @my_raw_data_block, i64 0, i64
   ↪  0

  ; Now, use llvm.experimental.asm to directly access this address.
  ; Use an "r" constraint to get the address in a register.
  ; The specific assembly instructions will depend on your CPU/ABI.
  ; This example assumes x86-64 and tries to load the first 8 bytes
   ↪ into RAX/RBX.
  call void @llvm.experimental.asm(
    metadata !"movq (%0), %%rax\n\tmovq 8(%0), %%rbx", ; Assembly to
   ↪ load 16 bytes directly from %0
    metadata !"r"(ptr %data_ptr)                       ; Pass the
   ↪ data_ptr (address) as an input operand in a register
  ) #0 ; Assuming #0 might include "volatile" and "memory" clobber.

  ret void
}

; Declare the intrinsic (signature varies based on operands)
declare void @llvm.experimental.asm(metadata, metadata, ...)

; Some attributes needed for asm (usually generated by clang for
   ↪ volatile asm)
attributes #0 = { "att_dialect" "inlineasm" "no-builtins" "thunk" "
   ↪ volatile-asm" }
‘‘‘

**Explanation of Dereferencing:**

1.  **‘%data_ptr = getelementptr [0 x i8], ptr @my_raw_data_block, i64
   ↪ 0, i64 0‘**
```

```
    *   This GEP instruction does **not** involve a memory dereference.
    *   It calculates the *address* of the global symbol ‘
    ↪ @my_raw_data_block‘.
    *   ‘ptr @my_raw_data_block‘ itself is a constant expression
    ↪ representing the *address* of the global symbol.
    *   The ‘getelementptr‘ just returns that address.
    *   In the final machine code, this will often resolve directly to
    ↪ the linker-resolved address being used as an immediate operand in
    ↪  an instruction, or loaded into a register. It will *not* be a ‘
    ↪ load‘ instruction.

2.  **‘call void @llvm.experimental.asm(...)‘**
    *   You pass ‘%data_ptr‘ to the ‘asm‘ intrinsic using the ‘"r"‘
    ↪ constraint. This tells LLVM to place the value of ‘%data_ptr‘ (
    ↪ which is the address of ‘my_raw_data_block‘) into a general-
    ↪ purpose register.
    *   The inline assembly ‘movq (%0), %%rax‘ then directly loads from
    ↪  the memory address held in that register.
    *   **This is your single, intended dereference, occurring within
    ↪ the inline assembly itself.**

### This Approach Achieves Your Goal:

*   The LLVM IR side does not introduce an extra dereference.
*   The address of ‘@my_raw_data_block‘ is a linker-resolved symbol.
*   You control the specific memory access inside the inline assembly
    ↪ using the address provided to you by LLVM.

This is the standard and correct way to bridge the gap between LLVM's
    ↪ symbolic global addresses and raw byte-level access via inline
    ↪ assembly without adding unexpected layers of indirection.
```

USER (Message 41) [gemini-2.5-flash] 2025/06/15 00:09

```
what does #0 mean?
```

AGENT (Message 42) [gemini-2.5-flash] 2025/06/15 00:09

```
That's a good question! The ‘#0‘ (or ‘#1‘, ‘#2‘, etc.) you see after a
    ↪ function declaration or definition in LLVM IR refers to an **
    ↪ attribute group**.

### What are Attribute Groups?

In LLVM IR, attributes are metadata that provide additional information
    ↪  about a function, its parameters, or its return value. This
    ↪ information helps the optimizer make better decisions and can
    ↪ also influence code generation.

When multiple functions share the same set of attributes, instead of
    ↪ repeating the full list of attributes for each function, LLVM
    ↪ allows you to define an **attribute group** at the module level.
    ↪ You then refer to this group by its numerical ID (‘#0‘, ‘#1‘, etc
    ↪ .).

### Structure in LLVM IR:

Attribute groups are defined at the end of an LLVM IR file (or usually
    ↪ near the end for readability) using the ‘attributes‘ keyword:

‘‘‘llvm
; ... (function declarations and definitions) ...

attributes #0 = { attribute1 attribute2 "string-attribute" }
attributes #1 = { attribute3 "another-string-attribute" }
‘‘‘

Then, in function declarations or definitions, you can reference them:

‘‘‘llvm
define void @my_func() #0 {
  ; ...
}

declare i32 @external_lib_func(ptr) #1
‘‘‘

### Example Breakdown:

Let’s revisit the ‘llvm.experimental.asm‘ example:

‘‘‘llvm
call void @llvm.experimental.asm(
  metadata !"movq (%0), %%rax\n\tmovq 8(%0), %%rbx",
  metadata !"r"(ptr %data_ptr)
) #0 ; <-- This refers to attribute group #0
‘‘‘

And then, typically at the bottom of the ‘.ll‘ file:

‘‘‘llvm
attributes #0 = { "att_dialect" "inlineasm" "no-builtins" "thunk" "
    ↪ volatile-asm" }
‘‘‘
```

```
In this specific case for `llvm.experimental.asm`, the attributes in group `#0` are often automatically generated by frontends like Clang when you use `asm volatile`. These attributes are critical for telling LLVM's optimizer and code generator how to treat the inline assembly block:

*   **`"inlineasm"`**: This is a core attribute that marks this call as  inline assembly.
*   **`"volatile-asm"`**: Corresponds to the `volatile` keyword in C/C++ `asm`. It prevents the optimizer from reordering, merging, or removing the assembly block, ensuring it executes exactly as specified and that its side effects are preserved.
*   **`"att_dialect"` / `"intel_dialect"`**: Specifies the assembly syntax being used within the string.
*   **`"no-builtins"`**: Implies the inline assembly doesn't rely on or  interact with compiler builtins.
*   **`"thunk"`**: A specific detail related to how the code might be called.

### Why Use Attribute Groups?

1.  **Readability:** Prevents long lists of attributes from cluttering every function signature.
2.  **Maintainability:** If you need to change a common set of attributes (e.g., adding a new optimization hint), you only change it in one place (the attribute group definition).
3.  **Efficiency:** For LLVM's internal representation, it's more efficient to store and process a pointer to a common attribute group rather than duplicating the entire set of attributes for every function.

So, `#0` is just a shorthand alias for a predefined set of function attributes.
```

Thread (4 messages)
ID: 765f7232-673f-4bf1-a4e4-a927957dbf00 **2025/06/15 00:29**

USER (Message 1) [gemini-2.5-flash] 2025/06/15 00:29

```
does llvm ir have namespacing options when performing static linkage
```

AGENT (Message 2) [gemini-2.5-flash] 2025/06/15 00:29

```
LLVM IR itself doesn't have a direct "namespacing" construct in the same way C++ or other high-level languages do. However, when
```

```
  ↪ performing static linkage, you achieve a similar effect of
  ↪ avoiding name collisions through the following mechanisms:

1.  **Global Variable and Function Naming:**
    LLVM IR uses string identifiers for global variables and functions.
  ↪  When you link multiple LLVM modules (e.g., from different source
  ↪  files or libraries), these global names exist in a single, flat
  ↪ namespace within the linked executable or library.

    *   **Mangling:** High-level languages like C++ use name mangling (
  ↪ e.g., Itanium C++ ABI for GCC/Clang on Linux) to encode namespace
  ↪  information, class names, function signatures, and more into the
  ↪  symbol name. When C++ code is compiled to LLVM IR, the mangled
  ↪ names are preserved. So, while LLVM IR doesn't have a ‘namespace‘
  ↪  keyword, the mangled names effectively create unique symbols
  ↪ even if the original source code had identically named entities
  ↪ in different namespaces.

        For example, ‘MyNamespace::MyClass::myFunction(int)‘ might be
  ↪ mangled into something like ‘
  ↪ _ZN9MyNamespace7MyClass10myFunctionEi‘.

    *   **Local (Module-Private) Linkage:** LLVM IR has different
  ↪ linkage types that affect symbol visibility during linking. For
  ↪ static linkage, the most relevant are:
        *   ‘external‘: Symbols visible to other modules. This is the
  ↪ default for most functions and global variables.
        *   ‘internal‘ (or ‘static‘ in C/C++): Symbols only visible
  ↪ within the current compilation unit (LLVM module). These symbols
  ↪ will not conflict with identically named symbols in other modules
  ↪  because they are not exported. This is the primary mechanism for
  ↪  "namespacing" at the LLVM IR level for symbols that are not
  ↪ meant to be shared across modules.
        *   ‘private‘: Similar to ‘internal‘ but even stronger in that
  ↪ the symbol's address is not exposed.

2.  **Weak Symbols:**
    LLVM IR supports weak linkage (‘weak‘ and ‘weak_odr‘). This allows
  ↪ multiple definitions of a symbol, with the linker choosing one (
  ↪ usually the non-weak one, or the first weak one encountered).
  ↪ While not directly namespacing, it can be used in scenarios where
  ↪  you want to provide default implementations that can be
  ↪ overridden by user-provided ones, which can sometimes be part of
  ↪ a larger "module" or "component" strategy.

3.  **Symbol Hiding and Versioning (at the linker level):**
    While not strictly an LLVM IR feature, the tools that consume LLVM
  ↪ IR (like ‘lld‘, GNU ‘ld‘, ‘gold‘) offer mechanisms for
  ↪ controlling symbol visibility and versioning in the final linked
  ↪ binary.

    *   **Version Scripts:** For shared libraries, you can use version
  ↪ scripts to explicitly control which symbols are exported and
  ↪ their versions. This provides a very granular way to define the
  ↪ public interface of a library, effectively "namespacing" the
  ↪ symbols you want to expose.
    *   **Hidden Visibility:** You can mark symbols as hidden or
  ↪ protected at the assembly or object file level (e.g., using GCC’s
  ↪  ‘__attribute__((visibility("hidden")))‘). This prevents them
  ↪ from being exported from a shared library, ensuring they don’t
  ↪ clash with symbols in other libraries that might be loaded into
  ↪ the same process.

**In summary:**

LLVM IR itself does not have a ‘namespace‘ keyword or a concept of
  ↪ nested symbol scopes beyond the flat global scope and module-
  ↪ private scope. However, the effect of namespacing is achieved
  ↪ through:

*   **Mangled names** from higher-level languages (like C++) which make
  ↪  symbols globally unique.
*   **Linkage types** like ‘internal‘ or ‘private‘ which restrict
  ↪ symbol visibility to the current LLVM module, preventing clashes
  ↪ with identically named symbols in other modules.
*   **External linker features** like version scripts and symbol
  ↪ visibility attributes for controlling exported symbols in the
  ↪ final binary.

So, while you won’t see ‘namespace‘ declarations in LLVM IR, the
  ↪ underlying mechanisms are robust enough to manage symbols
  ↪ effectively during static linkage.
```

USER (Message 3) [gemini-2.5-flash] 2025/06/15 00:29

```
so the onus is on me to deal with this?
```

AGENT (Message 4) [gemini-2.5-flash] 2025/06/15 00:29

```
Yes, largely, the "onus" of managing potential name collisions and
    ↪ visibility is on you as the developer, but with significant
    ↪ assistance from the tools you use:

1.  **High-Level Language Compilers (like Clang/GCC for C++/Rust):**
```

```
    If you’re writing code in a language like C++ or Rust that *has*
  ↪ namespaces, the compiler (Clang in the case of LLVM)
  ↪ automatically handles the most common and complex part of this
  ↪ for you: **name mangling**. It translates your C++ ‘MyNamespace::
  ↪ myFunction‘ into a unique symbol name in the LLVM IR (and
  ↪ subsequent object files) that is highly unlikely to conflict with
  ↪  another ‘myFunction‘ from a different namespace or class. So, in
  ↪  this common scenario, you write your code with namespaces, and
  ↪ the compiler deals with the LLVM IR representation.

2.  **Using ‘static‘ or Anonymous Namespaces (in C/C++):**
    If you have functions or global variables that are only intended
  ↪ for use within a single compilation unit (source file), using the
  ↪  ‘static‘ keyword (in C) or an anonymous namespace (in C++) is
  ↪ your responsibility. When Clang compiles this, it will emit these
  ↪  symbols in LLVM IR with ‘internal‘ linkage, ensuring they are
  ↪ not exported and will not conflict with identical names in other
  ↪ object files.

3.  **Explicit Visibility Control (for Libraries):**
    If you are developing a library (especially a shared library), then
  ↪  yes, it’s your responsibility to define its public API. This
  ↪ means explicitly deciding which symbols should be exported and
  ↪ which should remain internal. Tools and language features assist
  ↪ with this:
    *   ‘__attribute__((visibility("hidden")))‘ or ‘__declspec(
  ↪ dllexport)/__declspec(dllimport)‘: These are compiler-specific
  ↪ attributes that allow you to control symbol visibility. You would
  ↪  typically use these on your public API functions/variables.
    *   Version Scripts (for shared libraries on Linux/Unix): If you
  ↪ need very fine-grained control over exported symbols and their
  ↪ versions, you write a linker script. This is an advanced topic
  ↪ and not always necessary for simple static linking, but crucial
  ↪ for robust shared libraries.

4.  **Direct LLVM IR Generation (Advanced):**
    If you are directly generating LLVM IR (e.g., writing a custom
  ↪ language compiler or an obfuscator), then **yes, the entire onus
  ↪ is on you**. You must:
    *   Ensure all global variable and function names you generate are
  ↪ unique or use ‘internal‘ linkage appropriately.
    *   Implement your own name mangling scheme if your source language
  ↪  supports concepts like namespaces or overloading.

**In essence:**

*   **For most application developers using high-level languages:** The
```

```
  ↪  compiler handles most of the complexity of name uniqueness (via
  ↪ mangling) and local visibility (via ‘static‘/anonymous namespaces
  ↪ ). Your responsibility is to use these language features
  ↪ correctly.
*   **For library developers:** You have an additional responsibility
  ↪ to manage the public interface of your library using visibility
  ↪ attributes or linker scripts.
*   **For those directly generating LLVM IR:** You are fully
  ↪ responsible for all naming and linkage decisions.

So, it’s not a burden that you bear entirely alone, as the compiler and
  ↪  linker tools provide powerful mechanisms to assist you based on
  ↪ your source code. You just need to be aware of these mechanisms
  ↪ and use them appropriately for your project’s needs.
```

Thread (8 messages)
ID: 054b94b1-6432-4f9c-bcb6-42b0e0b916c5 **2025/06/15 13:18**

USER (Message 1) [gemini-2.5-flash] 2025/06/15 13:18

```
I am writing some llvm ir by hand for optimization purposes, i plan on
  ↪ linking all my ir units using llvm which will combine it all. Im
  ↪ wondering is it possible to export an inline function from one
  ↪ unit into another unit?
```

AGENT (Message 2) [gemini-2.5-flash] 2025/06/15 13:18

```
No, you cannot directly "export" an inline function from one LLVM IR
  ↪ unit to be inlined into another unit in the way you might think
  ↪ of a traditional export/import mechanism for functions.

