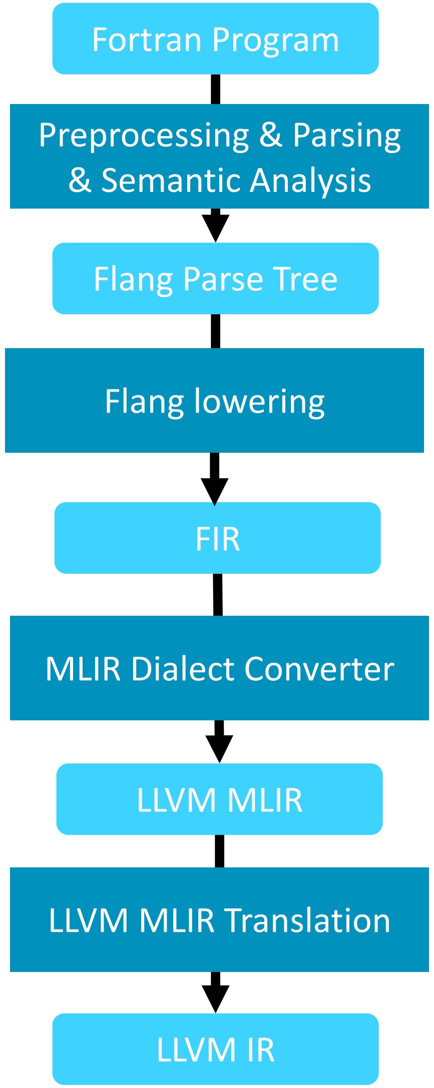
Introduction to the Flang Frontend : Tutorial

2021 LLVM Developers Meeting

Kiran Chandramohan

19 Nov 2021

Overview/High Level Flow



- The tutorial will follow this flow
  - Will go through each stage with examples
  - Will not cover Driver/Runtime/OpenMP
- Traditional Compiler Flow
  - Preprocessing, Parsing, Semantic Checks, and lowering to an IR
  - Flang frontend lowers to LLVM IR
  - Difference with Clang
    - Clang lowers from AST to LLVM IR
    - Flang has a high-level IR : FIR
  - Uses MLIR infrastructure for FIR
    - MLIR interfaces with LLVM IR through the LLVM Dialect
    - FIR lowers to LLVM Dialect

A quick Fortran introduction

- Fortran is a language popular with the Scientific and HPC community
- Fortran is the first commercially available general-purpose language
  - First appeared in the 1950s at IBM
- Fortran has great support for arrays and floating-point numbers
- Code written over several decades continue to work
- Fortran language continues to develop
  - Modern language with support for Modules, OOP, parallel features
  - Latest revision was in Fortran 2018
  - Another revision due next year
- Having a Fortran frontend is important for LLVM to be successful in HPC
- An open-source Fortran compiler with a permissive license can aid standardization
- A good Fortran compiler and tools can help scientists write better code

Fortran : First programs

Hello World

```
PROGRAM main
  write (*,*) “Hello World!”
END PROGRAM main
```

Fortran subroutine & sections

```
subroutine sb(x)
  ! Declaration section
  real :: x
  ! Executable section
  x =  1
  print *, x
end subroutine sb
```

Fortran : Fixed vs Free Form

Fixed Form

```
      PROGRAM main
      real :: x
c This is a comment
      x =  1
      print *, “Hello World!”
      END PROGRAM main
```

- For punched card machines
- Normal instructions from column 7 – 72
- c in column 1 indicates a comment
- Columns 1-5 can have labels
- Column 6 is for indication line continuation

Free Form

```
PROGRAM main
  real::x
  x = 1
  print *,"Hello World!"
END PROGRAM main
```

- No such restrictions

Fortran : Modules

Module declaration

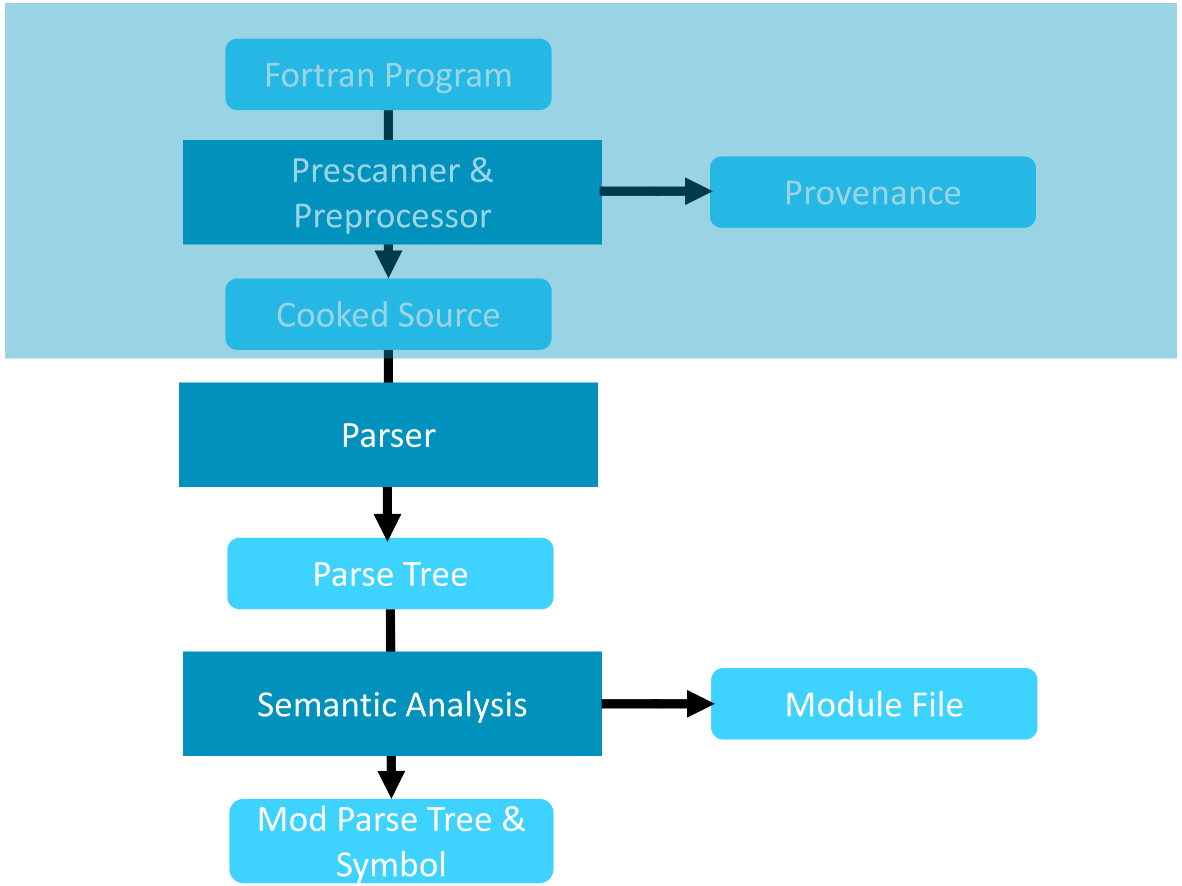
```
module one
  integer :: counter
contains
  subroutine incrementer()
    counter = counter + 1
  end subroutine
end module
```

