



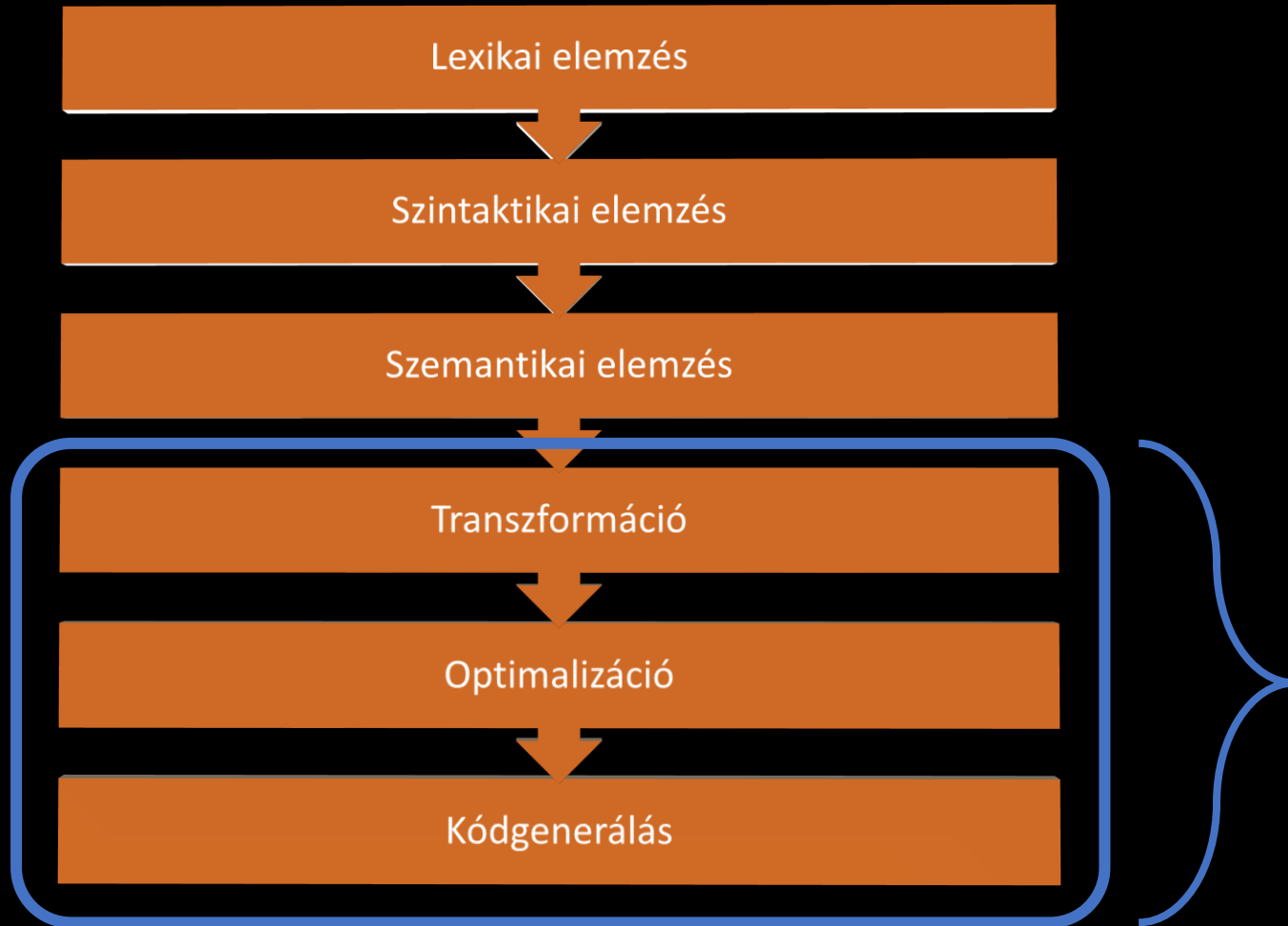
LLVM

Gembela Gergely

Mi is az LLVM?