Here’s why and what you can do instead:

**Why Direct Export of Inline Functions Doesn’t Work as Expected:**

*   **Inlining is a Code Transformation:** Inlining is an optimization
  ↪ pass that replaces a function call with the body of the called
  ↪ function. This happens *during* compilation, not as a linking-
  ↪ time export/import.
*   **LLVM Linker’s Role:** The LLVM linker (‘llvm-link‘) combines
  ↪ multiple LLVM IR modules into a single module. Its primary
  ↪ purpose is to resolve external function declarations and
  ↪ definitions, making them visible across the combined module. It
  ↪ doesn’t perform inlining itself.
*   **‘alwaysinline‘ and ‘inline‘ Attributes:**
    *   The ‘alwaysinline‘ function attribute tells the optimizer to *
```

```
   ↪ always* inline a function if possible.
    *   The ‘inline‘ attribute suggests that inlining is desirable, but
   ↪  the optimizer has the final say.
    *   These attributes are hints to the inliner pass. They don’t
   ↪ magically make a function available for inlining *before* the IR
   ↪ units are combined.

**What You Need to Do:**

For an inline function to be considered for inlining across different
   ↪ original source files (which become different LLVM IR units
   ↪ before linking), you need to ensure the following:

1.  **Combine the LLVM IR Units First:** You *must* link your LLVM IR
   ↪ units into a single module *before* running any inlining
   ↪ optimization passes. The ‘llvm-link‘ tool is perfect for this.

    ‘‘‘bash
    llvm-link unit1.ll unit2.ll -o combined.ll
    ‘‘‘

2.  **Define the Function in All Units (or in one linked unit) and Use
   ↪ Linkage:**
    *   **External Linkage (‘external‘):** If you define a function
   ↪ with ‘external‘ linkage in one IR unit, other units can declare
   ↪ it and then ‘llvm-link‘ will resolve the definition. Once linked
   ↪ into a single module, the inliner can see the definition and
   ↪ potentially inline it.
    *   **Linker-Specific Behavior for Inlining Hints:** The ‘
   ↪ alwaysinline‘ and ‘inline‘ attributes are meaningful to the
   ↪ inliner pass once all the code is in the same module. If your
   ↪ function is defined in ‘unit1.ll‘ with ‘alwaysinline‘ and called
   ↪ in ‘unit2.ll‘, after ‘llvm-link‘ combines them, the inliner will
   ↪ then be able to act on that hint.

3.  **Run Inlining Pass on the Combined Module:** After linking, you
   ↪ apply the inlining pass (and other optimization passes) to the
   ↪ combined LLVM IR module.

    ‘‘‘bash
    opt -O3 -S combined.ll -o optimized.ll
    ‘‘‘

    The ‘-O3‘ flag includes the inliner pass as part of a set of common
   ↪  optimizations. You can also specifically run the inliner pass:

    ‘‘‘bash
    opt -inline -S combined.ll -o optimized.ll
    ‘‘‘

**Example Scenario:**

Let’s say you have ‘unit1.ll‘ and ‘unit2.ll‘.

**‘unit1.ll‘:**

‘‘‘llvm
define internal fastcc void @my_inline_func(i32 %arg) alwaysinline {
  %1 = add i32 %arg, 10
  call void @print_int(i32 %1)
  ret void
}

declare void @print_int(i32)
‘‘‘

**‘unit2.ll‘:**

‘‘‘llvm
define void @caller_func() {
  call fastcc void @my_inline_func(i32 5)
  ret void
}

declare fastcc void @my_inline_func(i32)
declare void @print_int(i32)
‘‘‘

**Steps to achieve inlining:**

1.  **Link:**
    ‘‘‘bash
    llvm-link unit1.ll unit2.ll -o combined.ll
    ‘‘‘
    At this point, ‘my_inline_func‘ from ‘unit1.ll‘ and its declaration in ‘unit2.ll‘ will be resolved. The definition will be visible.

2.  **Optimize (Inline):**
    ‘‘‘bash
    opt -inline -S combined.ll -o optimized.ll
    ‘‘‘
    Now, the ‘opt‘ tool will see the ‘alwaysinline‘ attribute on ‘my_inline_func‘ in ‘combined.ll‘ and will attempt to inline it into ‘caller_func‘.
```

```
The key takeaway is that inlining is an optimization that operates on a
    ↪  *complete* module. You use ‘llvm-link‘ to create that complete
    ↪ module from your smaller units.
```

USER (Message 3) [gemini-2.5-flash] 2025/06/15 13:20

```
What i mean is, can i declare alwaysinline on a function that’s not
    ↪ internal?
```

AGENT (Message 4) [gemini-2.5-flash] 2025/06/15 13:20

```
Yes, absolutely! You can, and often should, declare ‘alwaysinline‘ on a
    ↪  function that has ‘external‘ or ‘linkonce_odr‘ linkage, not just
    ↪  ‘internal‘.

Here’s why and what it implies:

*   **‘alwaysinline‘ and Linkage are Orthogonal Concepts:**
    *   **Linkage (‘internal‘, ‘external‘, ‘linkonce_odr‘, etc.):**
    ↪ This determines the visibility and behavior of a global symbol (
    ↪ like a function) during the linking process. It tells the linker
    ↪ how to handle multiple definitions or declarations of the same
    ↪ symbol across different object files or modules.
    *   **‘alwaysinline‘ Function Attribute:** This is a hint to the
    ↪ LLVM inliner pass. It tells the inliner that if it encounters a
    ↪ call to this function, it should *always* attempt to replace the
    ↪ call with the function’s body, regardless of its heuristics (like
    ↪  function size, call frequency, etc.).

*   **Common Scenarios for ‘alwaysinline‘ on Non-Internal Functions:**

    1.  **Header-Defined Functions in C/C++:** When you write ‘inline‘
    ↪ functions in C++ headers (or C functions with ‘static inline‘ or
    ↪ ‘__attribute__((always_inline))‘), the compiler often generates a
    ↪  definition of that function in every translation unit that
    ↪ includes the header. These definitions typically have ‘
    ↪ linkonce_odr‘ linkage in LLVM IR. The ‘alwaysinline‘ attribute
    ↪ ensures that if the function is called within the same
    ↪ translation unit, it will be inlined. If it’s not inlined (e.g.,
    ↪ due to a taken address or some other factor that prevents
    ↪ inlining), the ‘linkonce_odr‘ linkage ensures that only one copy
    ↪ of the function exists in the final executable after linking.

    2.  **Shared Utility Functions in LLVM IR:** If you have a small,
    ↪ performance-critical utility function that you want to be
    ↪ aggressively inlined wherever it’s called, even if it’s called
```

```
   ↪ from multiple distinct LLVM IR units that you later ‘llvm-link‘,
   ↪ you would declare it with ‘external‘ or ‘linkonce_odr‘ linkage
   ↪ and the ‘alwaysinline‘ attribute.

        **Example:**

        Let’s say ‘MyUtility.ll‘ defines a helper function:

        ‘‘‘llvm
        ; MyUtility.ll
        define linkonce_odr i32 @fast_add(i32 %a, i32 %b) alwaysinline {
          %sum = add i32 %a, %b
          ret i32 %sum
        }
        ‘‘‘

        And ‘MainProgram.ll‘ uses it:

        ‘‘‘llvm
        ; MainProgram.ll
        declare i32 @fast_add(i32, i32)

        define i32 @main() {
          %result = call i32 @fast_add(i32 10, i32 20)
          ret i32 %result
        }
        ‘‘‘

        When you ‘llvm-link MyUtility.ll MainProgram.ll -o combined.ll‘
   ↪  and then run ‘opt -O3 combined.ll -o optimized.ll‘, the ‘
   ↪ fast_add‘ function will be inlined into ‘main‘ in the ‘optimized.
   ↪ ll‘ output.

**Important Considerations:**

*   **Definition Must Be Available:** Regardless of the linkage or ‘
   ↪ alwaysinline‘ attribute, the *definition* of the function must be
   ↪  present in the module when the inliner pass runs. This is why
   ↪ you first ‘llvm-link‘ your IR units.
*   **Overriding ‘alwaysinline‘:** While ‘alwaysinline‘ is a strong
   ↪ hint, there are still situations where a function might not be
   ↪ inlined:
    *   If its address is taken (e.g., it’s assigned to a function
   ↪ pointer).
    *   If it’s called indirectly through a function pointer.
    *   If it’s recursive (though LLVM might inline a few levels of
   ↪ recursion if specified by other attributes or passes).
```

```
    *   If some other very strong optimization reason prevents it (
    ↪ though `alwaysinline` usually overrides most heuristics).
*   **Code Bloat:** Be cautious with `alwaysinline`. While it can be
    ↪ great for performance in specific hot paths, overuse can lead to
    ↪ significant code bloat, increasing binary size and potentially
    ↪ negatively impacting instruction cache performance. Use it
    ↪ judiciously for small, frequently called, performance-critical
    ↪ functions.

In summary, yes, `alwaysinline` works perfectly fine with `external` or
    ↪  `linkonce_odr` linkage, and this is a common and intended use
    ↪ case in LLVM IR for aggressive inlining across translation unit
    ↪ boundaries after linking.
```

USER (Message 5) [gemini-2.5-flash] 2025/06/15 13:22

```
You're not understanding me. You mentioned "if the function is called
    ↪ within the same translation unit". I want this inlining hint to
    ↪ span ir units (not span object file units, i know that's
    ↪ impossible, but since we use llvm i want to span llvm ir units)
```

AGENT (Message 6) [gemini-2.5-flash] 2025/06/15 13:22

```
My apologies! I understand now. You are explicitly interested in the `
    ↪ alwaysinline` attribute having an effect across distinct LLVM IR
    ↪ modules *before* they are linked, or rather, the ability for the
    ↪ inliner to act on it once they are linked, regardless of where
    ↪ the call site or definition originated.

Let me clarify directly based on your specific concern:

**The `alwaysinline` attribute *absolutely* functions as an inlining
    ↪ hint across different LLVM IR units, provided those units are
    ↪ combined into a single module *before* the inliner pass runs.**

You are correct that it cannot cause inlining to happen between
    ↪ separately compiled object files during traditional system
    ↪ linking (like `ld`). However, when working purely within the LLVM
    ↪  ecosystem:

1.  **Your "IR units" become one module:** When you use `llvm-link
    ↪ unit1.ll unit2.ll -o combined.ll`, you are effectively merging `
    ↪ unit1.ll` and `unit2.ll` into a *single, larger LLVM IR module*
    ↪ (`combined.ll`).
2.  **The `opt` tool processes the single module:** When you then run `
    ↪ opt -inline combined.ll -o optimized.ll`, the `opt` tool (which
    ↪ contains the inliner pass) sees `combined.ll` as one cohesive
```

~~~
    ↪ unit. At this point, the inliner can see the definition of `
    ↪ FunctionA` from `unit1.ll` and a call to `FunctionA` from `unit2.
    ↪ ll`, and it can apply the `alwaysinline` hint because all the
    ↪ necessary information (definition and call site) is now present
    ↪ within the *same logical module*.

**To be crystal clear, if you have:**

*   **`unit1.ll`:**
    ```llvm
    ; Definition of a function with alwaysinline
    define external void @my_inlineable_function(i32 %arg) alwaysinline
    ↪ {
      ; ... function body ...
      ret void
    }
    ```

*   **`unit2.ll`:**
    ```llvm
    ; Declaration and call to the function defined in unit1.ll
    declare void @my_inlineable_function(i32)

    define void @caller_in_unit2() {
      call void @my_inlineable_function(i32 42)
      ret void
    }
    ```

**The workflow is:**

1.  **Link the IR units:**
    ```bash
    llvm-link unit1.ll unit2.ll -o combined.ll
    ```
    Result: `combined.ll` now contains both the definition of `
    ↪ @my_inlineable_function` and the call to it. The `alwaysinline`
    ↪ attribute is preserved on the definition.

2.  **Optimize (including inlining):**
    ```bash
    opt -O3 -S combined.ll -o optimized.ll
    ```
    OR specifically:
    ```bash
    opt -inline -S combined.ll -o optimized.ll
    ```
~~~

```
    Result: The ‘opt‘ tool, acting on ‘combined.ll‘, sees the ‘
    ↪ alwaysinline‘ attribute on ‘@my_inlineable_function‘ and will
    ↪ aggressively inline its body into ‘@caller_in_unit2‘. The fact
    ↪ that ‘my_inlineable_function‘ originated in ‘unit1.ll‘ and ‘
    ↪ @caller_in_unit2‘ in ‘unit2.ll‘ becomes irrelevant once ‘llvm-link
    ↪ ‘ has combined them into a single module.

This is the standard and intended way to achieve cross-unit inlining
    ↪ hints in LLVM IR. The ‘alwaysinline‘ attribute persists through
    ↪ the linking process (‘llvm-link‘) and then guides the
    ↪ optimization passes (‘opt‘) on the newly merged module.
```

USER (Message 7) [gemini-2.5-flash] 2025/06/15 13:32

```
Am i right to say that the c language does not allow you to give hints
    ↪ like than in source files that would span compilation units?
    ↪ However if clang goes to llvm first, i wonder does it achieve
    ↪ this desired result or does it resort to compiling each one to
    ↪ object files?
```

AGENT (Message 8) [gemini-2.5-flash] 2025/06/15 13:32

```
You are absolutely right about the C language itself!

### C Language and Cross-Compilation Unit Inlining Hints

**You are correct:** The C language, at its source level, does *not*
    ↪ provide a direct, standard mechanism (like an ‘alwaysinline‘
    ↪ keyword) that guarantees a function defined in one ‘.c‘ file will
    ↪  be inlined into a call site in a different ‘.c‘ file *across
    ↪ separate compilation processes*.

*   **‘inline‘ keyword in C99/C11:** The ‘inline‘ keyword in C is
    ↪ primarily a hint, but its main purpose is to allow multiple
    ↪ definitions of the same function (e.g., in a header file included
    ↪  by multiple ‘.c‘ files) without violating the One Definition
    ↪ Rule. It suggests inlining, but compilers are free to ignore it.
    ↪ Crucially, if a function with ‘inline‘ linkage is not inlined, a
    ↪ separate, non-inlineable definition (often with ‘external‘
    ↪ linkage) must be provided in exactly one compilation unit, or
    ↪ implicitly generated by the compiler.
*   **‘static inline‘:** This is effectively a private, translation-
    ↪ unit-local inline function. It won’t even be visible to other
    ↪ compilation units, so cross-unit inlining is impossible.
*   **Compiler-specific extensions:** Some compilers offer extensions
    ↪ like GCC’s ‘__attribute__((always_inline))‘ or MSVC’s ‘
    ↪ __forceinline‘. While these are stronger hints, they typically
    ↪ apply *within the current compilation unit*. For cross-unit
    ↪ inlining, the compiler would still need to see the function's
    ↪ definition when compiling the call site.

**The fundamental limitation is:** A traditional C compiler processes
    ↪ one ‘.c‘ file (a "compilation unit") at a time, generating an
    ↪ object file (‘.o‘). At this stage, it only has the full
    ↪ definition of functions *within that ‘.c‘ file*. It doesn't have
    ↪ the body of functions defined in other ‘.c‘ files, so it can't
    ↪ inline them. Inlining *must* happen when the full definition is
    ↪ available.

### How Clang/LLVM Achieves Cross-Compilation Unit Inlining

This is precisely where the power of LLVM's Intermediate Representation
    ↪  (IR) and its whole-program optimization capabilities come into
    ↪ play!