Use of Module

```
program main
  use one
  write (*,*) counter
  call incrementer()
  write (*,*) counter
end program
```

Flang Preprocessing

- Prescanner generates cooked character stream
  - Calls Preprocessor to expand macros
  - Normalize source : all lower case, free form, includes files
  - Hides complexity from rest of compiler
- Provenance
  - Maps maintained from cooked source location to original source location
  - Provide good debug and location information



Flang Preprocessing

Fortran Source

(file.f)

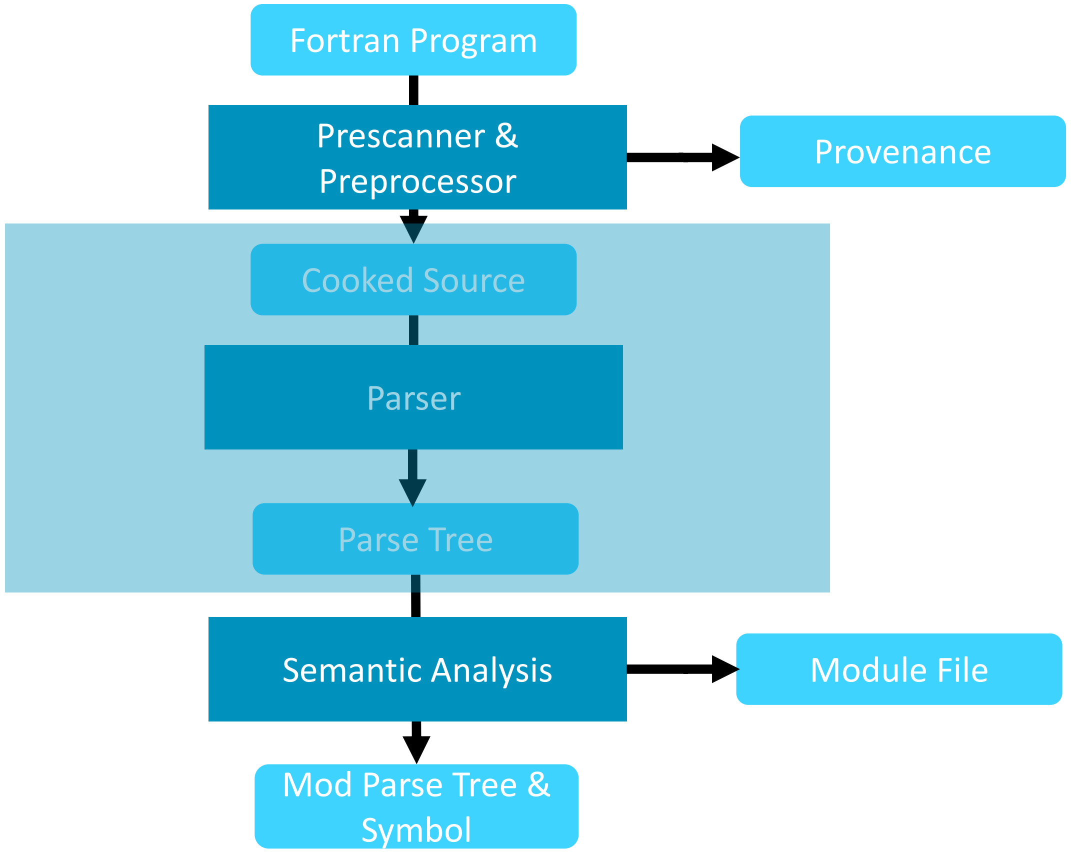
```
#define MESSAGE "Hello World!"
      PROGRAM main
      real :: x
c This is a comment
      x =  1
      print *, MESSAGE
      END PROGRAM main
```

Cooked character stream

(flang-new –fc1 –E -fnoreformat file.f)

```
programmain
real::x
x=1
print*,"Hello World!"
endprogrammain
```

Flang Parsing



Flang Parsing

- Recursive Descent Parsing
- Grammar taken from standard and suitably modified
  - Left recursion removed
- Uses Parser combinators
  - Token parser
  - Operators & functions to combine parsers
- Parser is written in a declarative style

```
!Fortran source
integer :: a = 1, b = 2
integer :: arr(3) = (/1, 2, 3/)
Character :: name*10
```

// 2018 standards document
// R801 type-declaration-stmt ->
// declaration-type-spec [[, attr-spec]... ::] entity-decl-list
// R803 entity-decl ->
// object-name [(array-spec)] [lbracket coarray-spec rbracket]
// [* char-length] [initialization]

//lib/parser/Fortran-parsers.cpp

```
PARSER(construct<EntityDecl>(objectName,
maybe(arraySpec), maybe(coarraySpec),
maybe ("*" >> charLength),
maybe(initialization))
```