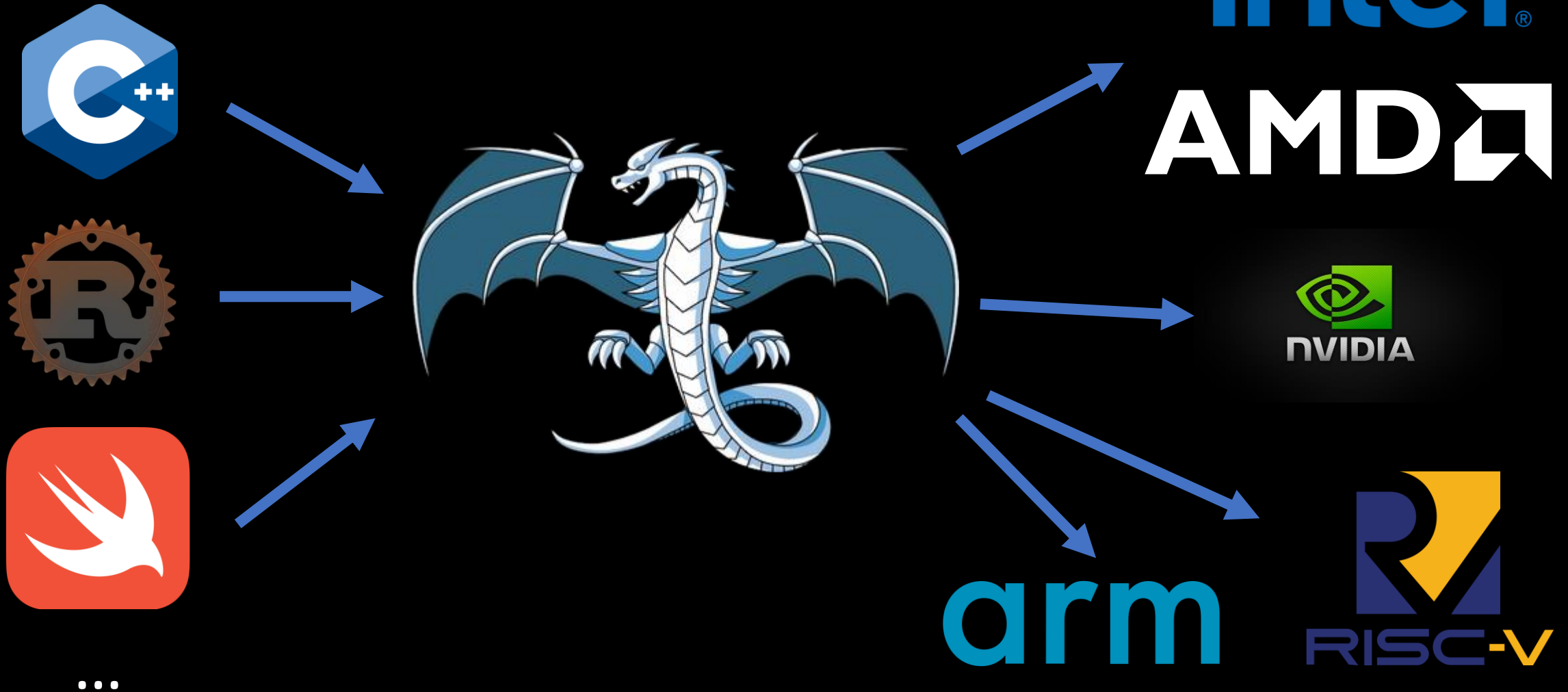
- Fordítók készítéséhez és teszteléséhez használható eszközök gyűjteménye (és már NEM a Low-Level Virtual Machine mozaikszava)
- Több népszerű nyelvhez van LLVM alapú fordító, vannak nyelvek amik kifejezetten LLVM alapú fordítóikkal együtt fejlődnek
 - Rust
 - Zig
 - Nim
 - Clang – C++, Objective C, C
 - Swift

Az LLVM feladatai egy fordítóban



Middle-end és backend: Az LLVM eszközei első sorban a fordítás ezen szakaszára adnak megoldást

Mit csinál az LLVM?