**Clang (and GCC, when used with LTO) bridges this gap through a
    ↪ technique called Link-Time Optimization (LTO).**

Here's how it works with Clang/LLVM:

1.  **Front-end (Clang) produces LLVM IR, not object files directly:**
    Instead of compiling each ‘.c‘ file directly to a native object
    ↪ file, Clang compiles each ‘.c‘ file into an LLVM Bitcode file (‘.
    ↪ bc‘ or ‘.ll‘). This Bitcode file contains the LLVM IR
    ↪ representation of that compilation unit, including function
    ↪ definitions, declarations, and crucially, any ‘alwaysinline‘
    ↪ attributes that were derived from source code attributes like ‘
    ↪ __attribute__((always_inline))‘.

    ‘‘‘bash
    clang -emit-llvm -c source1.c -o source1.bc
    clang -emit-llvm -c source2.c -o source2.bc
    ‘‘‘

    *If* ‘source1.c‘ had:
    ‘‘‘c
    // source1.c
    __attribute__((always_inline))
    int add_one(int x) {
        return x + 1;
    }
    ‘‘‘
    Then ‘source1.bc‘ would contain the LLVM IR for ‘add_one‘ with the
    ↪ ‘alwaysinline‘ attribute.
```

```
2.  **LLVM Linker combines IR:**
    When you link these ‘.bc‘ files using ‘clang‘ with LTO enabled (or
  ↪ ‘llvm-link‘ manually), the LLVM linker (‘lld‘ or ‘llvm-link‘
  ↪ internally) combines all these separate LLVM IR modules into a *
  ↪ single, monolithic LLVM IR module*.

    ‘‘‘bash
    clang source1.bc source2.bc -flto -o my_executable
    # Or manually for demonstration:
    # llvm-link source1.bc source2.bc -o combined.bc
    ‘‘‘

3.  **Whole-Program Optimization (including Inlining) on combined IR:**
    After the IR units are combined, the LLVM optimization passes (
  ↪ including the inliner) are run on this *single, combined LLVM IR
  ↪ module*. At this stage, the inliner has a complete view of the
  ↪ entire program’s code, including the definition of ‘add_one‘ (
  ↪ from ‘source1.bc‘) and any calls to it (from ‘source2.bc‘). If ‘
  ↪ add_one‘ had the ‘alwaysinline‘ attribute (derived from ‘
  ↪ __attribute__((always_inline))‘), the inliner will see that hint
  ↪ and inline it into calls even if those calls originated in ‘
  ↪ source2.c‘.

    ‘‘‘bash
    # This step is implicit with ’clang -flto’, but conceptually:
    # opt -O3 combined.bc -o optimized.bc
    ‘‘‘

4.  **Backend generates machine code:**
    Finally, this optimized, combined LLVM IR module is passed to the
  ↪ LLVM backend to generate the final machine code for the
  ↪ executable.

**In summary:**

*   **C source level:** No direct cross-unit inlining guarantee.
*   **Clang/LLVM with LTO:** Achieves precisely this cross-unit
  ↪ inlining by deferring the full compilation to native code until
  ↪ all compilation units’ LLVM IR is merged into a single module.
  ↪ The ‘__attribute__((always_inline))‘ hint propagates to the LLVM
  ↪ IR ‘alwaysinline‘ attribute, which is then honored by the LTO-
  ↪ stage inliner.

So, while C itself doesn’t offer it, the LLVM toolchain, when used with
  ↪  LTO, absolutely delivers the desired result of inlining
  ↪ functions across what were originally separate source files/
```

compilation units.

Thread (2 messages)
ID: 28da42c3-1ae9-4987-b23e-0595750ca2e5 **2025/06/16 09:43**

USER (Message 1) [gemini-2.5-flash] 2025/06/16 09:43

i rawdogging coding using llvm ir due to crucial optimization works i must perform. however i need to do a little transformation to my ir files. to prevent naming conflicts, i'd like to be able to take an "ll" file (or its binary equivalent) and rename all external references (whether that be an implementation of an external function, or the dependency on an externally-implemented function). this would require changing some names at the top level and then perhaps anywhere deeper where these names are referenced. does llvm offer a standard mechanism for performing these renames so that i can link further with other object files in the rest of my build process. i confidently have access to the desired name mappings in the rest of my build process, so it's just a matter of needing to perform these renames (essentially ensuring my ir modules are adequately scoped)

AGENT (Message 2) [gemini-2.5-flash] 2025/06/16 09:43

Yes, LLVM provides robust mechanisms to handle renaming external references within LLVM IR modules, primarily through its C++ API. This is a common requirement for tasks like linking different modules, avoiding name collisions, or preparing modules for specific optimization passes.

Here's a breakdown of how you can achieve this, focusing on the standard LLVM API and tools:

### Understanding LLVM IR Naming

In LLVM IR, global values (functions, global variables, aliases) have names. These names are crucial for linking. When you have external references, it means your module is calling a function defined elsewhere or using a global variable defined elsewhere. Similarly, if your module defines a function that other modules will call, that function's name is an external reference.

### Mechanisms for Renaming

The most reliable way to perform these renames is by writing a small LLVM pass or an independent tool using the LLVM C++ API.

```
#### 1. Renaming Global Values

The ‘GlobalValue‘ class (which ‘Function‘, ‘GlobalVariable‘, and ‘
   ↪ GlobalAlias‘ inherit from) has a ‘setName()‘ method. This is your
   ↪  primary tool for renaming top-level entities.

**Example (Conceptual):**

‘‘‘c++
#include "llvm/IR/Function.h"
#include "llvm/IR/GlobalVariable.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/Support/raw_ostream.h"

// Assuming you have a Module* M
void renameExternalReferences(llvm::Module* M,
                              const std::map<std::string, std::string>&
   ↪  renames) {
    // Rename Functions
    for (auto& F : M->functions()) {
        auto it = renames.find(F.getName().str());
        if (it != renames.end()) {
            F.setName(it->second);
        }
    }

    // Rename Global Variables
    for (auto& GV : M->globals()) {
        auto it = renames.find(GV.getName().str());
        if (it != renames.end()) {
            GV.setName(it->second);
        }
    }

    // You might also need to handle Aliases if they are relevant to
   ↪ your external references
    // for (auto& GA : M->aliases()) {
    //     auto it = renames.find(GA.getName().str());
    //     if (it != renames.end()) {
    //         GA.setName(it->second);
    //     }
    // }
}

int main() {
    llvm::LLVMContext Context;
    // Load your IR module here, for example:
    // std::unique_ptr<llvm::Module> M =
    //     llvm::parseIRFile("input.ll", ErrorMessage, Context);

    // For demonstration, let’s create a simple module
    llvm::Module* M = new llvm::Module("my_module", Context);
    llvm::FunctionType* FT =
        llvm::FunctionType::get(llvm::Type::getVoidTy(Context), false);
    llvm::Function* FuncA =
        llvm::Function::Create(FT, llvm::Function::ExternalLinkage,
                               "original_func_A", M);
    llvm::Function* FuncB =
        llvm::Function::Create(FT, llvm::Function::ExternalLinkage,
                               "original_func_B", M);

    std::map<std::string, std::string> renames = {
        {"original_func_A", "renamed_func_X"},
        {"original_func_B", "renamed_func_Y"}
    };

    llvm::outs() << "Before renaming:\n";
    M->print(llvm::outs(), nullptr);

    renameExternalReferences(M, renames);

    llvm::outs() << "\nAfter renaming:\n";
    M->print(llvm::outs(), nullptr);

    delete M;
    return 0;
}
‘‘‘

**Explanation of ‘setName()‘:**

*   When you call ‘setName()‘ on a ‘GlobalValue‘, LLVM automatically
   ↪ handles all internal references to that global value within the
   ↪ same module. For instance, if you rename a function ‘foo‘, all ‘
   ↪ call‘ instructions within that module that refer to ‘foo‘ will
   ↪ automatically be updated to refer to the new name. This is
   ↪ crucial as it prevents broken links within your module.

#### 2. Handling Linkage Types

When dealing with "external references," linkage types are important.
```

```
*   ‘ExternalLinkage‘: This is the default for globally visible
   ↪ functions and variables. They can be referenced by other modules
   ↪ and can resolve to definitions in other modules.
*   ‘ExternalWeakLinkage‘: Similar to ‘ExternalLinkage‘ but handles
   ↪ weak symbols (e.g., C++ template instantiations).
*   ‘PrivateLinkage‘, ‘InternalLinkage‘, ‘LinkOnceODRLinkage‘, etc.:
   ↪ These are for symbols not intended to be visible or used outside
   ↪ the current translation unit. You generally wouldn’t rename these
   ↪  for external linking purposes.

Your focus should primarily be on ‘ExternalLinkage‘ and ‘
   ↪ ExternalWeakLinkage‘ global values if you want to modify how your
   ↪  module interacts with others.

#### 3. Reading and Writing LLVM IR

You’ll need to load your ‘.ll‘ or ‘.bc‘ files and then write the
   ↪ modified module back.

*   **Reading:**
    *   ‘llvm::parseIRFile‘: For ‘.ll‘ (textual IR).
    *   ‘llvm::get:{MemBufferRef}‘ and ‘llvm::parseBitcodeFile‘: For ‘.
   ↪ bc‘ (binary IR).
*   **Writing:**
    *   ‘Module::print()‘: To write back to ‘.ll‘.
    *   ‘WriteBitcodeToFile()‘: To write back to ‘.bc‘.

**Example of loading/saving (within a tool):**

‘‘‘c++
#include "llvm/IR/Module.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IRReader/IRReader.h" // For parseIRFile
#include "llvm/Bitcode/BitcodeWriter.h" // For WriteBitcodeToFile
#include "llvm/Bitcode/BitcodeReader.h" // For parseBitcodeFile
#include "llvm/Support/SourceMgr.h"
#include "llvm/Support/MemoryBuffer.h"
#include "llvm/Support/Error.h"
#include "llvm/Support/raw_ostream.h"

// ... (your renameExternalReferences function)

int main(int argc, char** argv) {
    if (argc != 3) {
        llvm::errs() << "Usage: " << argv[0] << " <input.ll/.bc> <
   ↪ output.ll/.bc>\n";
        return 1;
    }

    llvm::LLVMContext Context;
    llvm::SMDiagnostic Err;
    std::unique_ptr<llvm::Module> M;

    // Determine if input is .ll or .bc
    std::string inputFile = argv[1];
    if (inputFile.size() > 3 && inputFile.substr(inputFile.size() - 3)
    ↪ == ".ll") {
        M = llvm::parseIRFile(inputFile, Err, Context);
    } else if (inputFile.size() > 3 && inputFile.substr(inputFile.size
    ↪ () - 3) == ".bc") {
        llvm::Expected<std::unique_ptr<llvm::MemoryBuffer>> MemBuf =
            llvm::MemoryBuffer::getFileOrSTDIN(inputFile);
        if (!MemBuf) {
            llvm::errs() << "Error reading input file: " << MemBuf.
    ↪ takeError() << "\n";
            return 1;
        }
        llvm::Expected<std::unique_ptr<llvm::Module>> ParsedModule =
            llvm::parseBitcodeFile(MemBuf.get()->getMemBufferRef(),
    ↪ Context);
        if (!ParsedModule) {
            llvm::errs() << "Error parsing bitcode: " << ParsedModule.
    ↪ takeError() << "\n";
            return 1;
        }
        M = std::move(*ParsedModule);
    } else {
        llvm::errs() << "Input file must be .ll or .bc\n";
        return 1;
    }

    if (!M) {
        Err.print(argv[0], llvm::errs());
        return 1;
    }

    // Define your renames (this would come from your build process)
    std::map<std::string, std::string> renames = {
        {"_Z7my_funcv", "my_scoped_namespace_my_func"}, // Example for
    ↪ a C++ mangled name
        {"external_data", "scoped_external_data"}
    };

    renameExternalReferences(M.get(), renames);
```

```
    // Write the modified module
    std::string outputFile = argv[2];
    std::error_code EC;
    llvm::raw_fd_ostream OS(outputFile, EC, llvm::sys::fs::OF_None);
    if (EC) {
        llvm::errs() << "Error opening output file: " << EC.message()
   ↪ << "\n";
        return 1;
    }

    if (outputFile.size() > 3 && outputFile.substr(outputFile.size() -
   ↪ 3) == ".ll") {
        M->print(OS, nullptr);
    } else if (outputFile.size() > 3 && outputFile.substr(outputFile.
   ↪ size() - 3) == ".bc") {
        llvm::WriteBitcodeToFile(*M, OS);
    } else {
        llvm::errs() << "Output file must be .ll or .bc\n";
        return 1;
    }

    return 0;
}
```

#### 4. Integration with Your Build Process

You mentioned having access to the desired name mappings. You can pass
   ↪ these mappings to your C++ tool in various ways:

*   **Command-line arguments:** For simple cases, key-value pairs.
*   **Configuration file (JSON, YAML):** For more complex mappings,
   ↪ read a file.
*   **Environment variables:** Less common for mappings, but possible.
*   **Shared memory/IPC:** If your build system can set this up.

### When to Perform the Renaming

The best time to perform this renaming is *before* the final linking
   ↪ stage, but *after* individual modules have been generated and
   ↪ potentially optimized. This ensures:

1. Your renaming tool can operate on individual LLVM IR modules (`.ll`
   ↪ or `.bc`).
2. The modified modules will then link cleanly with others, as their "
   ↪ scoped" names will be unique.
```
```

```
### Caveats and Considerations

*   **Mangled Names (C++):** If your source language is C++, the names
  ↪ of functions and global variables will be "mangled." You'll need
  ↪ to work with these mangled names. Tools like `llvm-cxxfilt` can
  ↪ help you demangle names for inspection, but your renaming logic
  ↪ needs to operate on the mangled form. For example, `void my_func
  ↪ ()` might become `_Z7my_funcv`.
*   **Linkage Errors:** If you try to rename an external reference to a
  ↪  name that already exists (or will exist after another module is
  ↪ renamed), you will still get a linker error. Your mapping
  ↪ generation logic in the build system needs to guarantee
  ↪ uniqueness for the new "scoped" names.
*   **Type Compatibility:** While `setName()` handles the textual/
  ↪ symbolic rename, it doesn't change the function signature or
  ↪ global variable type. When linking, LLVM's linker and the system
  ↪ linker will also check for type compatibility. Ensure your
  ↪ renames don't accidentally link incompatible types.
*   **Symbol Visibility:** Be mindful of symbol visibility (e.g., `
  ↪ default`, `hidden`, `protected`). While renaming, you're
  ↪ primarily changing the identifier, but the underlying visibility
  ↪ rules still apply during final linking.
*   **Optimization Passes:** Renaming should ideally happen *before*
  ↪ any inter-procedural optimization (IPO) passes that might rely on
  ↪  specific symbol names or could inline calls across modules, but
  ↪ *after* module-level optimizations. This ensures your names are
  ↪ stable when IPO runs and that the renamed symbols are correctly
  ↪ processed.
*   **Debug Information:** Renaming top-level symbols should ideally be
  ↪  transparent to debug information. LLVM typically stores debug
  ↪ info with references to the underlying `GlobalValue`, so changing
  ↪  the `GlobalValue`'s name will usually propagate correctly to the
  ↪  debug info if you preserve it.