Flang Parsing

Fortran Source

(file.f95)

```fortran
function add(x,y) result(z)

    integer :: x, y, z

    z = x + y

end function
```

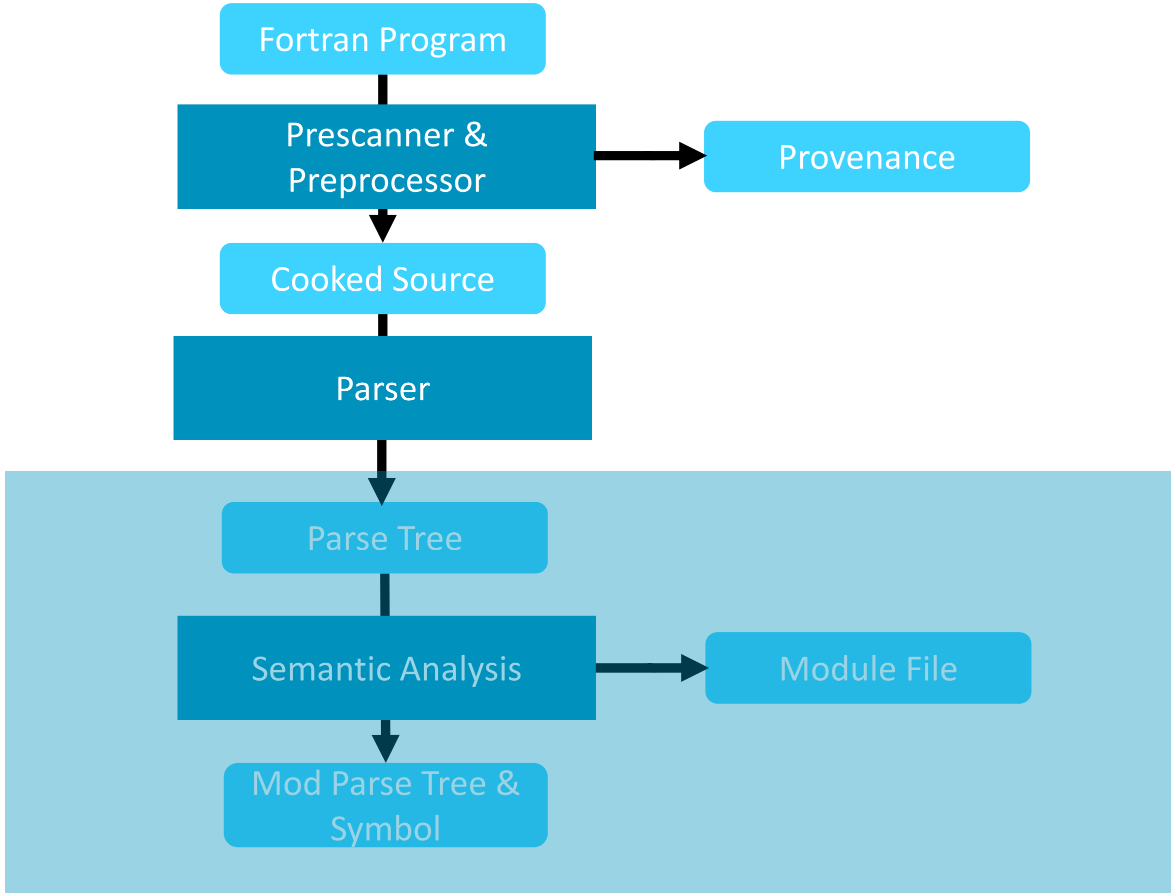
Parse Tree

(flang-new –fc1 –fdebug-dump-parse-tree file.f95)

```
Program -> ProgramUnit -> FunctionSubprogram
| FunctionStmt
| | Name = 'add'
| | Name = 'x'
| | Name = 'y'
| | Suffix
| | | Name = 'z'
| SpecificationPart
| | ImplicitPart ->
| | DeclarationConstruct -> SpecificationConstruct ->
TypeDeclarationStmt
| | | DeclarationTypeSpec -> IntrinsicTypeSpec ->
IntegerTypeSpec ->
| | | EntityDecl
| | | | Name = 'x'
| | | EntityDecl
| | | | Name = 'y'
| | | EntityDecl
| | | | Name = 'z'
| ExecutionPart -> Block
| | ExecutionPartConstruct -> ExecutableConstruct -> ActionStmt
-> AssignmentStmt = 'z=x+y'
| | | Variable = 'z'
| | | | Designator -> DataRef -> Name = 'z'
| | | Expr = 'x+y'
| | | | Add
| | | | | Expr = 'x'
| | | | | | Designator -> DataRef -> Name = 'x'
| | | | | Expr = 'y'
| | | | | | Designator -> DataRef -> Name = 'y'
| EndFunctionStmt ->
```

Flang Semantic Analysis

- Checks the rules/constraints mentioned in the standard
  - Expression and Statement Semantic Checks
  - Label resolution (Checks gotos)
  - Name resolution (Checks names, Symbol)
- Modifies parse tree if ambiguous
- Constant Expression evaluation
- Emits Module files



Flang Semantics : Checks

Fortran Source

(file.f95)

```
subroutine check(c)
    real :: c
    select case (c)
       case (1.0)
       case (2.0)
       case default
    end select
end subroutine
```

Parse Tree

(flang-new –fc1 –fsyntax-only file.f95)

```
error: Semantic errors in case4.f90
./case4.f90:3:16: error: SELECT CASE expression must be
integer, logical, or character
    select case (c)
                ^
```

Semantic Analysis : Rewriting parse-tree

Fortran Source

(file.f95)

```
  integer :: g(10)
  f(i) = i + 1
  g(i) = i + 2
end
```

Incorrect parse-tree

(./bin/flang-new -fc1 -fdebug-dump-parse-tree-no-sema file.f95)

```
| | DeclarationConstruct -> StmtFunctionStmt
| | | Name = 'g'
| | | Name = 'i'
| | | Scalar -> Expr -> Add
| | | | Expr -> Designator -> DataRef -> Name = 'i'
| | | | Expr -> LiteralConstant -> IntLiteralConstant = '2'
```