A sárkányon belül

- IR nyelv
 - Frontendfüggetlen*
 - Platformfüggetlen*
- Optimalizáció
 - Kiterjeszthető, személyre szabható lépések
 - Az IR nyelven elérhetők
- Platformspecifikus backendek
- Linker
 - Képes link-time optimalizációkra



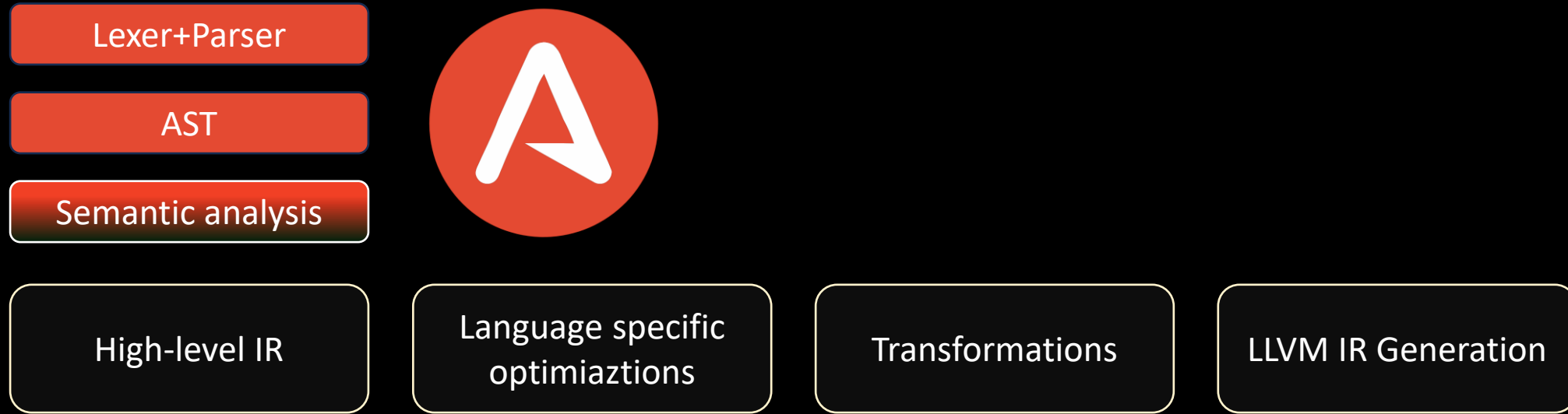
Egy modern LLVM alapú fordító

Lexer+Parser

AST



Egy modern LLVM alapú fordító



Egy modern LLVM alapú fordító

Lexer+Parser

AST

Semantic analysis



High-level IR

Language specific
optimiaztions

Transformations

LLVM IR Generation

Optimizations

Link-time
optimizations (lld)

Target-specific
backend



LLVM IR

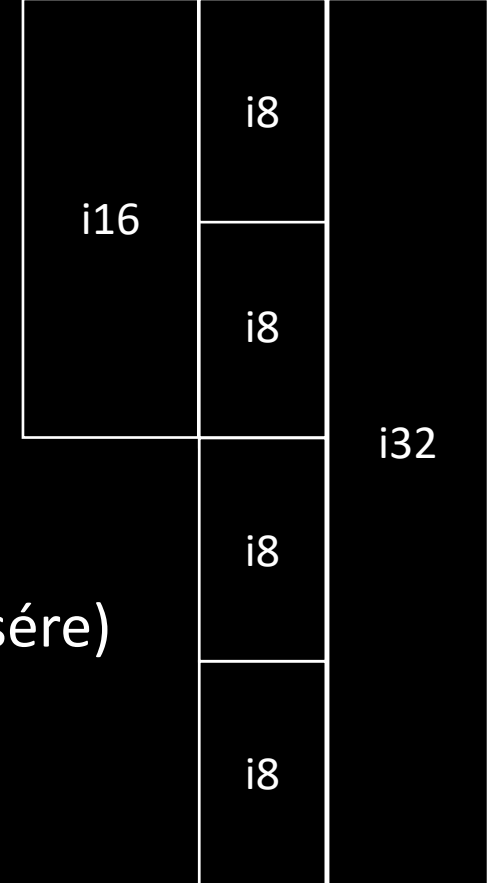
- A legtöbb modern fordítóban van megfelelője
 - GCC RTL
 - .A NET IL + Roslyn hasonló, de más célok
 - Kifejezetten .NET a CLR számára készült
 - Objektorientált, típusos nyelv
 - A .NET IL tároló is, biztosítja a bináris kompatibilitást
 - LLVM IR-t jellemzően nem terjesztenek a toolchainek (de link-time optimization miatt még ez is előfordulhat)
 - A .NET IL stabil, és tényleg platformfüggetlen
 - Java bytecode
 - SPIR-V (shaderesetén)