By implementing a small utility using the LLVM C++ API, you can
  ↪ precisely control the renaming of external references in your
  ↪ LLVM IR files, effectively scoping your modules for robust
  ↪ linking.
```

Thread (6 messages)
ID: 6deac944-2edd-4644-9955-c8a780772eb2 **2025/11/16 10:02**

USER (Message 1) [gpt-5-chat] 2025/11/16 10:02

```
what do you call the code generation step in a compiler?
```

AGENT (Message 2) [gpt-5-chat] 2025/11/16 10:02

Its typically called the code generation phase (or simply codegen). In many compiler texts, its part of the back end, where the intermediate representation (IR) is translated into target code (machine code or assembly). Some also distinguish instruction selection, register allocation, and instruction scheduling as sub-steps within code generation.

USER (Message 3) [gpt-5-chat] 2025/11/16 10:03

what are all the phases. i want just two phases (a) parser (which includes lexer), and (b) codegen which includes like everything else. what should i call these two phases

AGENT (Message 4) [gpt-5-chat] 2025/11/16 10:03

Two clear, conventional names you can use:

- Front end: Parser (including the lexer)
- Back end: Codegen (encompassing IR, analysis, optimization, instruction selection, register allocation, scheduling, and emission)

If you want slightly more formal labels:
- Front End (Parsing)
- Back End (Code Generation)

USER (Message 5) [gpt-5-chat] 2025/11/16 10:03

what overall directory should i call the orchestration of these layers

AGENT (Message 6) [gpt-5-chat] 2025/11/16 10:03

A concise, common choice is pipeline.

Other solid options, depending on vibe:
- compiler
- driver
- orchestrator
- engine
- pipeline-runner
- toolchain
- workflow

If this is the entry point that wires front end and back end together, driver or compiler are the most conventional in compiler repos.

```
If its a set of passes executed in sequence, pipeline is clearest .
```

Thread (6 messages)
ID: a9183424-da00-4790-9489-d0f65cddffef **2025/11/16 10:20**

USER (Message 1) [gpt-5-chat] 2025/11/16 10:20

```
nodejs take in two cli arguments
```

AGENT (Message 2) [gpt-5-chat] 2025/11/16 10:20

```
Heres a minimal Node.js example to read two CLI arguments.

- If you run: node app.js hello world
- You want to capture "hello" and "world".

Option 1: Using process.argv directly
‘‘‘js
// app.js
// process.argv[0] = node executable
// process.argv[1] = script path
// process.argv[2+] = user args

const [arg1, arg2] = process.argv.slice(2);

if (!arg1 || !arg2) {
  console.error("Usage: node app.js <arg1> <arg2>");
  process.exit(1);
}

console.log("Arg1:", arg1);
console.log("Arg2:", arg2);
‘‘‘

Run:
‘‘‘bash
node app.js foo bar
‘‘‘

Option 2: With named flags (e.g., --from=foo --to=bar)
‘‘‘js
// app.js
const args = process.argv.slice(2);
let from = null;
let to = null;
```

```
for (const a of args) {
  if (a.startsWith("--from=")) from = a.split("=")[1];
  if (a.startsWith("--to=")) to = a.split("=")[1];
}

if (!from || !to) {
  console.error("Usage: node app.js --from=<value> --to=<value>");
  process.exit(1);
}

console.log({ from, to });
‘‘‘

Run:
‘‘‘bash
node app.js --from=alice --to=bob
‘‘‘

Option 3: Using a small library (yargs)
‘‘‘bash
npm install yargs
‘‘‘

‘‘‘js
// app.js
import yargs from "yargs";
import { hideBin } from "yargs/helpers";

const argv = yargs(hideBin(process.argv))
  .option("from", { type: "string", demandOption: true })
  .option("to", { type: "string", demandOption: true })
  .strict()
  .parse();

console.log(argv.from, argv.to);
‘‘‘

Run:
‘‘‘bash
node app.js --from=alice --to=bob
‘‘‘

Pick Option 1 if you just need two positional args and want zero deps.
```

USER (Message 3) [gpt-5-chat] 2025/11/16 10:20

```
are there any node packages that make nice cli
```

AGENT (Message 4) [gpt-5-chat] 2025/11/16 10:20

```
Yesthere are several solid Node.js packages for building nice CLIs. Here are the go-to options, with when to use each and a tiny example.

Top choices
- Commander: Mature, simple API for commands, options, subcommands.
  - When: Most CLIs with flags/subcommands.
  - Install: ‘npm i commander‘
  - Example:
    ‘‘‘js
    // cli.js
    import { Command } from "commander";
    const program = new Command();

    program
      .name("mycli")
      .description("Example CLI")
      .version("1.0.0")
      .option("-v, --verbose", "verbose output")
      .command("greet <name>")
      .option("-t, --times <n>", "repeat times", parseInt, 1)
      .action((name, opts, cmd) => {
        const times = cmd.opts().times;
        for (let i = 0; i < times; i++) {
          console.log(‘Hello, ${name}${cmd.parent?.opts().verbose ? "!" : ""}‘);
        }
      });

    program.parse();
    ‘‘‘
    Run: ‘node cli.js greet Alice -t 2 -v‘

- Yargs: Powerful parsing with subcommands, strict mode, types, config files.
  - When: Complex flag parsing; config + argv merging.
  - Install: ‘npm i yargs‘
  - Example:
    ‘‘‘js
    // cli.js
    import yargs from "yargs";
    import { hideBin } from "yargs/helpers";
    yargs(hideBin(process.argv))
      .scriptName("mycli")
      .command(
        "greet <name>",
        "greet someone",
        (y) =>
          y.positional("name", { type: "string" }).option("times", {
            alias: "t",
            type: "number",
            default: 1,
          }),
        (argv) => {
          for (let i = 0; i < argv.times; i++) console.log(‘Hello, ${
   ↪ argv.name}‘);
        }
      )
      .demandCommand()
      .strict()
      .help().argv;
    ‘‘‘

- OCLIF (by Heroku/Salesforce): Framework for large, plugin-able CLIs.
  - When: Multi-command CLIs you might distribute, with plugins and
   ↪ scaffolding.
  - Install: ‘npm i -g @oclif/command‘ or use generator ‘npm i -g @oclif
   ↪ /cli‘
  - Notes: Generates a project with commands in separate files, built-
   ↪ in help/version, autocomplete.

- Clipanion: Modern, TypeScript-first CLI parser (used by Yarn).
  - When: TS heavy projects; great DX and typings.
  - Install: ‘npm i clipanion‘
  - Example:
    ‘‘‘ts
    // cli.ts
    import { Command, Option, Usage, runExit } from "clipanion";

    class GreetCommand extends Command {
      static paths = [["greet"]];
      static usage: Usage = { description: "Greet someone" };

      name = Option.String();
      times = Option.Integer("--times", 1);

      async execute() {
        for (let i = 0; i < this.times; i++) console.log(‘Hello, ${this
   ↪ .name}‘);
   }
  }

  runExit([GreetCommand]);
  ‘‘‘

- Meow: Minimal, fast, great for single-file CLIs.
 - When: Small tools with a help string and a handful of flags.
 - Install: ‘npm i meow‘
 - Example:
  ‘‘‘js
  import meow from "meow";

  const cli = meow(
   ‘
   Usage
    $ mycli <input>

   Options
    --times, -t  Number of repetitions

   Examples
    $ mycli Alice --times=2
  ‘,
   {
    importMeta: import.meta,
    flags: { times: { type: "number", default: 1, alias: "t" } },
   }
  );

  const [name] = cli.input;
  for (let i = 0; i < cli.flags.times; i++) console.log(‘Hello, ${
 ↪ name}‘);
  ‘‘‘

UX niceties to add
- Prompts:
 - Inquirer: ‘npm i inquirer‘ (classic, feature-rich).
 - Enquirer: ‘npm i enquirer‘ (lighter, fast).
 - Prompts: ‘npm i prompts‘ (async/await friendly).
- Output styling:
 - Chalk: ‘npm i chalk‘ (colors/styles).
 - Colorette / Picocolors: smaller alternatives.
- Spinners and progress:
 - Ora: ‘npm i ora‘ (simple spinners).
 - Listr2: ‘npm i listr2‘ (task lists with concurrency).
- Tables:
  - Table: ‘npm i table‘
  - Cli-table3: ‘npm i cli-table3‘
- Terminal UI:
  - Ink: ‘npm i ink‘ (React for CLIs).
- Autocomplete:
  - Commander, yargs, and oclif have solutions; oclif has built-in
    ↪ plugin.

Quick recommendation
- Small tool: Meow + Chalk + Prompts + Ora.
- Medium CLI with subcommands: Commander or Yargs (+ Ora/Chalk).
- Large/enterprise CLI: OCLIF or Clipanion (TS).

If you share your CLIs size and needs (subcommands, config files,
    ↪ interactive prompts, distribution), I can suggest a tailored
    ↪ stack and starter template.
```

USER (Message 5) [gpt-5-chat] 2025/11/16 10:21

```
ok so what if i want a compile command that takes in two arguments <src
    ↪ > and <dest>.
```

AGENT (Message 6) [gpt-5-chat] 2025/11/16 10:21

```
Here are two clean ways to implement a compile command with positional
    ↪ args <src> and <dest>.

Option 1: Commander (simple, readable)
‘‘‘js
// cli.js
import { Command } from "commander";

const program = new Command();

program
  .name("mycli")
  .description("My CLI")
  .version("1.0.0");

program
  .command("compile <src> <dest>")
  .description("Compile from <src> to <dest>")
  .option("-w, --watch", "watch for changes")
  .action(async (src, dest, options) => {
    console.log(‘Compiling from ${src} to ${dest}...‘);
    if (options.watch) {
      console.log("Watching for changes...");
      // your watch logic here
    }
    // your compile logic here
  });

program.parse();
‘‘‘

Run:
‘‘‘bash
node cli.js compile src/file.txt dist/file.out
node cli.js compile src dir/out --watch
‘‘‘

Option 2: Yargs (great parsing + help)
‘‘‘js
// cli.js
import yargs from "yargs";
import { hideBin } from "yargs/helpers";

yargs(hideBin(process.argv))
  .scriptName("mycli")
  .command(
    "compile <src> <dest>",
    "Compile from <src> to <dest>",
    (y) =>
      y
        .positional("src", { type: "string", describe: "Source path" })
        .positional("dest", { type: "string", describe: "Destination path" })
        .option("watch", {
          alias: "w",
          type: "boolean",
          describe: "Watch for changes",
          default: false,
        }),
    async (argv) => {
      console.log(‘Compiling from ${argv.src} to ${argv.dest}...‘);
      if (argv.watch) console.log("Watching for changes...");
      // your compile logic here
    }
  )
  .demandCommand(1)
  .strict()
  .help().argv;
‘‘‘
```

```
Run:
‘‘‘bash
node cli.js compile src/file.txt dist/file.out
node cli.js compile src dir/out -w
‘‘‘

Notes
- Both examples use positional args for <src> and <dest>.
- Add validation easily (e.g., check file existence) before running the compile.
- If you need ESM support, ensure "type": "module" is set in package.json, or switch to CommonJS ‘require(...)‘ style.
```

Thread (16 messages)
ID: f97198a6-f399-4912-a023-8740dc2583dd **2025/11/16 13:04**

USER (Message 1) [gpt-5-chat] 2025/11/16 13:04

```
how to setup a c++ project that will be using the llvm library for emitting some delicious llvm ir
```

AGENT (Message 2) [gpt-5-chat] 2025/11/16 13:04

```
Great goal. Heres a practical, modern setup to build a C++ project that emits LLVM IR using the LLVM libraries. Ill show:
- Recommended project layout
- CMake configuration using LLVMs CMake package
- A minimal C++ example that builds an LLVM module and prints IR
- How to build and run
- Notes for common platforms and pitfalls

Project structure
- llvm-ir-demo/
  - CMakeLists.txt
  - src/
    - main.cpp

CMake configuration (finds LLVM and links the right components)
‘‘‘cmake
# CMakeLists.txt
cmake_minimum_required(VERSION 3.20)
project(llvm_ir_demo LANGUAGES CXX)

# Require a reasonably recent LLVM; adjust version as needed
# You can also omit the version to accept any found LLVM
find_package(LLVM 15.0 REQUIRED CONFIG)
```

```
message(STATUS "Found LLVM ${LLVM_PACKAGE_VERSION}")
message(STATUS "LLVM include dirs: ${LLVM_INCLUDE_DIRS}")
message(STATUS "LLVM defs: ${LLVM_DEFINITIONS}")
message(STATUS "Using LLVM at: ${LLVM_DIR}")

# Set C++ standard
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Use LLVMs CMake utilities
list(APPEND CMAKE_MODULE_PATH "${LLVM_CMAKE_DIR}")
include(AddLLVM)

# Recommended compile defs and include directories
add_definitions(${LLVM_DEFINITIONS})
include_directories(${LLVM_INCLUDE_DIRS})

# Choose the LLVM components you need.
# For emitting/printing IR you typically need:
#   - Core (Module, Function, IRBuilder)
#   - Support (Command-line, raw_ostream)
#   - IRReader (if parsing IR from text)
#   - AsmPrinter/Target components if you later emit assembly/object
set(LLVM_COMPONENTS
  Core
  Support
  IRReader
)

llvm_map_components_to_libnames(LLVM_LIBS ${LLVM_COMPONENTS})

add_executable(llvm_ir_demo src/main.cpp)
target_link_libraries(llvm_ir_demo PRIVATE ${LLVM_LIBS})

# On Windows with static LLVM builds, you may need this:
if(MSVC)
  target_compile_definitions(llvm_ir_demo PRIVATE
   ↪ _SILENCE_ALL_CXX17_DEPRECATION_WARNINGS)
endif()
‘‘‘

Minimal C++ example that creates and prints IR
‘‘‘cpp
// src/main.cpp
#include <llvm/IR/IRBuilder.h>
#include <llvm/IR/LLVMContext.h>
```

```
#include <llvm/IR/Module.h>
#include <llvm/IR/Verifier.h>
#include <llvm/Support/raw_ostream.h>

int main() {
  // Create core LLVM data structures
  llvm::LLVMContext context;
  auto module = std::make_unique<llvm::Module>("demo_module", context);
  llvm::IRBuilder<> builder(context);

  // Create function: int add(int a, int b)
  auto intTy = llvm::Type::getInt32Ty(context);
  auto funcTy = llvm::FunctionType::get(intTy, {intTy, intTy}, /*
   ↪ isVarArg=*/false);
  auto func = llvm::Function::Create(
      funcTy, llvm::Function::ExternalLinkage, "add", module.get());

  // Name the function arguments
  auto argIter = func->arg_begin();
  llvm::Argument *a = argIter++;
  a->setName("a");
  llvm::Argument *b = argIter++;
  b->setName("b");

  // Create a basic block and emit IR for return a + b;
  auto entryBB = llvm::BasicBlock::Create(context, "entry", func);
  builder.SetInsertPoint(entryBB);
  llvm::Value *sum = builder.CreateAdd(a, b, "sum");
  builder.CreateRet(sum);

  // Validate the IR
  if (llvm::verifyModule(*module, &llvm::errs())) {
    llvm::errs() << "Error: module verification failed\n";
    return 1;
  }

  // Print IR to stdout
  module->print(llvm::outs(), nullptr);
  return 0;
}
‘‘‘

Build and run
- Prerequisites:
  - Install LLVM with development files (headers + libs)
    - Linux (Ubuntu example): ‘sudo apt-get install llvm-17-dev clang
   ↪ -17‘ (match your version)
```

```
    - macOS (Homebrew): ‘brew install llvm‘ and note the keg path (e.g
  ↪ ., /opt/homebrew/opt/llvm)
    - Windows:
      - Use LLVM prebuilt package or build from source with Visual
  ↪ Studio and CMake.
      - Ensure LLVM_DIR points to the cmake directory of your build/
  ↪ install.