Corrected parse-tree

(./bin/flang-new -fc1 -fdebug-dump-parse-tree file.f95)

```
| ExecutionPart -> Block
| | ExecutionPartConstruct -> ExecutableConstruct -> ActionStmt -> AssignmentStmt = 'g(int(i,kind=8))=i+2_4'
| | | Variable = 'g(int(i,kind=8))'
| | | | Designator -> DataRef -> ArrayElement
```

Flang Module Format

- Modules will be stored as Fortran source
  - Module files will contain a header
    - Magic string, Version, Checksum
  - The body will contain declarations of all user visible entities
- Reading module files is fast
  - Fast parser, No pre-processing necessary

Fortran Source

```
!mymod.f90
module vars
integer :: a
real :: b
contains
   subroutine add_val_a(x)
      integer :: x
      a = a + x
   end subroutine
end module
```

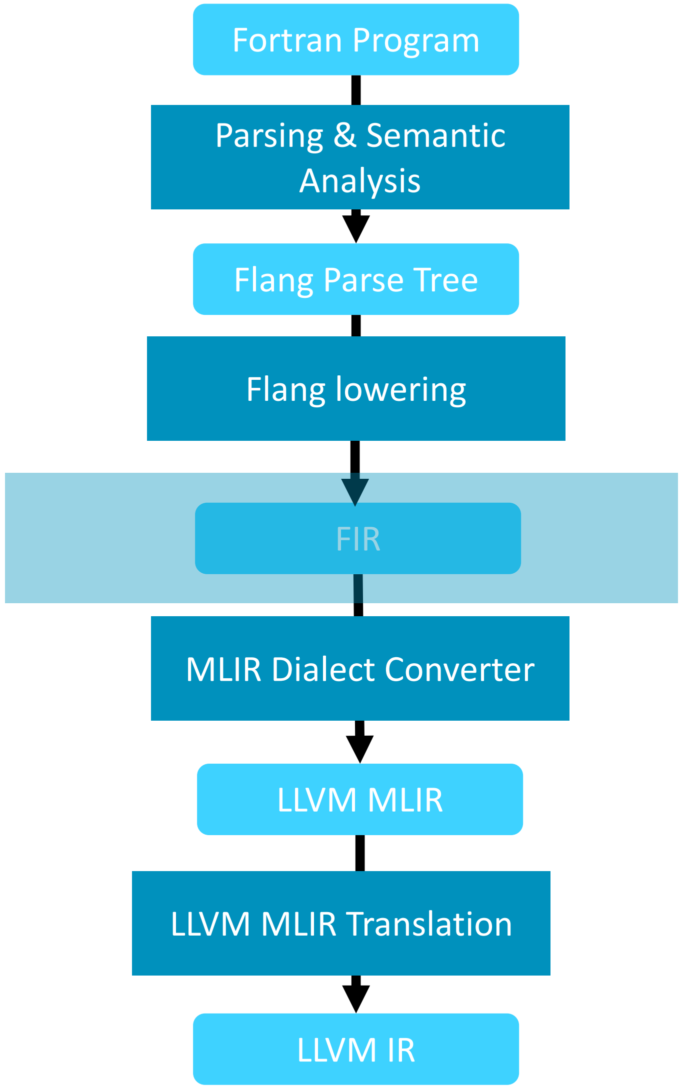
Module File

```
!vars.mod
!mod$ v1 sum:672b5185d5193446
module vars
integer(4)::a
real(4)::b
contains
subroutine add_val_a(x)
integer(4)::x
end
end
```

FIR

FIR

- Skipping the lowering
  - Traverse the parse-tree and generate FIR



FIR

- Fortran IR (FIR) is built using MLIR
- Types and Operations which model Fortran
  - Types for Arrays, Strings, Derived Types
  - Higher level operations for arrays, loops etc
- All names are uniqued by mangling before lowering to FIR
- Transformation passes on FIR
  - CSE, simplifies higher-level operations to lower-level operations
- Tools
  - Use the –emit-fir option of the driver for emitting FIR
    - ./bin/flang-new -emit-fir file.f95
  - fir-opt is the tool for FIR related MLIR transformations

FIR: Do Loop

Structured Do loop

```
do i = 1, N
  res = res + i
end do
```

Unstructured Do loop

```
do i = 1, N
  if (res .gt. 0) then
    exit
  else
    res = res + i
  end if
end do
```

FIR: Do Loop

- fir.do_loop operation in FIR models Fortran Do Loops.
  - Similar to scf.for loops (in the scf MLIR dialect)
- All structured Fortran Do loops are converted to fir.do_loop
- Later converted to affine loops (if possible) for optimizations
- fir.do_loop is not created for unstructured loops
  - i.e if there are jumps or branches or exits
  - A control-flow-graph structure is generated for these loops

FIR: Do Loop (Fortran -> FIR)

```fortran
function sumN(N) result(res)
  integer :: N
  integer :: res
  res = 0
  do i = 1, N
    res = res + i
  end do
end function
```

```
func @_QPsumn(%arg0: !fir.ref<i32>) -> i32 {
  %0 = fir.alloca i32 {uniq_name = "_QFsumnEi"}
  %1 = fir.alloca i32 {uniq_name = "_QFsumnEres"}
  %c0_i32 = constant 0 : i32
  fir.store %c0_i32 to %1 : !fir.ref<i32>
  %c1_i32 = constant 1 : i32
  %2 = fir.convert %c1_i32 : (i32) -> index
  %3 = fir.load %arg0 : !fir.ref<i32>
  %4 = fir.convert %3 : (i32) -> index
  %c1 = constant 1 : index
  %5 = fir.do_loop %arg1 = %2 to %4 step %c1 -> index{
    %8 = fir.convert %arg1 : (index) -> i32
    fir.store %8 to %0 : !fir.ref<i32>
    %9 = fir.load %1 : !fir.ref<i32>
    %10 = fir.load %0 : !fir.ref<i32>
    %11 = addi %9, %10 : i32
    fir.store %11 to %1 : !fir.ref<i32>
    %12 = addi %arg1, %c1 : index
    fir.result %12 : index
  }
  // convert and store %5 into %0
  %7 = fir.load %1 : !fir.ref<i32>
  return %7 : i32
}
```

