AymaraLang (aym) y su compilador aymc

Notas del orador: Presentación de alto nivel del lenguaje AymaraLang y del compilador nativo que genera ejecutables x86_64 para Linux y Windows.

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: Resalte el propósito cultural/educativo y el alcance técnico: sin dependencias pesadas, pipeline completo clásico.

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: Muestre el flujo completo y dónde se materializa cada artefacto en disco.

CLI y Flujo (main)

- Archivo: `compiler/main.cpp:15`.
- 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: Destaque el modo REPL para demos rápidas.

Léxico (Lexer)

- Archivos: `compiler/lexer/lexer.h:69`, `compiler/lexer/lexer.cpp:8`.
- 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: Comente el soporte de Unicode simple y los tokens multi-palabra sin romper posición.

Parser y AST

- Archivos: `compiler/parser/parser.h:13`, `compiler/parser/parser.cpp:8`.
- AST: `compiler/ast/ast.h:34` 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)` → init/cond/post.

Notas del orador: Muestre cómo un `willt'aña(1);` genera `PrintStmt(NumberExpr(1))`.

Análisis Semántico

- Archivos: `compiler/semantic/semantic.h:12`, `compiler/semantic/semantic.cpp:35`.
- Escopos/Tipos: declaraciones, uso, verificación simple de tipos, contextos de control.
- Builtins: `compiler/builtins/builtins.h:1`, `compiler/builtins/builtins.cpp:1`.
- Exporta a CodeGen: globales, tipos globales, tipos de parámetros.

Notas del orador: Explique la inferencia básica de strings en parámetros por uso.

Generación de Código (ASM x86_64)

- Archivo: `compiler/codegen/codegen.cpp:734` (entrada).
- Datos: literales + formatos `fmt_int`/`fmt_str`; extern `printf/scanf/strlen` (`compiler/codegen/codegen.cpp:656`).
- 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: Subraye la alineación de pila y el manejo de 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:24`, `compiler/interpreter/interpreter.cpp:81`.
- Builtins: `willt'aña`, `input`, `length`.

Notas del orador: Buen modo 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: Muestre el artefacto `build/hola.asm` si 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: Comente cómo el lexer construye el token compuesto “jan uka”.

Multiplataforma

- Selección: `--windows` / `--linux`.
- Win64: RCX/RDX/R8/R9, shadow space (32B) (`compiler/codegen/codegen.cpp:233`).
- Linux: SysV y `-no-pie` (`compiler/codegen/codegen.cpp:725`).

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

Pruebas

- Samples automatizados: `tests/run_tests.sh:1`.
- Unit test básico: `tests/unittests/test_compiler.cpp:1`.
- Linux: `make test`.

Notas del orador: Explique que 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: Invite a contribuciones y a probar nuevos casos.

Referencias Rápidas

- `compiler/main.cpp:15` — Entrada CLI y orquestación.
- `compiler/lexer/lexer.cpp:8` — Tokenización.
- `compiler/parser/parser.cpp:8` — Construcción de AST.
- `compiler/semantic/semantic.cpp:35` — Análisis semántico.
- `compiler/codegen/codegen.cpp:656` — Sección `.data` y formatos.
- `compiler/codegen/codegen.cpp:679` — Emisión de `main`.
- `compiler/codegen/codegen.cpp:725` — Enlace por plataforma.

Notas del orador: Tenga 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.