LLVM IR

- 3 alak – bitcode, in-memory IR, szöveges
- Az LLVM IR típusos
 - Régebben még a pointerek is, ez mostanra eltűnőben van (szerencsére)
- Definiálhatók saját típusok
 - Ezek leginkább a C struktúrákhoz hasonlítanak, DE alignment!
- 2 féle ID
 - @global – globális
 - %local – lokális
- A kódot modulokba szervezi

LLVM IR

- 3 formában használjuk – bitcode, in-memory IR, szöveges
- Az LLVM IR típusos `%struct.vec4 = type { i16, i16, i16, i16 }`
 - Régebben még a pointerek is, ez mostanra eltűnőben van (szerencsére)
- Definiálhatók saját típusok
 - Ezek leginkább a C struktúrákhoz hasonlítanak, DE alignment!
- 2 féle ID – minden elemhez (változó, függvény, modul)
 - @global – globális
 - %local – lokális
- A kódot modulokba szervezi (egy fordítási egység, pl. egy .cpp file)
 - Jelenleg csak modulon belüli optimalizációkkal foglalkozunk



@square – C++

```
int square(int num) {  
    return num * num;  
} //C/C++
```

```
define i32 @square(int)(i32 @noundef  
%0) {  
    %2 = mul nsw i32 %0, %0  
    ret i32 %2  
} ;LLVM IR
```

@square – rs

```
pub fn square(num: i32) -> i32 {  
    num * num  
} //RS
```

```
define i32 @square(int)(i32 noundef  
%0) {  
    %2 = mul nsw i32 %0, %0  
    ret i32 %2  
} ;LLVM IR
```

@square – Swift

```
func square(n: Int) -> Int {  
    return n * n  
} //Swift
```

```
define i32 @square(int)(i32 noundef  
%0) {  
    %2 = mul nsw i32 %0, %0  
    ret i32 %2  
} ;LLVM IR
```

@square – llc (debug/release/ssa/...)

```
define dso_local noundef i32
@square(int)(i32 noundef %0) #0 !dbg !10 {
    %2 = alloca i32, align 4
    store i32 %0, ptr %2, align 4
    tail call void @llvm.dbg.declare(metadata
ptr %2, metadata !16, metadata
!DIExpression()), !dbg !17
    %3 = load i32, ptr %2, align 4, !dbg !18
    %4 = load i32, ptr %2, align 4, !dbg !19
    %5 = mul nsw i32 %3, %4, !dbg !20
    ret i32 %5, !dbg !21
}
```

```
define i32 @square(int)(i32 noundef
%0) {
    %2 = mul nsw i32 %0, %0
    ret i32 %2
} ;LLVM IR
```

@square – a kimenet

```
square:
  # x86-64
  mov    eax, edi
  imul   eax, eax
  ret
```

```
square(int): # WASM
    local.get    0
    local.get    0
    i32.mul
    end_function
```

```
define i32 @square(int)(i32 noundef %0)
{
    %2 = mul nsw i32 %0, %0
    ret i32 %2
} ;LLVM IR
```

```
square: //arm64
mul    w0, w0, w0
ret
```

```
square: # RISC-V
mul    a0, a0, a0
ret
```


