AymaraLang (aym) y su compilador aymc

Lenguaje con palabras clave en aymara + compilador C++17 que genera binarios nativos para Linux/Windows.

Notas del orador: Presentación de alto nivel del lenguaje y del compilador que produce ejecutables x86_64 multiplataforma.

Agenda

- Estructura del repositorio
- Flujo de trabajo del proyecto
- Arquitectura y funcionamiento del compilador
- CLI, build y herramientas
- Ejemplos de código y demos
- Pruebas y calidad
- Roadmap y contribuciones

Notas del orador: Enmarcar la charla: primero repo y dinámica, luego cómo funciona, después demos y cierre.

Estructura del Repositorio

```
.
├─ compiler/              # Código del compilador 'aymc'
│  ├─ lexer/              # Análisis léxico → tokens
│  ├─ parser/             # Análisis sintáctico → AST
│  ├─ ast/                # Nodos y visitante del AST
│  ├─ semantic/           # Tipos, ámbitos, validaciones
│  ├─ codegen/            # Emisión NASM + enlace
│  ├─ interpreter/        # Intérprete (usado por REPL)
│  ├─ builtins/           # Funciones integradas
│  ├─ utils/              # Utilidades (fs, errores)
│  └─ main.cpp            # Entrada y orquestación CLI
├─ runtime/               # Biblioteca mínima de E/S
├─ samples/               # Programas de ejemplo .aym
├─ tests/                 # Unit tests y pruebas de samples
├─ docs/                  # Guías, arquitectura, slides
├─ CMakeLists.txt, Makefile, build.bat
└─ README.md, LICENSE
```

Notas del orador: Mostrar el mapa mental del repo. Mencionar docs/BUILD.md, docs/arquitectura.md, docs/repl.md como referencia.

Objetivo y Panorama

- Lenguaje: palabras clave en aymara; sintaxis sencilla para docencia.
- Compilador: C++17 + NASM + GCC/MinGW.
- Plataformas: Linux y Windows (ABI y enlace adaptados).
- Repositorio: `compiler/`, `runtime/`, `samples/`, `tests/`, `docs/`.

Notas del orador: Propósito cultural/educativo y alcance técnico: pipeline clásico sin dependencias pesadas.

Flujo de Trabajo (equipo)

- Issues → ramas con prefijo (`feat/`, `fix/`, `docs/`).
- Desarrollo: compilar con `cmake`/`make` y probar local con `make test`.
- Commits pequeños y descriptivos; referencias a issues.
- Pull Request: revisión de pares; ejecutar samples y unit tests.
- Merge a `main`; actualizar `docs/` y `samples/` cuando aplique.

Notas del orador: Para cambios del lenguaje: lexer → parser/AST → semantic → codegen → tests.

Pipeline de Compilación

- Entrada: `.aym` + flags CLI.
- Fases: Léxico → Parser/AST → Semántica → ASM → Objeto → Enlace.
- Salida: `build/<nombre>.asm`, `build/*.o|.obj`, `bin/<nombre>[.exe]`.

Notas del orador: Mostrar el flujo completo y dónde se materializa cada artefacto en disco.

CLI y Flujo (main)

- Archivo: `compiler/main.cpp`.
- Flags: `-o`, `--debug`, `--dump-ast`, `--repl`, `--windows`, `--linux`.
- Entrada múltiple: concatena fuentes; salida por defecto: `build/<stem-del-primer-archivo>.asm` y binario en `bin/`.

Notas del orador: Destacar el modo REPL para demos rápidas.

Build y Herramientas

- Requisitos: `g++` (>=8), `nasm`, `gcc` (link), `cmake` (>=3.15) o `make`.
- Linux/macOS:
  - `cmake -S . -B build -DCMAKE_BUILD_TYPE=Release`
  - `cmake --build build -j`
- Windows (MinGW):
  - `cmake -S . -B build -G "MinGW Makefiles" -DCMAKE_BUILD_TYPE=Release`
  - `cmake --build build -j`
- Alternativas: `make` (Linux) o `build.bat` (Windows).

Notas del orador: El binario queda en `build/bin/aymc[.exe]`. La salida de programas .aym termina en `bin/`.

Léxico (Lexer)

- Archivos: `compiler/lexer/lexer.h`, `compiler/lexer/lexer.cpp`.
- Tareas: ignora espacios/comentarios; reconoce números, strings, símbolos, keywords.
- Keywords: `jach'a`(int), `lliphiphi`(float), `qillqa`(string), `chuymani`(bool), `willt'aña`, `si`, `sino`, `mientras`, `para`, `tantachaña`, `lurüwi`, `kutiyana`.
- Compuestos: "jan uka" (OR), "cheka" (1), "jan cheka" (0).

Notas del orador: Tokens multi-palabra y soporte básico de Unicode para palabras clave.