FIR: Array ops

- Fortran allows array expressions
- Example shows the summation of two arrays
- FIR models this by a few operations

```
subroutine add(a,b,c,n)
  integer :: n
  real,intent(out)::a(n)
  real,intent(in)::b(n)
  real,intent(in)::c(n)
  a = b + c
end subroutine add
```

Operations	Description
fir.array_load	loads an array into an SSA value
fir.array_fetch	fetches an element
fir.array_update	Updates the element
fir.array_merge_store	Stores the updated array

FIR: Array ops

```
subroutine add(a,b,c,n)
  integer :: n
  real,intent(out)::a(n)
  real,intent(in)::b(n)
  real,intent(in)::c(n)
  a = b + c
end subroutine add
```

```
  func @_QPadd(%arg0: !fir.ref<!fir.array<?xf32>>, %arg1:
!fir.ref<!fir.array<?xf32>>, %arg2: !fir.ref<!fir.array<?xf32>>, %arg3:
!fir.ref<i32>) {
    %c0 = arith.constant 0 : index
    %c1 = arith.constant 1 : index
    %0 = fir.load %arg3 : !fir.ref<i32>
    %1 = fir.convert %0 : (i32) -> index
    %2 = fir.shape %1 : (index) -> !fir.shape<1>
    %3 = fir.array_load %arg0(%2) : (!fir.ref<!fir.array<?xf32>>,
!fir.shape<1>) -> !fir.array<?xf32>
    %4 = fir.array_load %arg1(%2) : (!fir.ref<!fir.array<?xf32>>,
!fir.shape<1>) -> !fir.array<?xf32>
    %5 = fir.array_load %arg2(%2) : (!fir.ref<!fir.array<?xf32>>,
!fir.shape<1>) -> !fir.array<?xf32>
    %6 = arith.subi %1, %c1 : index
    %7 = fir.do_loop %arg4 = %c0 to %6 step %c1 unordered iter_args(%arg5
= %3) -> (!fir.array<?xf32>) {
      %8 = fir.array_fetch %4, %arg4 : (!fir.array<?xf32>, index) -> f32
      %9 = fir.array_fetch %5, %arg4 : (!fir.array<?xf32>, index) -> f32
      %10 = arith.addf %8, %9 : f32
      %11 = fir.array_update %arg5, %10, %arg4 : (!fir.array<?xf32>, f32,
index) -> !fir.array<?xf32>
      fir.result %11 : !fir.array<?xf32>
    }
    fir.array_merge_store %3, %7 to %arg0 : !fir.array<?xf32>,
!fir.array<?xf32>, !fir.ref<!fir.array<?xf32>>
    return
  }
```

FIR: Array ops

```
subroutine assign(a)
    integer :: a(4)
    a(2:4) = a(1:3)
end subroutine
```

- Array expression evaluation requires insertion of a copy
- In the example if the contents of array `a` are 1, 2, 3, 4
- The contents of array `a` after evaluation will be
  - 1, 1, 1, 1 without a temporary
  - 1, 1, 2, 3 with a temporary
  - The latter is expected
- An array value copy pass inspects the array updates and checks for conflicts:
  - if there is an array update to an array value (x) from the same memory reference as another array value (y) but with different shapes.
  - If there is a conflict a temporary is inserted
  - Pass replaces array_load, array_fetch, array_update and array_merge_store with normal loads and stores and loops

FIR: Array ops

Fortran source

(file.f95)

```fortran
subroutine assign(a)
    integer :: a(4)
    a(2:4) = a(1:3)
end subroutine
```

FIR

(./bin/flang-new -fc1 -emit-fir file.f95)

```
func @_QPassign(%arg0: !fir.ref<!fir.array<4xi32>>) {
    // %c0, %c1, %c2, %c4 : constants 0, 1, 2, 4 of index type
    // %c1_i64, %c2_i64, %c3_i64, %c4_i64  : constants 1, 2, 3, 4 of i64 type
    %0 = fir.shape %c4 : (index) -> !fir.shape<1>
    %1 = fir.slice %c2_i64, %c4_i64, %c1_i64 : (i64, i64, i64) -> !fir.slice<1>
    %2 = fir.array_load %arg0(%0) [%1] : (!fir.ref<!fir.array<4xi32>>,
!fir.shape<1>, !fir.slice<1>) -> !fir.array<4xi32>
    %3 = fir.slice %c1_i64, %c3_i64, %c1_i64 : (i64, i64, i64) -> !fir.slice<1>
    %4 = fir.array_load %arg0(%0) [%3] : (!fir.ref<!fir.array<4xi32>>,
!fir.shape<1>, !fir.slice<1>) -> !fir.array<4xi32>
    %5 = fir.do_loop %arg1 = %c0 to %c2 step %c1 unordered iter_args(%arg2 = %2) -
(!fir.array<4xi32>) {
      %6 = fir.array_fetch %4, %arg1 : (!fir.array<4xi32>, index) -> i32
      %7 = fir.array_update %arg2, %6, %arg1 : (!fir.array<4xi32>, i32, index) ->
!fir.array<4xi32>
      fir.result %7 : !fir.array<4xi32>
    }
    fir.array_merge_store %2, %5 to %arg0[%1] : !fir.array<4xi32>,
!fir.array<4xi32>, !fir.ref<!fir.array<4xi32>>, !fir.slice<1>
    return
  }
```

FIR: Array ops

Fortran source

(file.f95)

```fortran
subroutine assign(a)
    integer :: a(4)
    a(2:4) = a(1:3)
end subroutine
```