@érdekesség – nvcc

```
.visible .entry square(int*, int)(  
    .param .u64 square(int*, int)_param_0,  
    .param .u32 square(int*, int)_param_1  
)  
{  
  
    ld.param.u64    %rd1, [square(int*, int)_param_0];  
    ld.param.u32    %r1, [square(int*, int)_param_1];  
    cvta.to.global.u64    %rd2, %rd1;  
    mul.lo.s32      %r2, %r1, %r1;  
    st.global.u32   [%rd2], %r2;  
    ret;  
  
}
```



LLVM Pass (IR->IR)

- Kód2kód transzformáció
- Akár mi is írhatunk
- De van *néhány* ami már meg van írva, és nyelvfüggetlenül elérhető
 - Érdekesség: ennyire egyszerű egy saját pass-t készíteni

```
#include "llvm/IR/PassManager.h"
```

```
namespace llvm {
```

```
class HelloWorldPass : public PassInfoMixin<HelloWorldPass> {  
public:  
    PreservedAnalyses run(Function &F, FunctionAnalysisManager &AM);  
};
```

```
} // namespace llvm
```

Optimalizálás LLVM segítségével

- Az összes tanult optimalizáló algoritmus elérhető, és akár saját implementáció is készíthető rájuk
- Sok pass nem része az alapértelmezetten engedélyezett szinteknek (Itt a szintek Pl. O[0-3], Ofast, Os, Oz)
- Nem minden optimalizáció platformfüggetlen!
 - Pl. SIMD instrukciók generálása (SSE/AVX, Arm Neon,)
- Egy később futó pass visszavonhatja egy előtte futó transzformációját
- A passek külön is tesztelhetők az opt eszközzel, pl.
 - `opt -passes='loop-unroll'`
 - `opt -passes='dce'`

Az optimalizáló pipeline megtekintése

The image shows a screenshot of the Visual Studio Code editor interface. On the left, a C++ source file named 'C++ source #1' is open, displaying a factorial function and a main function that calls it with the value 14. On the right, the 'armv8-a clang (trunk) (Editor #1)' window is open, showing the compiler options '-O2 -emit-llvm'. A context menu is displayed over the compiler options, listing various LLVM tools and optimization options. The 'Opt Pipeline' option is highlighted with a red underline. The background of the right window shows LLVM IR code.

```
#include <stdio.h>

inline long factorial(long n){
    long res = 1;
    for(long i = 2; i <= n; i++){
        res *= i;
    }
    return res;
}

int main() {
    volatile auto a = factorial(14);
}
```