Parser y AST

- Archivos: `compiler/parser/parser.h`, `compiler/parser/parser.cpp`.
- AST: `compiler/ast/ast.h` con nodos `Expr`/`Stmt` (If/While/For/Do/Switch/Func...).
- Precedencia: lógica → igualdad → comparación → suma → término → potencia → factor.
- Azúcar: `para i en range(a, b)` se desazucara a init/cond/post.

Notas del orador: `willt'aña(1);` → `PrintStmt(NumberExpr(1))`.

Análisis Semántico

- Archivos: `compiler/semantic/semantic.h`, `compiler/semantic/semantic.cpp`.
- Escopos/Tipos: declaraciones, uso, verificación de tipos, contextos de control.
- Builtins: `compiler/builtins/` para `willt'aña`, `input`, `length`.
- Exporta a CodeGen: globales, tipos de parámetros y globales.

Notas del orador: Inferencia básica de strings en parámetros por uso.

Generación de Código (ASM x86_64)

- Archivo: `compiler/codegen/codegen.cpp`.
- Datos: literales y formatos `fmt_int`/`fmt_str`; extern `printf/scanf/strlen`.
- ABI: Win64 (RCX/RDX/R8/R9 + shadow space 32B) vs SysV (RDI/RSI/...; `-no-pie` al enlazar).
- Flujo: etiquetas para `if/else` y ciclos; pilas de `break/continue`.

Notas del orador: Alineación de pila y booleanos en `rax` (0/1).

Ensamblado y Enlace

- Ensamblador: `nasm -f win64` (Windows) / `nasm -felf64` (Linux).
- Enlace: `gcc`/`MinGW` → `bin/<nombre>[.exe]`.
- Mensaje final: `[aymc] Ejecutable generado: ...`.

Notas del orador: Ventajas de delegar a toolchain estándar.

REPL e Intérprete

- REPL: `./bin/aymc --repl`.
- Intérprete: `compiler/interpreter/interpreter.h`, `compiler/interpreter/interpreter.cpp`.
- Builtins: `willt'aña`, `input`, `length`.

Notas del orador: Útil para enseñar sin compilar a binario.

Demo Rápida (Hola)

Comandos (Linux):

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build -j
./build/bin/aymc samples/hola.aym
./bin/hola
```

Comandos (Windows MSYS2 MINGW64):

```
cmake -S . -B build -G "MinGW Makefiles" -DCMAKE_BUILD_TYPE=Release
cmake --build build -j
./build/bin/aymc.exe samples/hola.aym
./bin/hola.exe
```

Salida esperada:

```
Kamisaraki!
```

Notas del orador: Mostrar el artefacto `build/hola.asm` si se quiere profundizar.

Demo Lógica y Control

Fuente:

```
// samples/logic.aym
si (1 uka janiwa 0) {
    willt'aña("ok");
} sino {
    willt'aña("fail");
}
```

Comandos:

```
./build/bin/aymc samples/logic.aym
./bin/logic
```

Salida esperada:

```
ok
```

Notas del orador: Cómo el lexer construye el token compuesto "jan uka".

Errores Comunes

- Variable no declarada: uso antes de declarar.
- Tipos incompatibles en asignaciones o expresiones.
- `break`/`continue` fuera de bucles o `switch`.
- `return` fuera de una función.

Notas del orador: Mostrar ejemplo mínimo que dispare cada error y el mensaje.

Multiplataforma

- Selección: `--windows` / `--linux`.
- Win64: RCX/RDX/R8/R9, shadow space (32B).
- Linux: SysV y `-no-pie`.

Notas del orador: Cruzar plataformas requiere toolchain de destino (mingw-w64 o build en Linux).

Pruebas

- Samples automatizados: `tests/run_tests.sh`.
- Unit test básico: `tests/unittests/test_compiler.cpp`.
- Linux: `make test`.

Notas del orador: Los tests compilan samples y validan su salida.

Roadmap

- Tipado más rico (float/bool) y coerciones.
- Más builtins y runtime.
- Backend LLVM opcional.
- Mensajería de errores mejorada (UTF-8, subrayado).

Notas del orador: Invitar a contribuciones y a probar nuevos casos.

Referencias Rápidas

- `compiler/main.cpp` - Entrada CLI y orquestación.
- `compiler/lexer/lexer.cpp` - Tokenización.
- `compiler/parser/parser.cpp` - Construcción de AST.
- `compiler/semantic/semantic.cpp` - Análisis semántico.
- `compiler/codegen/codegen.cpp` - Data/`main`/enlace por plataforma.

Notas del orador: Tener el editor listo en estos archivos para saltar rápido.

Q&A

- ¿Cómo extender el lenguaje? Añadir tokens/producciones/nodos/visitas/soporte en codegen.
- ¿Cómo portar a otra arquitectura? Nuevo backend (idealmente LLVM) o ASM objetivo.
- ¿Qué dependencias mínimas? `nasm` y `gcc/MinGW` para los binarios generados.

Notas del orador: Cierre mostrando el ejecutable final en `bin/` y el mensaje de éxito.