FIR

(./bin/flang-new -fc1 -emit-fir file.f95 -o - | ./bin/fir-opt --array-value-copy --canonicalize --basic-cse)

```
%1 = fir.slice %c2_i64, %c4_i64, %c1_i64 : (i64, i64, i64) -> !fir.slice<1>
// Create a copy of the array a, tmp
%2 = fir.allocmem !fir.array<4xi32>
fir.do_loop %arg1 = %c0 to %c3 step %c1 {
}
// Copy a(1:3) to tmp(2:4)
%4 = fir.slice %c1_i64, %c3_i64, %c1_i64 : (i64, i64, i64) -> !fir.slice<1>
%5 = fir.do_loop %arg1 = %c0 to %c2 step %c1 unordered iter_args(%arg2 = %3)
-> (!fir.array<4xi32>) {
  %6 = arith.addi %arg1, %c1 : index
  %7 = fir.array_coor %arg0(%0) [%4] %6 : (!fir.ref<!fir.array<4xi32>>,
!fir.shape<1>, !fir.slice<1>, index) -> !fir.ref<i32>
  %8 = fir.load %7 : !fir.ref<i32>
  %9 = fir.array_coor %2(%0) [%1] %6 : (!fir.heap<!fir.array<4xi32>>,
!fir.shape<1>, !fir.slice<1>, index) -> !fir.ref<i32>
  fir.store %8 to %9 : !fir.ref<i32>
  fir.result %3 : !fir.array<4xi32>
}
// Copy tmp array to array a
fir.do_loop %arg1 = %c0 to %c3 step %c1 {
}
fir.freemem %2 : !fir.heap<!fir.array<4xi32>>
```


arm

FIR: Array Passing

- Fortran supports many kinds of arrays
- Sometimes the shape of the array is known at compile time
  - These can be passed as references to arrays
- Sometimes the shape is known only at runtime
  - An array descriptor (fat pointer) is needed for these
  - fir.embox creates an array descriptor

FIR: Fixed size Array

Fortran source

(file.f95)

```
subroutine sb
  interface
    subroutine arr_sum(arr)
      integer :: arr(100)
    end subroutine arr_sum
  end interface
  integer :: arr1(100)
  call arr_sum(arr1)
end subroutine sb
```

FIR

./bin/flang-new -fc1 -emit-fir file.f95 -o - | ./bin/fir-opt --canonicalize --basic-cse

```
func @_QPsb() {
  %0 = fir.alloca !fir.array<100xi32> {bindc_name = "arr1", uniq_name = "_QFsbEarr1"}
  fir.call @_QParr_sum(%0) : (!fir.ref<!fir.array<100xi32>>) -> ()
  return
}
```

FIR: Array Descriptor

Fortran source

file.f95

```
subroutine sb(n)
    interface
        subroutine arr_sum(arr)
          integer :: arr(:)
        end subroutine arr_sum
    end interface
    integer :: n
    integer :: arr1(n)
    call arr_sum(arr1)
end subroutine sb
```

FIR

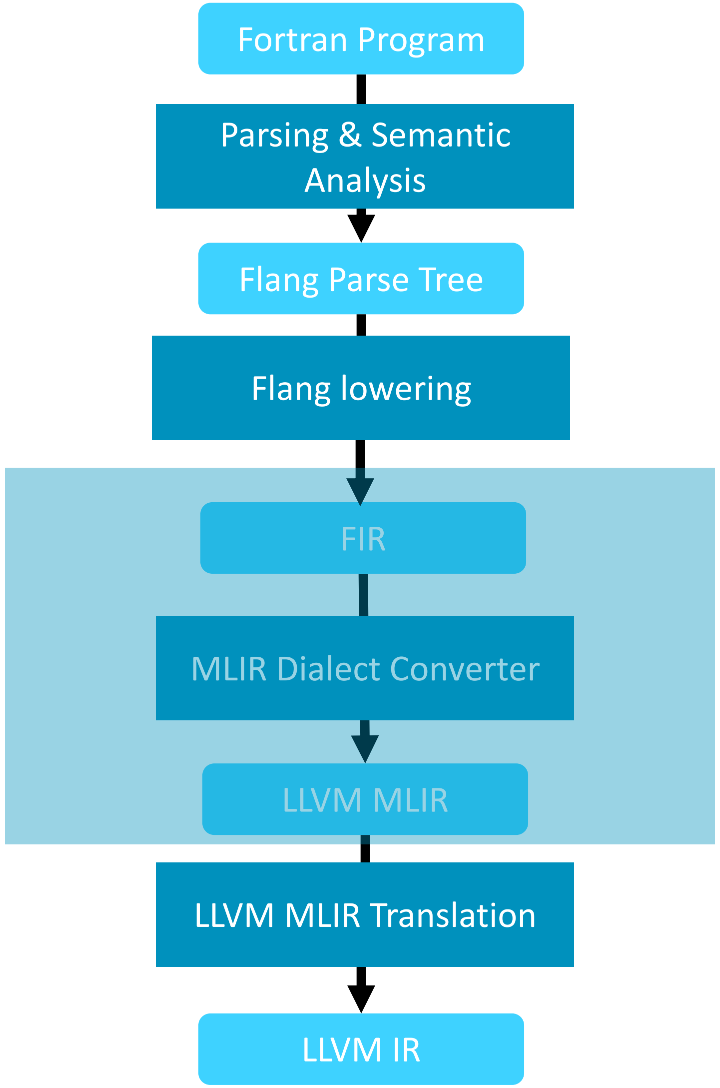
./bin/flang-new -fc1 -emit-fir file.f95 -o - | ./bin/fir-opt --canonicalize --basic-cse

```
func @_QPsb(%arg0: !fir.ref<i32>) {
  %0 = fir.load %arg0 : !fir.ref<i32>
  %1 = fir.convert %0 : (i32) -> index
  %2 = fir.alloca !fir.array<?xi32>, %1 {bindc_name = "arr1", uniq_name = "_QFsbEarr1"}

  %3 = fir.shape %1 : (index) -> !fir.shape<1>
  %4 = fir.embox %2(%3) : (!fir.ref<!fir.array<?xi32>>, !fir.shape<1>) -> !fir.box<!fir.array<?xi32>>
  fir.call @_QParr_sum(%4) : (!fir.box<!fir.array<?xi32>>) -> ()
  return
}
```