armv8-a clang (trunk) (Editor #1) -O2 -emit-llvm

- Clone Compiler
- Optimization
- Stack Usage
- Preprocessor
- AST
- LLVM IR
- Opt Pipeline
- Device
- Control Flow Graph

```
ef i32 @main() local_unnamed_addr #0 !dbg
    gn 8
    .dbg.assign(metadata i1 undef, metadata !
time.start.p0(i64 8, ptr nonnull %a), !dbg
7178291200, ptr %a, align 8, !dbg !23
    .dbg.assign(metadata i64 poison, metadata
time.end.p0(i64 8, ptr nonnull %a), !dbg

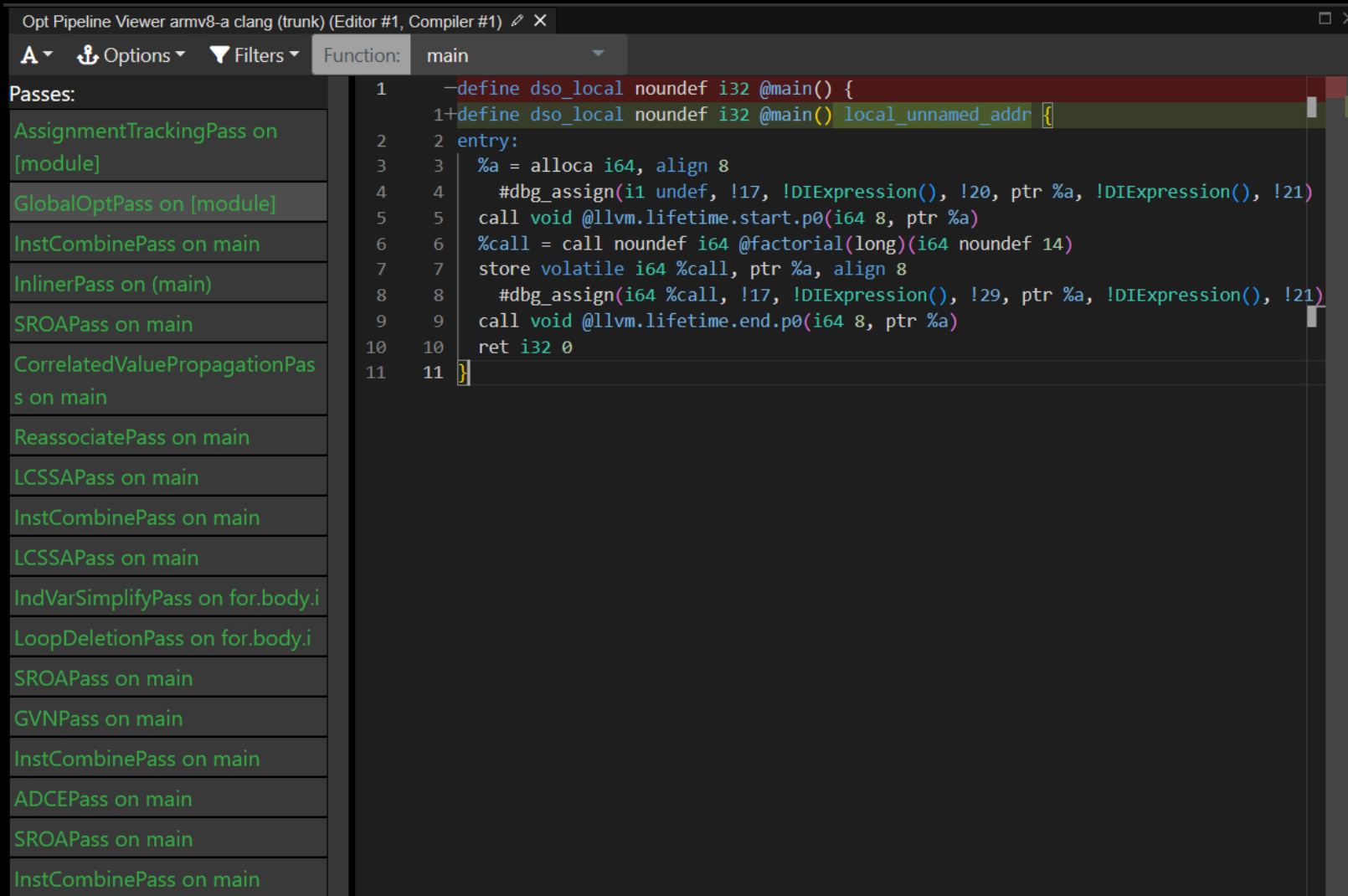
etime.start.p0(i64 immarg, ptr nocapture)
etime.end.p0(i64 immarg, ptr nocapture) #

declare void @llvm.dbg.assign(metadata, metadata, metadata, met

attributes #0 = { mustprogress norecurse nounwind memory
attributes #1 = { mustprogress nocallback norecurse nounwind
attributes #2 = { nocallback norecurse nounwind speculatable

!20 = distinct !DIAssignID()
!28 = distinct !DIAssignID()
```

Az optimalizáló pipeline megtekintése



The screenshot displays the LLVM Opt Pipeline Viewer for the 'main' function. The interface is divided into two main sections: a list of optimization passes on the left and the corresponding LLVM IR code on the right.