- Configure and build:
‘‘‘bash
# From project root
cmake -S . -B build -DLLVM_DIR=/path/to/llvm/lib/cmake/llvm
cmake --build build --config Release
‘‘‘

Notes:
- On macOS with Homebrew, LLVM_DIR is typically:
  - Apple Silicon: /opt/homebrew/opt/llvm/lib/cmake/llvm
  - Intel: /usr/local/opt/llvm/lib/cmake/llvm
- On many Linux distros with llvm-X-dev packages, LLVMs CMake config is
  ↪  auto-discovered. If not, locate with:
  - ‘llvm-config --cmakedir‘

- Run:
‘‘‘bash
./build/llvm_ir_demo
‘‘‘
Expected output IR:
‘‘‘text
; ModuleID = ’demo_module’
source_filename = "demo_module"

define i32 @add(i32 %a, i32 %b) {
entry:
  %sum = add i32 %a, %b
  ret i32 %sum
}
‘‘‘

Extending to emit files or target code
- Write IR to a .ll file:
  - Replace ‘module->print(llvm::outs(), nullptr);‘ with:
    ‘‘‘cpp
    #include <llvm/Support/FileSystem.h>
    #include <llvm/Support/ToolOutputFile.h>

    std::error_code ec;
```

~~~
    llvm::raw_fd_ostream os("out.ll", ec, llvm::sys::fs::OF_Text);
    if (ec) { llvm::errs() << "error: " << ec.message() << "\n"; return 1; }
    module->print(os, nullptr);
    ```
- Emit bitcode (.bc):
  ```cpp
  #include <llvm/Bitcode/BitcodeWriter.h>
  std::error_code ec;
  llvm::raw_fd_ostream bc("out.bc", ec, llvm::sys::fs::OF_None);
  if (ec) { llvm::errs() << "error: " << ec.message() << "\n"; return 1; }
  llvm::WriteBitcodeToFile(*module, bc);
  ```
- Generate object code or assembly:
  - Youll need Target, MC, AsmPrinter, etc. components and target initialization. Add components to CMake:
    - X86 example: `set(LLVM_COMPONENTS Core Support Target MC MCParser AsmPrinter X86CodeGen X86AsmParser X86Desc X86Info)`
  - Initialize targets in code:
    ```cpp
    #include <llvm/Support/Host.h>
    #include <llvm/Target/TargetMachine.h>
    #include <llvm/Target/TargetOptions.h>
    #include <llvm/Target/TargetRegistry.h>
    #include <llvm/IR/LegacyPassManager.h>
    #include <llvm/MC/TargetRegistry.h>

    using namespace llvm;

    auto triple = sys::getDefaultTargetTriple();
    module->setTargetTriple(triple);

    std::string error;
    auto target = TargetRegistry::lookupTarget(triple, error);
    if (!target) { errs() << error << "\n"; return 1; }

    TargetOptions opt;
    auto tm = std::unique_ptr<TargetMachine>(
      target->createTargetMachine(triple, "generic", "", opt, std::nullopt));

    module->setDataLayout(tm->createDataLayout());

    legacy::PassManager pm;
    std::error_code ec;
    raw_fd_ostream dest("out.o", ec, sys::fs::OF_None);
~~~

```
    if (ec) { errs() << "error: " << ec.message() << "\n"; return 1; }

    if (tm->addPassesToEmitFile(pm, dest, nullptr, CGFT_ObjectFile)) {
      errs() << "TargetMachine can’t emit a file of this type\n";
   ↪ return 1;
    }
    pm.run(*module);
    dest.flush();
    ‘‘‘
  - Then link with your system linker or clang: ‘clang out.o -o program
   ↪ ‘

Common pitfalls and tips
- Match LLVM versions: Headers and libs must come from the same
   ↪ installation.
- Use LLVMs CMake config (find_package) instead of rolling your own ‘
   ↪ llvm-config --libs‘ calls; its more portable.
- ABI and C++ runtime: Prefer the same compiler toolchain used to build
   ↪  LLVM.
- On Windows with MSVC, prefer linking against the same CRT settings (
   ↪ MD/MT) as LLVM was built with.
- If you get missing symbols at link time, ensure you included all
   ↪ required components via llvm_map_components_to_libnames.

Want me to tailor this for your OS, your preferred LLVM version, or add
   ↪  a starter template repo with a CI workflow?
```

USER (Message 3) [gpt-5-chat] 2025/11/16 13:06

```
do i need cmake? i am familiar with hand-writing makefiles... can you
   ↪ explain how cmakelists works and if that’s something i should do,
   ↪  or whether it is possible to hand craft a Makefile?
```

AGENT (Message 4) [gpt-5-chat] 2025/11/16 13:06

```
Short answer
- You do not need CMake. You can absolutely handcraft a Makefile and
   ↪ link against LLVM.
- CMake is strongly recommended for LLVM projects because it:
  - Finds the correct include/lib paths and components across OSes and
   ↪ versions
  - Handles long/fragile link lines for you
  - Makes cross-platform builds and IDEs easier
But if youre comfortable with Make, you can use ‘llvm-config‘ to
   ↪ generate the right flags and keep it portable.

How CMakeLists.txt works (at a glance)
```

```
- find_package(LLVM ... CONFIG): Loads LLVMs CMake package, which
    ↪ exposes variables like LLVM_INCLUDE_DIRS, LLVM_DEFINITIONS,
    ↪ LLVM_CMAKE_DIR, LLVM_PACKAGE_VERSION.
- list(APPEND CMAKE_MODULE_PATH ...)/include(AddLLVM): Imports LLVM-
    ↪ provided CMake helpers.
- llvm_map_components_to_libnames(LLVM_LIBS Core Support IRReader ...):
    ↪  Translates logical components into actual library names for your
    ↪  platform/build (static vs shared, suffixed names, etc.).
- target_link_libraries(llvm_ir_demo PRIVATE ${LLVM_LIBS}): Links the
    ↪ resolved libraries.
- include_directories/add_definitions: Applies LLVMs required include
    ↪ paths and compile defs to your target.
In practice, you specify which LLVM components you need (Core, Support,
    ↪  IRReader, Target, etc.), and CMake resolves the actual library
    ↪ list and paths. This saves you from maintaining long, version-
    ↪ specific link lines.

Hand-written Makefile approach
Use ‘llvm-config‘ (installed with LLVM) to ask LLVM for the correct
    ↪ flags. This keeps your Makefile portable across versions and
    ↪ distros.

Example Makefile (single binary, minimal IR printing)
‘‘‘make
# Makefile
# Adjust these if you want a specific llvm toolchain (e.g., clang-17)
CXX := clang++
CXXFLAGS := -std=c++17 -O2 -fno-exceptions -fno-rtti
# If you need exceptions/RTTI, drop those flags or set according to
    ↪ your needs.

# Query llvm-config for flags and libs. Choose the components you need.
LLVM_COMPONENTS := core support irreader
LLVM_CXXFLAGS := $(shell llvm-config --cxxflags)
LLVM_LDFLAGS  := $(shell llvm-config --ldflags)
LLVM_LIBS     := $(shell llvm-config --libs $(LLVM_COMPONENTS))
# If you later emit object/asm, you'll add target components here, e.g.
    ↪  x86codegen.

# Some platforms require --system-libs to pull in zlib, libxml2, etc.
SYSTEM_LIBS   := $(shell llvm-config --system-libs)

SRC := src/main.cpp
BIN := build/llvm_ir_demo

# Default target
all: $(BIN)
```

```
# Ensure build directory exists
$(BIN): $(SRC) | build
	$(CXX) $(CXXFLAGS) $(LLVM_CXXFLAGS) -o $@ $^ $(LLVM_LDFLAGS) $(
   ↪ LLVM_LIBS) $(SYSTEM_LIBS)

build:
	mkdir -p build

run: $(BIN)
	./$(BIN)

clean:
	rm -rf build

.PHONY: all run clean
‘‘‘

Notes on the Makefile
- llvm-config --cxxflags/--ldflags/--libs/--system-libs are the key to
   ↪ portability. They encode the right include paths, defines, link
   ↪ directories, and the long library list that varies by platform
   ↪ and static/shared builds.
- Components:
  - For basic IR building/printing: core support irreader (irreader
   ↪ only if you parse .ll text).
  - For codegen to object/asm: add target, mc, mcparser, asmprinter and
   ↪  the architecture family (e.g., x86codegen x86asmparser x86desc
   ↪ x86info). Then re-run ‘llvm-config --libs ...‘.
- Compilers: Prefer the same compiler family LLVM was built with. If
   ↪ LLVM was built with Clang and libc++, matching that reduces ABI
   ↪ issues. If you see link errors, try building with ‘clang++‘ and
   ↪ add ‘llvm-config --cxxflags‘ first in CXXFLAGS to let LLVM set C
   ↪ ++ stdlib/defines.

When to pick CMake vs Make
- Choose CMake if:
  - Cross-platform (Linux/macOS/Windows) or you want IDE generators (
   ↪ Visual Studio, Xcode).
  - You foresee adding multiple executables/libraries, tests, install
   ↪ rules, or optional features.
  - You want the build to just work on machines with different LLVM
   ↪ versions/paths without editing your Makefile.
- Choose Make if:
  - Youre comfortable with Make and want a small, transparent build.
  - Single or few binaries, and you control the environment (e.g., same
   ↪  LLVM version).
  - Youre fine calling ‘llvm-config‘ for flags and updating components
    ↪ manually.

Equivalent CMake and Make mental mapping
- CMake find_package(LLVM ...) ~ Makes use of ‘llvm-config‘ to discover
    ↪  flags/paths.
- llvm_map_components_to_libnames(...) ~ ‘llvm-config --libs <
    ↪ components>‘ selects actual libs.
- target_link_libraries(...) ~ passing $(LLVM_LIBS) $(SYSTEM_LIBS) on
    ↪ your link line.
- include_directories/add_definitions ~ using ‘llvm-config --cxxflags‘
    ↪ in your compile flags.

Common pitfalls (both approaches)
- Mismatched LLVM version: headers/libs must come from the same install
    ↪ .
- Static vs shared LLVM: Static builds require many more system libs;
    ↪ ‘--system-libs‘ helps.
- RTTI/exceptions: Many LLVM libs are built without RTTI/exceptions. If
    ↪  you need them (e.g., for dynamic_cast across library boundaries)
    ↪ , ensure your flags and linking are consistent, or enable RTTI in
    ↪  LLVM when building from source.
- Windows: Prefer CMake unless youre very comfortable; library names
    ↪ and CRT settings can be tricky. If you do Make on Windows, use
    ↪ MSYS2/MinGW or clang-cl with care.

If you want, tell me your OS and LLVM version and whether you plan to
    ↪ emit object code, and Ill tailor a Makefile with the exact
    ↪ components and flags.
```

USER (Message 5) [gpt-5-chat] 2025/11/16 13:12

```
im gonna be building in WSL (or Docker down the road if it makes it
    ↪ easy to resolve dependencies). i assume LLVM supports cross-
    ↪ compilation and that i can stick with windows? if not i would be
    ↪ rather disheartened. does cmake replace my traditional makefile,
    ↪ or can it go alongside it. because i really want to know how to
    ↪ add some traditional build rules to my project, where i choose a
    ↪ file pattern and decide on the commands that build such files.
```

AGENT (Message 6) [gpt-5-chat] 2025/11/16 13:12

```
You can stay on Windows and be happy:

- WSL: Great choice. Youll be building in a Linux environment with
    ↪ native LLVM packages, but editing/running from Windows is fine.
    ↪ This is the smoothest path.
```

```
- Docker: Also fine; good for reproducible builds. You can base off
    ↪ ubuntu:24.04 and apt-get llvm-17-dev, clang, etc.
- Native Windows: LLVM works, but dependency/linking quirks are more
    ↪ common. If you stay on Windows-native, prefer CMake.

LLVM and cross-compilation
- LLVM supports cross-compiling. You can generate IR, bitcode, or
    ↪ object code for other targets by setting the module triple and
    ↪ selecting a TargetMachine (e.g., x86_64-pc-windows-gnu, aarch64-
    ↪ unknown-linux-gnu, wasm32-unknown-unknown). You still need the
    ↪ right target components and, if you want final executables, a
    ↪ linker and sysroot for that target.
- In WSL or Docker, cross-compiling to Windows PE/COFF is possible
    ↪ using lld-link or MinGW-w64; conversely you can target Linux,
    ↪ macOS (limited), WASM, etc.

Does CMake replace Makefiles? How they relate
- CMake is a meta-build tool:
  - You write CMakeLists.txt (declarative build description).
  - CMake generates buildsystem files for your chosen generator:
    ↪ Makefiles (Unix Makefiles), Ninja files, Visual Studio solutions,
    ↪  etc.
  - So, yes, CMake can replace your hand-written Makefile by generating
    ↪  one. But it can also generate Ninja or MSBuild files instead.
- Can CMake coexist with a manual Makefile? Yes, but they are separate
    ↪ build systems:
  - You can keep a hand-written Makefile at the repo root for custom
    ↪ tasks, while using a build/ directory for the CMake-generated
    ↪ build.
  - Alternatively, use CMakes custom commands/targets to express your
    ↪ pattern rules so a single system runs everything.