FIR Conversion to LLVM Dialect

Conversion to LLVM Dialect



FIR: Conversion to LLVM Dialect

- Add the conversion patterns
  - MLIR provides a systematic way for converting Operations
  - For each operation in FIR define a pattern that converts it to LLVM dialect operations
  - Not all FIR operations are convertable to LLVM dialect
  - Some (like the fir.do_loop) have been converted to CFG operations
- Specify which are the dialects/operations legal after conversion
  - In this case it is the llvm dialect
- Apply conversion
- fir-opt tool with --fir-to-llvm-ir option

FIR: Conversion to LLVM

```
void runOnOperation() override final {

  mlir::OwningRewritePatternList pattern;
  pattern.insert<
      AbsentOpConversion, AddcOpConversion, AddrOfOpConversion,
      AllocaOpConversion, AllocMemOpConversion, … ,
      CmpcOpConversion, …, ZeroOpConversion>(context, typeConverter);

  // …
  mlir::ConversionTarget target{*context};
  target.addLegalDialect<mlir::LLVM::LLVMDialect>();
  // …

  // apply the patterns
  if (mlir::failed(mlir::applyFullConversion(getModule(), target,
                                             std::move(pattern)))) {
    mlir::emitError(loc, "error in converting to LLVM-IR dialect\n");
    signalPassFailure();
  }
}
```

FIR: Conversion Patterns

FIR
(cplx.mlir)

```
func @cmp(%a : !fir.complex<4>,
          %b : !fir.complex<4>) -> i1 {
  %1 = fir.cmpc "oeq", %a, %b : !fir.complex<4>
  return %1 : i1
}
```

LLVM Dialect
(./bin/fir-opt --fir-to-llvm-ir cplx.mlir)

```
 llvm.func @cmp3(%arg0: !llvm.array<2 x f32>,
%arg1: !llvm.array<2 x f32>) -> i1 {
    %3 = llvm.load %2 : !llvm.ptr<struct<(f32,
f32)>>
    %7 = llvm.load %6 : !llvm.ptr<struct<(f32,
f32)>>
    %8 = llvm.extractvalue %7[0 : i32] :
!llvm.struct<(f32, f32)>
    %9 = llvm.extractvalue %3[0 : i32] :
!llvm.struct<(f32, f32)>
    %10 = llvm.fcmp "oeq" %8, %9 : f32
    %11 = llvm.extractvalue %7[1 : i32] :
!llvm.struct<(f32, f32)>
    %12 = llvm.extractvalue %3[1 : i32] :
!llvm.struct<(f32, f32)>
    %13 = llvm.fcmp "oeq" %11, %12 : f32
    %14 = llvm.and %10, %13  : i1
    llvm.return %14 : i1
  }
```

FIR: Conversion Patterns

```
struct CmpcOpConversion : public FIROpConversion<fir::CmpcOp> {
  using FIROpConversion::FIROpConversion;
  mlir::LogicalResult
  matchAndRewrite(fir::CmpcOp cmp, OperandTy operands,
                  mlir::ConversionPatternRewriter &rewriter) const override {
    // Init
    auto pos0 = mlir::ArrayAttr::get(ctxt, rewriter.getI32IntegerAttr(0));
    SmallVector<mlir::Value, 2> rp{
        rewriter.create<mlir::LLVM::ExtractValueOp>(loc, ty, operands[0], pos0),
        rewriter.create<mlir::LLVM::ExtractValueOp>(loc, ty, operands[1], pos0)};
    auto rcp = rewriter.create<mlir::LLVM::FCmpOp>(loc, resTy, rp, cmp->getAttrs());

    // Similarly create comparison for imaginary part in icp
    auto pos1 = mlir::ArrayAttr::get(ctxt, rewriter.getI32IntegerAttr(1));
    // …

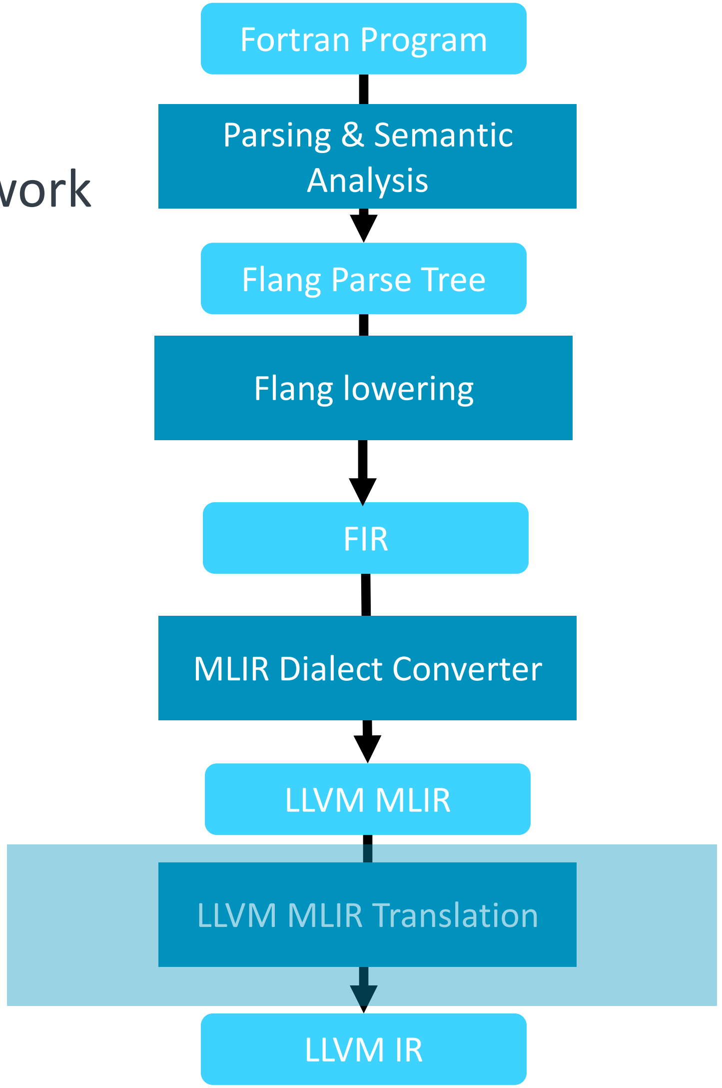
    SmallVector<mlir::Value, 2> cp{rcp, icp};

    switch (cmp.getPredicate()) {
    case mlir::CmpFPredicate::OEQ: // .EQ.
      rewriter.replaceOpWithNewOp<mlir::LLVM::AndOp>(cmp, resTy, cp); break;
    // …
    }
    return success();
  }
};
```