Passes:

- AssignmentTrackingPass on [module]
- GlobalOptPass on [module]
- InstCombinePass on main
- InlinerPass on (main)
- SROAPass on main
- CorrelatedValuePropagationPass on main
- ReassociatePass on main
- LCSSAPass on main
- InstCombinePass on main
- LCSSAPass on main
- IndVarSimplifyPass on for.body.i
- LoopDeletionPass on for.body.i
- SROAPass on main
- GVNPass on main
- InstCombinePass on main
- ADCEPass on main
- SROAPass on main
- InstCombinePass on main

Function: main

```
1  -define dso_local noundef i32 @main() {  
1+define dso_local noundef i32 @main() local_unnamed_addr {  
2  2 entry:  
3  3   %a = alloca i64, align 8  
4  4   #dbg_assign(i1 undef, !17, !DIExpression(), !20, ptr %a, !DIExpression(), !21)  
5  5   call void @llvm.lifetime.start.p0(i64 8, ptr %a)  
6  6   %call = call noundef i64 @factorial(long)(i64 noundef 14)  
7  7   store volatile i64 %call, ptr %a, align 8  
8  8   #dbg_assign(i64 %call, !17, !DIExpression(), !29, ptr %a, !DIExpression(), !21)  
9  9   call void @llvm.lifetime.end.p0(i64 8, ptr %a)  
10 10  ret i32 0  
11 11 }
```

Példa: dead code elimination

```
long dce(long n){  
    long res = 1;  
    auto c = res - 3;  
    return res;  
}
```

Példa: Mem2Reg (valójában SROA+Mem2Reg)

```
long sroa(long n){  
    long res = 1;  
    return res + n;  
}
```

Példa: loop átalakítása (InvVarSimplify, deletion)

```
int whatever42(int a){  
    int x = 3;  
    for(int i = 0; i < a; i++){  
        int c = a*a;  
        x += c;  
    }  
    return x;  
}
```


Példa: constant inlining

```
inline long factorial(long n){
    long res = 1;
    for(long i = 2; i <= n; i++){
        res *= i;
    }
    return res;
}

int main() {
    volatile auto a = factorial(14);
}
```

Példa: loop unroll

```
void vecadd(float a[4], float b[4]){  
    for(int i = 0; i < 4; i++){  
        a[i] = b[i];  
    }  
}
```

Extrém példa: dead code elimination #2

```
fn square(num: i32) -> i32 {  
    num * num  
} //RS
```

Extrém példa: dead code elimination #2

```
fn square(num: i32) -> i32 {  
    num * num  
}
```