If you want traditional pattern rules
Option A: Stay with a hand-written Makefile using llvm-config
- Youll control pattern rules exactly as you like. Example with object
    ↪ pattern and link:

‘‘‘make
# Makefile (WSL/Docker-friendly)
CXX := clang++
CXXFLAGS := -std=c++20 -O2 -Wall -Wextra

# Ask LLVM for the correct flags/libs. Add components as needed.
LLVM_COMPONENTS := core support irreader
LLVM_CXXFLAGS := $(shell llvm-config --cxxflags)
LLVM_LDFLAGS  := $(shell llvm-config --ldflags)
LLVM_LIBS     := $(shell llvm-config --libs $(LLVM_COMPONENTS))
```

```
SYSTEM_LIBS   := $(shell llvm-config --system-libs)

SRC_DIR := src
BUILD_DIR := build
SRCS := $(wildcard $(SRC_DIR)/*.cpp)
OBJS := $(patsubst $(SRC_DIR)/%.cpp,$(BUILD_DIR)/%.o,$(SRCS))
BIN := $(BUILD_DIR)/llvm_ir_demo

# Pattern rule for compiling .cpp -> .o
$(BUILD_DIR)/%.o: $(SRC_DIR)/%.cpp | $(BUILD_DIR)
    $(CXX) $(CXXFLAGS) $(LLVM_CXXFLAGS) -c $< -o $@

# Link rule
$(BIN): $(OBJS)
    $(CXX) -o $@ $^ $(LLVM_LDFLAGS) $(LLVM_LIBS) $(SYSTEM_LIBS)

# Phonies
all: $(BIN)
run: $(BIN)
    ./$(BIN)
clean:
    rm -rf $(BUILD_DIR)
$(BUILD_DIR):
    mkdir -p $(BUILD_DIR)

.PHONY: all run clean
```

- You can add more pattern rules (e.g., generate .ll from .cpp via
   ↪ clang -S -emit-llvm, or run a custom IR pass).

Option B: Use CMake but keep Make-like pattern behavior via custom
   ↪ commands
- CMake isnt pattern-rule oriented, but you can emulate it with file(
   ↪ GLOB) or explicit lists and add_custom_command/add_custom_target.

Example: auto-generate .ll from each .cpp, plus your main target
```cmake
cmake_minimum_required(VERSION 3.20)
project(demo LANGUAGES CXX)
find_package(LLVM 15 REQUIRED CONFIG)
set(CMAKE_CXX_STANDARD 20)
list(APPEND CMAKE_MODULE_PATH "${LLVM_CMAKE_DIR}")
include(AddLLVM)
add_definitions(${LLVM_DEFINITIONS})
include_directories(${LLVM_INCLUDE_DIRS})
```

```
set(LLVM_COMPONENTS Core Support IRReader)
llvm_map_components_to_libnames(LLVM_LIBS ${LLVM_COMPONENTS})

file(GLOB SRC_FILES "${CMAKE_SOURCE_DIR}/src/*.cpp")
add_executable(llvm_ir_demo ${SRC_FILES})
target_link_libraries(llvm_ir_demo PRIVATE ${LLVM_LIBS})

# Pattern-like rule: for each .cpp, also produce a .ll using clang
find_program(CLANGPP NAMES clang++ clang++)
if(CLANGPP)
  set(GENERATED_LL)
  foreach(src ${SRC_FILES})
    get_filename_component(name ${src} NAME_WE)
    set(ll "${CMAKE_BINARY_DIR}/${name}.ll")
    add_custom_command(
      OUTPUT ${ll}
      COMMAND ${CLANGPP} -S -emit-llvm -std=c++20 -O0 -g -c ${src} -o ${ll}
      DEPENDS ${src}
      COMMENT "Generating LLVM IR: ${ll}"
      VERBATIM
    )
    list(APPEND GENERATED_LL ${ll})
  endforeach()

  add_custom_target(ir ALL DEPENDS ${GENERATED_LL})
  # Optional: make the main build depend on IR side products
  add_dependencies(llvm_ir_demo ir)
endif()
‘‘‘

- This gives you repeatable pattern-like behavior in CMake while retaining portability and LLVM discovery.

Option C: Hybrid
- Keep a simple CMakeLists.txt for the main binary and cross-platform LLVM discovery.
- Keep a top-level Makefile that calls cmake and also provides your custom pattern utilities:

Example top-level Makefile driving CMake and also offering custom commands
‘‘‘make
# Top-level Makefile that orchestrates CMake and adds extra rules
BUILD_DIR := build
GENERATOR := Ninja # or "Unix Makefiles"
CONFIG := Release
```

```
all: configure build

configure:
    cmake -S . -B $(BUILD_DIR) -G "$(GENERATOR)"

build:
    cmake --build $(BUILD_DIR) --config $(CONFIG)

run: build
    $(BUILD_DIR)/llvm_ir_demo

# Custom pattern-like utility: turn any .cpp into .ll with clang
   ↪ directly
%.ll: %.cpp
    clang++ -S -emit-llvm -std=c++20 -O0 -g -c $< -o $@

clean:
    rm -rf $(BUILD_DIR)

.PHONY: all configure build run clean
‘‘‘

WSL/Docker specifics
- WSL:
  - Install toolchain: sudo apt-get update && sudo apt-get install -y
   ↪ llvm-17-dev clang-17 cmake ninja-build
  - Use llvm-config-17 if your distro suffixes the version: set
   ↪ LLVM_CONFIG := llvm-config-17 in your Makefile.
  - File I/O across Windows <-> WSL is fine; keep the build tree inside
   ↪  the Linux filesystem (/home/) for performance.
- Docker:
  - Minimal Dockerfile:
    ‘‘‘dockerfile
    FROM ubuntu:24.04
    RUN apt-get update && apt-get install -y \
        build-essential cmake ninja-build curl git \
        llvm-17-dev clang-17 lld-17 \
      && ln -sf /usr/bin/clang-17 /usr/bin/clang++ \
      && ln -sf /usr/bin/clang-17 /usr/bin/clang
    WORKDIR /app
    COPY . .
    RUN cmake -S . -B build -G Ninja -DLLVM_DIR=$(llvm-config --
   ↪ cmakedir)
    RUN cmake --build build --config Release
    ‘‘‘
  - For Makefile-only builds, skip CMake lines and just ‘make‘.
```

```
Cross-target examples in your code
- To emit object code for x86_64 Linux while running in WSL:
  - Add LLVM components: target mc mcparser asmprinter x86codegen
   ↪ x86asmparser x86desc x86info
  - Set triple to x86_64-pc-linux-gnu and use TargetMachine to emit an
   ↪ object file. Link with clang.
- To target Windows PE from WSL:
  - Install MinGW-w64 or use lld-link.
  - Use triple x86_64-w64-mingw32 and ensure you have the MinGW sysroot
   ↪ . Link with ‘x86_64-w64-mingw32-g++‘ or ‘lld-link‘ plus proper
   ↪ CRT libs.

Recommendation
- If you like Make and want explicit pattern rules, use a hand-written
   ↪ Makefile that shells out to ‘llvm-config‘. Its perfectly viable
   ↪ in WSL/Docker and very transparent.
- If you want portability and painless library/component resolution
   ↪ across machines, use CMake and express your pattern rules via
   ↪ add_custom_command/target. You can still add a small top-level
   ↪ Makefile to drive CMake and host ad-hoc patterns.

If you tell me:
- your preferred LLVM version in WSL,
- whether you plan to emit object code for x86_64 Linux or Windows PE
   ↪ from WSL,
I can provide a tailored Makefile or CMakeLists with the exact
   ↪ component list and a sample rule to emit .o/.ll/.bc and link.
```

USER (Message 7) [gpt-5-chat] 2025/11/16 13:58

```
I previously used this Dockerfile in a Rust project. How can I adapt it
   ↪  for this C++ scenario? It should either delegate to a build.sh
   ↪ script or call a Makefile phony target, whatever you think is
   ↪ more appropriate.

# Build stage for RaptorQ CLI binary
FROM rust:1.86-slim

RUN apt-get update

# Install system dependencies for cross-compilation
RUN apt-get install -y \
    build-essential \
    curl \
    pkg-config \
    clang \
    llvm \
    lld \
    # Install GCC cross-compiler for ARM64 to get system libraries
    gcc-aarch64-linux-gnu \
    libc6-dev-arm64-cross \
    # Install MinGW cross-compiler for Windows
    gcc-mingw-w64-x86-64 \
    && rm -rf /var/lib/apt/lists/*

# Set up the working directory
WORKDIR /app

# Copy the entire internal directory
COPY . .

# Create output directory for volume mounting
RUN mkdir -p /volume/bin

# Set the output directory for the build script
ENV OUTPUT_DIR="/app/bin"

# Run the build script to build all targets
RUN chmod +x build.sh && ./build.sh

# Set the volume for binary output
VOLUME ["/volume/bin"]

CMD ["sh", "-c", "cp -r /app/bin/* /volume/bin/"]
```

AGENT (Message 8) [gpt-5-chat] 2025/11/16 13:58

```
GreatWSL/Docker is a solid path. Since you want a traditional Makefile
   ↪ with clear pattern rules, Ill give you:

- A clean, minimal Dockerfile tailored for building a C++ LLVM project
- A Makefile that uses llvm-config and supports optional cross-targets
- An optional build.sh wrapper (if you prefer a script entrypoint
   ↪ instead of invoking make directly)

You can pick either Makefile-only or build.sh + Makefile. I recommend
   ↪ Makefile-only for simplicity.

Dockerfile (Makefile-driven)
- Uses Ubuntu base with LLVM dev headers and clang.
- Keeps image slim while providing optional cross linkers (MinGW and
   ↪ AArch64) if you plan to cross-compile.
- Delegates to the Makefiles phony target.
```

```
‘‘‘dockerfile
# C++ LLVM build image
FROM ubuntu:24.04

# Non-interactive apt and basic tools
ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install -y \
  build-essential \
  cmake \
  ninja-build \
  curl \
  git \
  # LLVM toolchain (headers + libs + clang + lld)
  llvm-17 llvm-17-dev clang-17 lld-17 \
  # Optional cross toolchains (uncomment if needed)
  gcc-aarch64-linux-gnu libc6-dev-arm64-cross \
  gcc-mingw-w64-x86-64 \
  && rm -rf /var/lib/apt/lists/*

# Make clang-17 the default clang/clang++
RUN ln -sf /usr/bin/clang-17 /usr/bin/clang && \
    ln -sf /usr/bin/clang++-17 /usr/bin/clang++

# Helpful: expose llvm-config path to Makefile/CMake
ENV LLVM_CONFIG=llvm-config-17

WORKDIR /app

# Copy sources
COPY . .

# Default output dir used by Makefile/build.sh
ENV OUTPUT_DIR=/app/bin
RUN mkdir -p ${OUTPUT_DIR} /volume/bin

# Build via Makefile phony target (edit as needed)
# If you prefer a script, comment this and use build.sh below.
RUN make -j$(nproc) all

# Volume to collect binaries/artifacts
VOLUME ["/volume/bin"]

# Copy build outputs to the mounted volume at container run
CMD ["sh", "-c", "cp -r ${OUTPUT_DIR}/* /volume/bin/ || true"]
‘‘‘
```

```
If you prefer a build.sh entrypoint
- Replace the build RUN line with:
  - RUN chmod +x build.sh && ./build.sh
- And keep CMD the same.

Makefile (portable with llvm-config, pattern rules)
- Standard pattern rule for .cpp -> .o
- Links with LLVM libs discovered via llvm-config
- Optional cross targets for Linux AArch64 and Windows (MinGW), guarded behind phony targets so you can enable only when needed.

‘‘‘make
# Makefile
# Use clang by default. You can also allow overriding via env: make CXX=g++
CXX ?= clang++
AR ?= ar

# Allow overriding this if you need a specific llvm-config
LLVM_CONFIG ?= $(shell command -v llvm-config-17 >/dev/null 2>&1 && echo llvm-config-17 || echo llvm-config)

# Core flags
CXXFLAGS ?= -std=c++20 -O2 -Wall -Wextra -Wpedantic
LDFLAGS  ?=
OUTPUT_DIR ?= bin
BUILD_DIR  ?= build
SRC_DIR    ?= src

# Choose LLVM components
# For IR building/printing, these are sufficient. Add more as needed.
LLVM_COMPONENTS := core support irreader

# Query llvm-config for flags and libs
LLVM_CXXFLAGS := $(shell $(LLVM_CONFIG) --cxxflags)
LLVM_LDFLAGS  := $(shell $(LLVM_CONFIG) --ldflags)
LLVM_LIBS     := $(shell $(LLVM_CONFIG) --libs $(LLVM_COMPONENTS))
SYSTEM_LIBS   := $(shell $(LLVM_CONFIG) --system-libs)

# Sources/Objects
SRCS := $(wildcard $(SRC_DIR)/*.cpp)
OBJS := $(patsubst $(SRC_DIR)/%.cpp,$(BUILD_DIR)/%.o,$(SRCS))

BIN := $(OUTPUT_DIR)/llvm_ir_demo

# Default target
```

```
all: $(BIN)

# Pattern rule: compile .cpp -> .o
$(BUILD_DIR)/%.o: $(SRC_DIR)/%.cpp | $(BUILD_DIR)
    $(CXX) $(CXXFLAGS) $(LLVM_CXXFLAGS) -c $< -o $@

# Link the main binary
$(BIN): $(OBJS) | $(OUTPUT_DIR)
    $(CXX) -o $@ $^ $(LDFLAGS) $(LLVM_LDFLAGS) $(LLVM_LIBS) $(
   ↪ SYSTEM_LIBS)

# Utility targets
run: $(BIN)
    ./$(BIN)

# Generate human-readable LLVM IR from each .cpp via clang (optional)
# Example: make src/main.ll
%.ll: %.cpp
    clang++ -S -emit-llvm -std=c++20 -O0 -g -c $< -o $@

# Cross target: build object code for AArch64 Linux (needs gcc-aarch64-
   ↪ linux-gnu)
# Produces a .o and links via the cross linker
aarch64: CXX := clang++
aarch64: TARGET_TRIPLE := aarch64-linux-gnu
aarch64: $(BIN)-aarch64

$(BIN)-aarch64: $(OBJS)
    $(CXX) --target=$(TARGET_TRIPLE) -o $@ $^ \
    $(LDFLAGS) $(LLVM_LDFLAGS) $(LLVM_LIBS) $(SYSTEM_LIBS) \
    -fuse-ld=lld -static-libgcc -static-libstdc++

# Cross target: Windows (MinGW) using x86_64-w64-mingw32
# Produces a .exe
windows: CXX := x86_64-w64-mingw32-g++
windows: $(BIN).exe

$(BIN).exe: $(OBJS)
    $(CXX) -o $@ $^ \
    $(LDFLAGS) $(LLVM_LDFLAGS) $(LLVM_LIBS) $(SYSTEM_LIBS)

# Dirs
$(BUILD_DIR):
    mkdir -p $(BUILD_DIR)
$(OUTPUT_DIR):
    mkdir -p $(OUTPUT_DIR)
```

```
clean:
    rm -rf $(BUILD_DIR) $(OUTPUT_DIR)

print-llvm:
    @echo "LLVM_CONFIG=$(LLVM_CONFIG)"
    @echo "CXXFLAGS+=$(LLVM_CXXFLAGS)"
    @echo "LIBS=$(LLVM_LIBS)"

.PHONY: all run clean aarch64 windows print-llvm
‘‘‘

Optional build.sh (wrapper)
- Useful if you want one entrypoint that can choose targets (e.g., via env vars) and copy artifacts into OUTPUT_DIR.

‘‘‘bash
#!/usr/bin/env bash
set -euo pipefail

# Respect OUTPUT_DIR if provided by Dockerfile or caller
OUTPUT_DIR="${OUTPUT_DIR:-$(pwd)/bin}"
mkdir -p "$OUTPUT_DIR"

# Choose target via BUILD_TARGET env: all | aarch64 | windows
BUILD_TARGET="${BUILD_TARGET:-all}"

echo "Building target: ${BUILD_TARGET}"
make -j"$(nproc)" "${BUILD_TARGET}"

echo "Artifacts in ${OUTPUT_DIR}:"
find "${OUTPUT_DIR}" -maxdepth 1 -type f -printf "%f\n" 2>/dev/null || true
‘‘‘

Repository layout
- .
  - Dockerfile
  - Makefile
  - build.sh (optional)
  - src/
    - main.cpp

Notes and tips
- In Ubuntu 24.04, llvm-config-17 is available; the Dockerfile sets LLVM_CONFIG=llvm-config-17 so linking stays consistent.
- If you later need LLVM codegen libraries (to emit object files yourself), add components to LLVM_COMPONENTS in the Makefile, e.g
```

```
    ↪ .:
  - target mc mcparser asmprinter x86codegen x86asmparser x86desc
    ↪ x86info
- For best Docker build caching, you can COPY just the build metadata
    ↪ first (Makefile, CMakeLists.txt if any), run dependency installs,
    ↪  then COPY src last. For small projects, the straightforward COPY
    ↪  . . is fine.
- Keep your build tree inside the containers Linux filesystem. For WSL
    ↪ dev outside Docker, build under /home for performance.

Want me to tailor the Makefiles component list for emitting object
    ↪ files or cross-targeting Windows PE from WSL, including a tiny
    ↪ example that writes out.o?
```

USER (Message 9) [gpt-5-chat] 2025/11/16 14:00

```
Why did we install cmake? it looks like we are just using make
```

AGENT (Message 10) [gpt-5-chat] 2025/11/16 14:00

```
Youre right to question it. If were driving the build with a hand-
    ↪ written Makefile that uses llvm-config, CMake isnt required.

Why I included CMake earlier
- Its useful if you later switch to a CMake-based build or want IDE/
    ↪ Ninja/VS generators.
- But for a Makefile-only workflow, its extra.

What to do: drop CMake from the image
Use this slimmer Dockerfile that aligns with a Makefile-driven build:

‘‘‘dockerfile
FROM ubuntu:24.04
ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install -y \
  build-essential \
  curl git \
  llvm-17 llvm-17-dev clang-17 lld-17 \
  # Optional cross toolchains (uncomment if you need them)
  # gcc-aarch64-linux-gnu libc6-dev-arm64-cross \
  # gcc-mingw-w64-x86-64 \
  && rm -rf /var/lib/apt/lists/*

# Make clang-17 the default
RUN ln -sf /usr/bin/clang-17 /usr/bin/clang && \
    ln -sf /usr/bin/clang++-17 /usr/bin/clang++
```

````
# Expose llvm-config for Makefile
ENV LLVM_CONFIG=llvm-config-17

WORKDIR /app
COPY . .

# Output dir used by the Makefile
ENV OUTPUT_DIR=/app/bin
RUN mkdir -p ${OUTPUT_DIR} /volume/bin

# Build via Makefile phony target
RUN make -j"$(nproc)" all

VOLUME ["/volume/bin"]
CMD ["sh", "-c", "cp -r ${OUTPUT_DIR}/* /volume/bin/ || true"]
‘‘‘

If you keep a build.sh instead of make
- Replace the build RUN step with:
  - RUN chmod +x build.sh && ./build.sh

Summary
- CMake is optional. Remove it if youre sticking with Make.
- You can always add it back later if you move to a CMakeLists.txt
   ↪ workflow or want multi-generator support.
````

USER (Message 11) [gpt-5-chat] 2025/11/16 14:02

````
Why is curl and build-essential and ninja and all added. What precisely
   ↪  do i actually need
````

AGENT (Message 12) [gpt-5-chat] 2025/11/16 14:02

````
Short answer
- If youre using a hand-written Makefile with llvm-config: you only
   ↪ need clang, llvm-dev, and a linker (lld is nice but optional).
- Everything else (curl, ninja, cmake, build-essential) is optional
   ↪ depending on how you build.

What you actually need for a Makefile-only LLVM C++ build
- clang-17 (or gcc): C/C++ compiler
- llvm-17-dev: LLVM headers and libraries (and llvm-config)
- lld-17 (optional, recommended): Fast LLVM linker; otherwise the
   ↪ system linker will be used
- make: Already in Ubuntu base via make or build-essential; if you
   ↪ remove build-essential, install make explicitly
````

```
Minimal Dockerfile (strictly necessary components)
- Uses clang++ and llvm-config; no cmake, no ninja, no curl

‘‘‘dockerfile
FROM ubuntu:24.04
ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install -y \
  make \
  clang-17 \
  llvm-17-dev \
  lld-17 \
  && rm -rf /var/lib/apt/lists/*

# Make clang-17 the default
RUN ln -sf /usr/bin/clang-17 /usr/bin/clang && \
    ln -sf /usr/bin/clang++-17 /usr/bin/clang++

# Point builds at the matching llvm-config
ENV LLVM_CONFIG=llvm-config-17
ENV OUTPUT_DIR=/app/bin

WORKDIR /app
COPY . .
RUN mkdir -p ${OUTPUT_DIR} /volume/bin

# Build using your Makefile
RUN make -j"$(nproc)" all

VOLUME ["/volume/bin"]
CMD ["sh", "-c", "cp -r ${OUTPUT_DIR}/* /volume/bin/ || true"]
‘‘‘

When you might add the other packages
- build-essential: Installs gcc/g++, make, libc headers. Not needed if
   ↪ you use clang and explicitly install make (as above). Keep it if
   ↪ you prefer gcc/g++ or commonly need standard dev tools.
- ninja-build: Only needed if you use Ninja as your build system (e.g.,
   ↪  with CMake or meson). Not needed for a Makefile workflow.
- cmake: Only needed if you switch to a CMake-based build.
- curl/git: Only if your Docker build fetches external sources or you
   ↪ need them inside the image. Not needed for compiling your local
   ↪ code.
- gcc-aarch64-linux-gnu, libc6-dev-arm64-cross: Only if you cross-
   ↪ compile to Linux AArch64.
- gcc-mingw-w64-x86-64: Only if you cross-compile to Windows using
```

```
    ↪ MinGW.
- lld-17: Optional but recommended; faster and pairs well with clang.
    ↪ If you drop it, the system linker (ld.bfd or ld.gold) will be
    ↪ used automatically.

If you want the absolute bare minimum and are okay with the system
    ↪ linker and default clang version:
- Replace clang-17, llvm-17-dev, lld-17 with distro-default clang, llvm
    ↪ -dev, lld (may track a different LLVM version). Just ensure llvm-
    ↪ config matches the headers/libs used.
```

USER (Message 13) [gpt-5-chat] 2025/11/16 14:33

```
what does ?= do
```

AGENT (Message 14) [gpt-5-chat] 2025/11/16 14:33

```
In a Makefile, the ?= operator is a conditional (set-if-unset)
    ↪ assignment.

- X ?= Y means: If X is not already defined (by the environment,
    ↪ command line, or earlier in the Makefile), set X to Y. Otherwise,
    ↪  leave X as-is.

Why its useful
- Lets users override variables from the command line or environment
    ↪ without editing the Makefile.
- Provides sane defaults while remaining configurable.

Precedence summary (high level)
- Command line: make CXX=g++  overrides everything inside the Makefile.
- Environment variables (exported) are imported unless the Makefile
    ↪ sets the variable with :=, =, or ?= (details below).
- Earlier assignments in the Makefile can be preserved by using ?=
    ↪ later.

Examples
- Default compiler, but allow override:
  CXX ?= clang++
  # User can run: make CXX=g++ to switch to g++

- Default optimization, but allow overrides:
  CXXFLAGS ?= -O2 -g

- Default llvm-config tool, but prefer versioned one if present:
  LLVM_CONFIG ?= llvm-config-17
```

```
Contrast with other assignment operators
- = (recursive/late expansion): Expands the right-hand side each time
   ↪ the variable is used, with current values of referenced variables
   ↪ .
  FOO = $(BAR)
  BAR = hi
  # Using $(FOO) later expands to "hi" (whatever BAR is at use time)

- := (simple/immediate expansion): Expands the right-hand side once, at
   ↪  assignment time.
  BAR = hi
  FOO := $(BAR)
  BAR = bye
  # $(FOO) stays "hi"

- ?= (set-if-unset): Assigns only if the variable isnt already defined.
  FOO ?= default
  # If FOO was set via env/CLI/earlier, this line does nothing.

- += (append): Appends text to an existing variable (creates it if it
   ↪ didnt exist; append semantics depend on = vs := context).
  CXXFLAGS += -Wall -Wextra

Edge notes
- Unset means not defined at all. If a variable is defined but empty,
   ↪ GNU Make treats it as set; ?= will not overwrite it.
- Exported environment variables count as already defined. To ignore
   ↪ env values, you can do: override VAR := value, or unexport VAR,
   ↪ or assign before including environment.

Quick demo
- Makefile:
  GREETING ?= hello
  all:
    @echo "$(GREETING)"

- Run:
  make           # prints "hello"
  make GREETING=hey    # prints "hey"
  GREETING=yo make     # prints "yo"
```

USER (Message 15) [gpt-5-chat] 2025/11/16 14:33

```
so i should just use := if i want to ensure it is determonistic?
```

AGENT (Message 16) [gpt-5-chat] 2025/11/16 14:33

```
Short answer: Usually yesuse := when you want a variable to be expanded
    ↪  once, right now, so its value wont change later as other
    ↪ variables change. That gives you deterministic, time-of-
    ↪ assignment values.

What := does
- Simple (immediate) expansion: the right-hand side is expanded exactly
    ↪  once when the line is read. References to other variables are
    ↪ resolved then, and the result is stored.
- Later changes to referenced variables wont affect it.

When to use :=
- You compute a value from other variables and want it frozen (e.g.,
    ↪ probing tools, composing paths).
- You shell out once and cache the result (e.g., llvm-config output).
- You want predictable behavior independent of variable mutation order.

Example
- Deterministic capture of llvm-config flags:
  LLVM_CONFIG := $(shell command -v llvm-config-17 >/dev/null 2>&1 &&
    ↪ echo llvm-config-17 || echo llvm-config)
  LLVM_CXXFLAGS := $(shell $(LLVM_CONFIG) --cxxflags)
  LLVM_LDFLAGS  := $(shell $(LLVM_CONFIG) --ldflags)

- Even if LLVM_CONFIG changes later, LLVM_CXXFLAGS and LLVM_LDFLAGS
    ↪ stay as computed.

How it differs from =
- = is recursive (lazy) expansion: the right-hand side is re-evaluated
    ↪ each time the variable is used.
- That can be handy, but can also lead to surprising changes if
    ↪ dependencies mutate.

Example showing the difference
- Recursive (=):
  BAR = later
  FOO = $(BAR)
  BAR = now
  # Using $(FOO) prints "now"
- Simple (:=):
  BAR = later
  FOO := $(BAR)
  BAR = now
  # Using $(FOO) prints "later"

Where ?= fits
- ?= sets a default only if the variable is not already defined (by env
/CLI/earlier lines).
- Use it for user-overridable defaults.
- Combine with := when you want a deterministic computed default:
  PREFIX ?= /usr/local
  BIN_DIR := $(PREFIX)/bin
  # BIN_DIR freezes to the chosen PREFIX at parse time.

Append (+=) nuance
- With := context, appends happen immediately:
  CFLAGS := -O2
  CFLAGS += -Wall
- With = context, appends affect the recursive text and are expanded at use time.

Practical guidance
- Use := for values from $(shell ), path probes, or any computed value that you want stable.
- Use ?= for user-overridable defaults (CXX ?= clang++).
- Use = only when you intentionally want lazy/recursive behavior.
```

Thread (8 messages)
ID: 42e8ae10-5d13-4ecb-8821-841f9ec04e0e **2025/11/16 15:33**

USER (Message 1) [gpt-5-chat] 2025/11/16 15:33

```
i need you to generate a ".tex" file that has the following sections:

1) Project Objectives

2) Description of work completed

2.1) Evidence of work completed

2.2) Literature review

2.2) Use of GenAI and tools

3) Future Work

At the top it should have a table.

Student Name / blank / Student Number / blank
Supervisor / blank / ECTS Credits / blank
Project Title / blank / blank / blank [these 3 columns merged].

There should be a header repeated on each page.
```

The header should contain the following text centered in gray:

FYP Interim Report 2025-2026
Department of Computer Science

Floating to the right of that text should be an image "./mu.png"

AGENT (Message 2) [gpt-5-chat] 2025/11/16 15:33

Below is a complete, standalone LaTeX document you can save as report.tex. It sets up a repeated page header with centered gray text and a right-aligned image, includes the requested table at the top of the first page, and defines the sections in the order you specified.

Notes:
- The header is repeated on each page using fancyhdr.
- The centered gray header text is stacked on two lines; the image ./mu.png is floated to the right.
- The top table uses a four-column layout; the last row merges the last three columns for Project Title.
- Compile with pdflatex or xelatex. Ensure mu.png is in the same directory.

```
‘‘‘latex
\documentclass[12pt,a4paper]{article}

% Encoding and font
\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}
\usepackage{lmodern}

% Page geometry
\usepackage[a4paper,margin=1in]{geometry}

% Colors and graphics
\usepackage{xcolor}
\usepackage{graphicx}

% Tables
\usepackage{array}
\usepackage{booktabs}
\usepackage{multirow}
\usepackage{tabularx}

% Header/footer
```

```
\usepackage{fancyhdr}
\usepackage{lastpage}

% Spacing
\usepackage{setspace}
\onehalfspacing

% Links (optional)
\usepackage[hidelinks]{hyperref}

% Define a light gray color for header text
\definecolor{headergray}{gray}{0.35}

% Header setup
\pagestyle{fancy}
\fancyhf{} % clear all header and footer fields

% Centered two-line gray header text
\newcommand{\HeaderCenterText}{%
  {\color{headergray}\sffamily\bfseries FYP Interim Report 2025-2026}\\
  {\color{headergray}\sffamily Department of Computer Science}%
}

% Right-aligned image command (ensure mu.png exists)
\newcommand{\HeaderRightImage}{%
  \raisebox{-0.2\height}{\includegraphics[height=1.2cm]{mu.png}}%
}

% Header layout: center text, image to the right
\fancyhead[C]{\HeaderCenterText}
\fancyhead[R]{\HeaderRightImage}

% Optional footer with page number
\fancyfoot[C]{\sffamily Page \thepage\ of \pageref{LastPage}}

% Reduce header rule and increase headheight if needed for the image
\renewcommand{\headrulewidth}{0pt}
\setlength{\headheight}{24pt} % adjust if warning appears

% Title spacing (we won't use \maketitle)
\setlength{\parskip}{0.6em}
\setlength{\parindent}{0pt}

\begin{document}

% Top table
\begin{table}[ht]
```

~~~
  \renewcommand{\arraystretch}{1.4}
  \setlength{\tabcolsep}{8pt}
  \begin{tabularx}{\textwidth}{@{} >{\bfseries}l X >{\bfseries}l X @{}}
    \toprule
    Student Name & & Student Number & \\
    Supervisor & & ECTS Credits & \\
    Project Title & \multicolumn{3}{l}{\rule{0pt}{0pt}} \\
    \bottomrule
  \end{tabularx}
\end{table}

% Sections
\section{Project Objectives}
% Describe the primary goals, success criteria, and scope boundaries.

\section{Description of Work Completed}
% Provide a narrative of tasks completed to date, milestones achieved,
   ↪ and timelines.

\subsection{Evidence of Work Completed}
% Include links, figures, code snippets, commit logs, screenshots, or
   ↪ experiment outputs.
% Example: Figure references, tables, or brief summaries of artifacts.

\subsection{Literature Review}
% Summarize key works, methods, and how they relate to your project.
% Cite sources here if using BibTeX or \bibitem later.

\subsection{Use of GenAI and Tools}
% Detail any GenAI models, prompts, and tools used.
% Include rationale, limitations, and safeguards for academic integrity
   ↪ .

\section{Future Work}
% Outline next steps, risks, mitigation plans, and deliverables for the
   ↪  next phase.