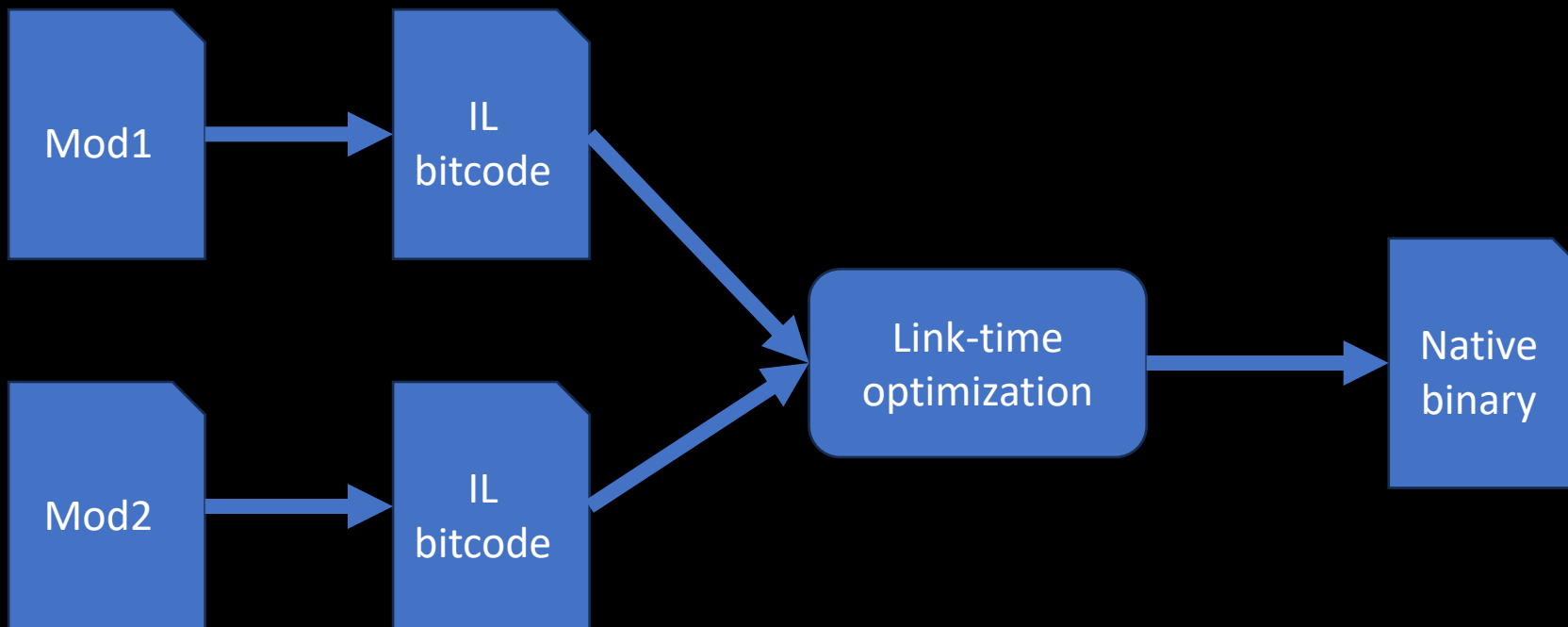


;igen, ez egy üres textbox :D

Mivel a függvény nem publikus és nincsen felhasználva, a fordító egyszerűen megszabadul tőle...

Link-time optimalizáció

- A modulok szintjénél magasabb szintű, a teljes programot fedő optimalizáció
- Fordításidő szempontjából jelentős többletet jelenthet, nehezen párhuzamosítható



Ki használja?



SONY



493,156 Commits

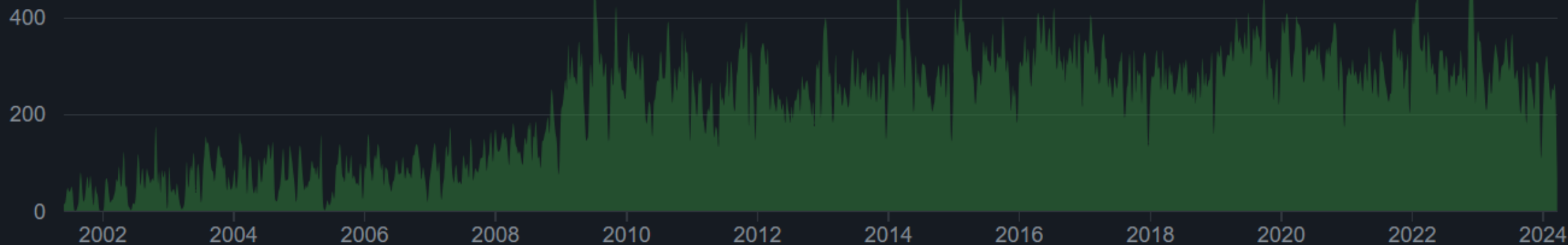
Stats



493,156 Commits

Jun 3, 2001 – Mar 26, 2024

Contributions to main, line counts have been omitted because commit count exceeds 10,000.



Köszönöm a figyelmet!

Források, hasznos anyagok

[The Architecture of Open Source Applications \(Volume 1\)LLVM \(aosabook.org\)](#)

[LLVM IR and Go | Gopher Academy Blog](#)

[Using the New Pass Manager — LLVM 19.0.0git documentation](#)

[Compiling With Clang Optimization Flags – Incredibuild](#)

LLVM IR:

[LLVM Assembly Language Reference Manual \(apple.com\)](#)

Videók:

[LLVM in 100 Seconds \(youtube.com\)](#)

[\(5\) 2023 EuroLLVM - Tutorial: A whirlwind tour of the LLVM optimizer – YouTube](#)