% Optional: References (uncomment and use if needed)
% \begin{thebibliography}{9}
% \bibitem{key1} Author, Title, Venue/Publisher, Year.
% \end{thebibliography}

\label{LastPage}
\end{document}
```
~~~

USER (Message 3) [gpt-5-chat] 2025/11/16 15:46

i also need an appendix section

AGENT (Message 4) [gpt-5-chat] 2025/11/16 15:46

Understood. Here is an updated standalone LaTeX document that adds an Appendix section at the end. Save as report.tex and compile with pdflatex or xelatex. Ensure mu.png is in the same directory.

```latex
\documentclass[12pt,a4paper]{article}

% Encoding and font
\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}
\usepackage{lmodern}

% Page geometry
\usepackage[a4paper,margin=1in]{geometry}

% Colors and graphics
\usepackage{xcolor}
\usepackage{graphicx}

% Tables
\usepackage{array}
\usepackage{booktabs}
\usepackage{multirow}
\usepackage{tabularx}

% Header/footer
\usepackage{fancyhdr}
\usepackage{lastpage}

% Spacing
\usepackage{setspace}
\onehalfspacing

% Links (optional)
\usepackage[hidelinks]{hyperref}

% Define a light gray color for header text
\definecolor{headergray}{gray}{0.35}

% Header setup
\pagestyle{fancy}
\fancyhf{} % clear all header and footer fields
```

```
% Centered two-line gray header text
\newcommand{\HeaderCenterText}{%
  {\color{headergray}\sffamily\bfseries FYP Interim Report 2025-2026}\\
  {\color{headergray}\sffamily Department of Computer Science}%
}

% Right-aligned image command (ensure mu.png exists)
\newcommand{\HeaderRightImage}{%
  \raisebox{-0.2\height}{\includegraphics[height=1.2cm]{mu.png}}%
}

% Header layout: center text, image to the right
\fancyhead[C]{\HeaderCenterText}
\fancyhead[R]{\HeaderRightImage}

% Optional footer with page number
\fancyfoot[C]{\sffamily Page \thepage\ of \pageref{LastPage}}

% Reduce header rule and increase headheight if needed for the image
\renewcommand{\headrulewidth}{0pt}
\setlength{\headheight}{24pt} % adjust if warning appears

% Title spacing (we won't use \maketitle)
\setlength{\parskip}{0.6em}
\setlength{\parindent}{0pt}

\begin{document}

% Top table
\begin{table}[ht]
  \renewcommand{\arraystretch}{1.4}
  \setlength{\tabcolsep}{8pt}
  \begin{tabularx}{\textwidth}{@{} >{\bfseries}l X >{\bfseries}l X @{}}
    \toprule
    Student Name & & Student Number & \\
    Supervisor & & ECTS Credits & \\
    Project Title & \multicolumn{3}{l}{\rule{0pt}{0pt}} \\
    \bottomrule
  \end{tabularx}
\end{table}

% Sections
\section{Project Objectives}
% Describe the primary goals, success criteria, and scope boundaries.

\section{Description of Work Completed}
% Provide a narrative of tasks completed to date, milestones achieved,
```

~~~
    ↪ and timelines.

\subsection{Evidence of Work Completed}
% Include links, figures, code snippets, commit logs, screenshots, or
    ↪ experiment outputs.
% Example: Figure references, tables, or brief summaries of artifacts.

\subsection{Literature Review}
% Summarize key works, methods, and how they relate to your project.
% Cite sources here if using BibTeX or \bibitem later.

\subsection{Use of GenAI and Tools}
% Detail any GenAI models, prompts, and tools used.
% Include rationale, limitations, and safeguards for academic integrity
    ↪ .

\section{Future Work}
% Outline next steps, risks, mitigation plans, and deliverables for the
    ↪  next phase.

% Optional: References (uncomment and use if needed)
% \begin{thebibliography}{9}
% \bibitem{key1} Author, Title, Venue/Publisher, Year.
% \end{thebibliography}

% Appendix
\appendix
\section{Appendix}
% Place supplementary materials here: extended tables/figures, detailed
    ↪  methods,
% additional experiments, raw outputs, ethics statements,
    ↪ questionnaires, etc.

\label{LastPage}
\end{document}
```
~~~

USER (Message 5) [gpt-5-chat] 2025/11/16 15:47

```
can there be multiple appendices
```

AGENT (Message 6) [gpt-5-chat] 2025/11/16 15:47

```
Yes. In LaTeX, after you call \appendix, each subsequent \section
    ↪ creates a new appendix, automatically labeled A, B, C, etc. You
    ↪ can also create sub-sections within each appendix.
```

```
Heres how to extend your document to have multiple appendices:

‘‘‘latex
% ... keep the preamble and main sections as before ...

% Appendices
\appendix

\section{Appendix A: Extended Methods}
% Content for appendix A

\section{Appendix B: Additional Results}
% Content for appendix B

\section{Appendix C: Survey Instrument}
% Content for appendix C
‘‘‘

Tips:
- If you prefer Appendix A, Appendix B to appear in the table of
   ↪ contents, ensure you have ‘\tableofcontents‘ earlier in the
   ↪ document.
- If you want Appendix to appear as a heading before the list of
   ↪ appendices, you can add:
  ‘‘‘latex
  \clearpage
  \phantomsection
  \addcontentsline{toc}{section}{Appendices}
  ‘‘‘
  just before ‘\appendix‘ (and then keep the individual appendix
   ↪ sections as shown).
```

USER (Message 7) [gpt-5-chat] 2025/11/16 15:48

```
in a programming project, what should be in my appendix? so far i have
   ↪ one for source code, and another for llm prompts. what else?
```

AGENT (Message 8) [gpt-5-chat] 2025/11/16 15:48

```
Great start with source code and LLM prompts. For a programming project
   ↪ , appendices are ideal for detailed artifacts that support but
   ↪ would clutter the main report. Consider these options (pick whats
   ↪  relevant to your project and assessment rubric):

- Architecture and design artifacts
  - System architecture diagrams (high-level and component-level)
  - Data model/ER diagrams and database schema DDL
  - API specifications (OpenAPI/Swagger), endpoint tables, request/
   ↪ response examples
  - State diagrams, sequence/activity diagrams, threat model diagrams (
   ↪ e.g., STRIDE)

- Experimental setup and results
  - Reproducibility details: hardware specs, OS versions, container
   ↪ images, random seeds
  - Hyperparameters, training/validation splits, and evaluation
   ↪ protocols
  - Full result tables and ablation studies, error analyses, confusion
   ↪ matrices
  - Raw or extended benchmark outputs beyond whats summarized in the
   ↪ main text

- Build, deployment, and ops
  - Installation and build instructions (full version), dependency
   ↪ lockfiles, environment files
  - CI/CD pipeline configs, test coverage reports
  - Dockerfiles, docker-compose, Kubernetes manifests, IaC snippets (
   ↪ Terraform)
  - Logging/monitoring configurations, dashboards, alert rules

- Testing and verification
  - Test plans, test case matrices, and representative test data (
   ↪ sanitized)
  - Performance/load test scripts and results (e.g., k6/JMeter, Locust
   ↪ outputs)
  - Security testing evidence: SAST/DAST report summaries, dependency
   ↪ vulnerability scans
  - Formal specifications or property checks (if applicable)

- Data and datasets
  - Dataset cards or documentation (provenance, licenses, splits,
   ↪ preprocessing)
  - Schema of input/output data, validation rules, and example records
  - Data collection instruments (surveys, consent forms) and ethics
   ↪ approvals

- Governance, ethics, and compliance
  - Risk assessment, model cards, privacy impact assessments
  - Accessibility checks (WCAG) and internationalization details
  - License choices and third-party license summaries/notices

- Configuration and customization
  - Full application configuration files (with secrets redacted)
  - Feature flags, parameter files, and example runtime profiles
- User-facing materials
  - User manual (full version), CLI help screens, screenshot
   ↪ walkthroughs
  - UX research artifacts: personas, task analyses, wireframes

- Project management evidence
  - Gantt charts/roadmaps, milestone burndowns, issue logs
  - Meeting notes/decision records (ADR documents)

- LLM/GenAI specifics (beyond prompts)
  - Prompt templates, few-shot exemplars, system instructions
  - Output sampling settings (temperature, top-p), token limits, safety
   ↪  filters
  - Retrieval-augmented generation details: index schema, chunking
   ↪ strategy, embeddings model
  - Guardrails, prompt injection defenses, red-teaming results

- Reproducibility bundle
  - A one-command repro script, Makefile, or runner
  - Checksums of datasets/models, and instructions to re-download or
   ↪ rebuild
  - Seeded demo data for local runs

Practical tips:
- Keep secrets out; include redacted examples and document secret
   ↪ management.
- Reference appendices in the main text where relevant, so reviewers
   ↪ know why theyre there.
- If space is limited, move large artifacts to a repository release or
   ↪ appendix zip and include links with checksums.
```

Thread (6 messages)
ID: 84682898-db2e-4b16-83d1-a793f25c8ed0 **2025/11/17 14:22**

USER (Message 1) [gpt-5-chat] 2025/11/17 14:22

```
How to make "cp -r /app/out/* /volume/out/" fail silently when no files
   ↪  are found?
```

AGENT (Message 2) [gpt-5-chat] 2025/11/17 14:22

```
Use a glob that doesnt error when it matches nothing, or guard the copy
   ↪ .

Options:
```

````
- Enable nullglob (Bash)
  - Makes unmatched globs expand to nothing instead of the literal
 ↪ pattern.
  - Then cp sees no arguments and does nothing (exit 0).
  - Example:
    ‘‘‘bash
    shopt -s nullglob
    cp -r /app/out/* /volume/out/
    ‘‘‘
  - If you want this only for one command:
    ‘‘‘bash
    bash -O nullglob -c ’cp -r /app/out/* /volume/out/’
    ‘‘‘

- Use a conditional test
  - Only run cp if there is at least one non-hidden file:
    ‘‘‘bash
    set -e
    files=(/app/out/*)
    [ -e "${files[0]}" ] && cp -r /app/out/* /volume/out/
    ‘‘‘
  - To include dotfiles:
    ‘‘‘bash
    shopt -s nullglob dotglob
    files=(/app/out/*)
    [ "${#files[@]}" -gt 0 ] && cp -r /app/out/* /volume/out/
    ‘‘‘

- Use find to drive cp (works in POSIX sh)
  - Non-recursive copy of files in top-level:
    ‘‘‘bash
    find /app/out -mindepth 1 -maxdepth 1 -exec cp -r -t /volume/out {}
 ↪  +
    ‘‘‘
    This does nothing if nothing is found and exits 0.

- Suppress the specific No such file or directory error
  - If youre okay ignoring only that case:
    ‘‘‘bash
    cp -r /app/out/* /volume/out/ 2> >(grep -v ’No such file or
 ↪ directory’ >&2)
    ‘‘‘
    Still fails on other errors.

Notes:
- cp -r will not copy dotfiles when using /app/out/* unless dotglob is
````

```
    ↪ set.
- If you need to ensure destination exists, create it first: ‘mkdir -p
    ↪ /volume/out‘.
- For POSIX sh without Bash features, prefer the find-based solution.
```

USER (Message 3) [gpt-5-chat] 2025/11/17 14:24

```
making the glob expand correctly sounds like the safest bet. but will
    ↪ an empty "null" list result in the parameter of files omitted
    ↪ entirely? will cp think there is one less parameter, or will it
    ↪ know there is a list parameter containing no entries? essentially
    ↪ , i want cp to recognize that there are no files, instead of
    ↪ having it fail due to a weird syntax issue that is potentially
    ↪ dangerous
```

AGENT (Message 4) [gpt-5-chat] 2025/11/17 14:24

```
Great question. In Bash with nullglob enabled, unmatched globs expand
    ↪ to zero words, not to an empty string and not to a list argument.
    ↪  That means the entire word is removed from the command line
    ↪ before cp is invoked.

Consequence:
- With nullglob set, the command
  ‘‘‘bash
  shopt -s nullglob
  cp -r /app/out/* /volume/out/
  ‘‘‘
  becomes, at execution time:
  - If there are matches: cp -r /app/out/file1 /app/out/dir2 ... /
    ↪ volume/out/
  - If there are no matches: cp -r /volume/out/

In the no matches case, cp sees only the options plus the destination
    ↪ directory, with no source operands. GNU cp and most
    ↪ implementations will then print missing file operand and exit
    ↪ nonzero. So simply enabling nullglob is not enough if you want a
    ↪ silent no-op.

How to make it safe and silent:
- Use a pre-expansion check to ensure there is at least one source
    ↪ before running cp:
  ‘‘‘bash
  shopt -s nullglob
  files=(/app/out/*)
  if (( ${#files[@]} )); then
    cp -r "${files[@]}" /volume/out/
  fi
  ```
  - This is safe: if there are no matches, files is an empty array and
    ↪ cp is not invoked.
  - When there are matches, you pass them as separate arguments,
    ↪ exactly as the shell would.

Include dotfiles too (optional):
  ```bash
  shopt -s nullglob dotglob
  files=(/app/out/*)
  if (( ${#files[@]} )); then
    cp -r "${files[@]}" /volume/out/
  fi
  ```

POSIX sh approach (no Bash arrays/features):
  ```bash
  set -- /app/out/*  # expands to zero or more words; in POSIX sh,
    ↪ unmatched globs stay literal, so guard it:
  if [ -e /app/out/* ]; then
    cp -r /app/out/* /volume/out/
  fi
  ```
  But since unmatched globs dont disappear in plain sh, a robust,
    ↪ portable method is find:
  ```bash
  find /app/out -mindepth 1 -maxdepth 1 -exec cp -R -t /volume/out {} +
  ```
  This does nothing and exits 0 if nothing is found.

Summary:
- nullglob removes the glob entirely when it has no matches, which
    ↪ would make cp complain about missing operands.
- The safe pattern is: enable nullglob, expand into an array, check the
    ↪ array length, and only then call cp with "${files[@]}".
```

USER (Message 5) [gpt-5-chat] 2025/11/17 14:26

```
does shopt last for the duration of the shell script?
```

AGENT (Message 6) [gpt-5-chat] 2025/11/17 14:26

```
Short answer: Yes, within the current shell. A shopt setting persists
    ↪ for the lifetime of the current shell (and any code that runs in
    ↪ it) until you change it back or the shell exits.
```
```

~~~
Details:
- In an interactive session, shopt settings remain in effect until you unset them, start a new shell, or exit.
- In a shell script, they apply from the point theyre set until the script ends (or you toggle them back).
- They do not automatically propagate to child processes; subshells start with their own option state. Examples:
  - Subshells: ( shopt -s nullglob; ... ) limits the change to that subshell.
  - Separate processes: bash -c '...' starts with default shopt values unless you set them inside that -c.
- You can scope changes:
  - Use a subshell to confine: ( shopt -s nullglob; your commands )
  - Or save/restore:
    ```bash
    # Save current state (1 if set, 0 if unset)
    save=$(shopt -p nullglob)
    shopt -s nullglob
    # ... your commands ...
    eval "$save"  # restores previous state
    ```

So in your cp case, enabling nullglob once near the top of the script will cover subsequent commands in that script unless you deliberately revert it or enter a new subshell.
~~~