Translation to LLVM IR

Translation to LLVM IR

- Provided by the MLIR framework
  - /bin/mlir-translate --mlir-to-llvmir file.mlir



Join Us

- Flang biweekly calls
  - Mondays and Wednesdays
  - https://github.com/llvm/llvm-project/blob/main/flang/docs/GettingInvolved.md#calls
- Mailing list
  - https://lists.llvm.org/mailman/listinfo/flang-dev
- Slack Channel
  - https://flang-compiler.slack.com/
  - Invite link: https://join.slack.com/t/flang-compiler/shared_invite/zt-2pcn51lh-VrRQL_YUOkxA_1CEfMGQhw
- Status
  - Parsing, Semantic Checks, Runtime : Code is developed in upstream llvm-project/flang
  - Upstreaming of rest of the code is in progress
    - FIR, FIR passes, conversion to LLVM is mostly upstreamed
    - Major remaining portion to upstream is Lowering from parse-tree to FIR.
    - Downstream branch: https://github.com/flang-compiler/f18-llvm-project/tree/fir-dev

Thank You
Danke
Gracias
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה

MLIR Quick Start

MLIR

- Customizable Framework for creating SSA based IRs
- Progressive Lowering
  - Maintain higher level of abstractions
  - Framework to support lowering
- Common Utilities for IR
  - Printer
  - Parser
  - Verifier
- Interfaces for defining transformations so that all dialects benefit
- Location tracking

MLIR: Structure

- IR structure is recursively defined
- MLIR has operations.
  - Even Modules and Functions are operations
- Operations can contain regions
  - Region is a list of basic blocks
- Basic Blocks contain a list of operations

MLIR: Structure

```
%results:2 = "d.operation"(%arg0, %arg1) ({
  // Regions belong to Ops and can have multiple blocks.              Region
  ^block(%argument: !d.type):
                                                                  Block
    %value = "nested.operation"() ({
      // Ops can contain nested regions.                  Region
      "d.op"() : () -> ()
    }) : () -> (!d.other_type)
    "consume.value"(%value) : (!d.other_type) -> ()
  ^other_block:
    "d.terminator"() [^block(%argument : !d.type)] : ()
-> ()
}) : () -> (!d.type, !d.other_type)
```

Source : MLIR tutorial at LLVM Dev Meeting 2020, Mehdi Amini, River Riddle

Dialect

- Dialects are a logical way to organize
  - Operations
  - Types
  - Attributes
  - Analysis, Transformations, Dialect Conversions
  - Custom parser and printer
- Different dialects can co-exist
- Pre-defined dialects
  - Standard
  - SCF (Structured Control Flow/Loop) : Contains Loops and Control Flow
  - Linalg : Linear Algebra
  - Affine : Polyhedral like analysis and transformations
  - LLVM : 1-1 correspondence with LLVM IR
  - OpenMP

Pretty Print

```
func @demo(%lb1 : i32, %ub1 : i32, %step1 : i32,
%data1 : memref<?xi32>) -> () {

  omp.wsloop (%iv) : i32 = (%lb1) to (%ub1) step
(%step1) {

    %1 = test.compute(%iv) : (i32) -> (index)

    memref.store %iv, %data1[%1] : memref<?xi32>

    omp.yield

  }

  return

}
```

Generic Syntax

```
"module"() ( {
  "func"() ( {
  ^bb0(%arg0: i32, %arg1: i32, %arg2: i32, %arg3:
memref<?xi32>):  // no predecessors

    "omp.wsloop"(%arg0, %arg1, %arg2) ( {

    ^bb0(%arg5: i32):  // no predecessors

      %1 = "test.compute"(%arg5) : (i32) -> index

      "memref.store"(%arg5, %arg3, %1) : (i32,
memref<?xi32>, index) -> ()

      "omp.yield"() : () -> ()

    }) {operand_segment_sizes = dense<[1, 1, 1, 0, 0,
0, 0, 0, 0]> : vector<9xi32>} : (i32, i32, i32) -> ()

    "std.return"() : () -> ()

  }) {sym_name = "demo", type = (i32, i32, i32,
memref<?xi32>, memref<?xi32>) -> ()} : () -> ()
}) : () -> ()
```

Operation

- Name/Opcode
- Input, Output Operands
- Attributes : Constant values
- Can contain regions with terminator instruction
  - Useful for modelling OpenMP, loop etc
- Operations defined using Operation Definition Specification (ODS)

ODS

Linalg Op Base

```
// Base class for Linalg dialect ops that do not correspond to library calls.
class Linalg_Op<string mnemonic, list<OpTrait> traits = []> : Op<Linalg_Dialect, mnemonic, traits> {
let printer = [{ return ::print(p, *this); }];
  let verifier = [{ return ::verify(*this); }];
  let parser = [{ return ::parse$cppClass(parser, result); }];
}
```

Linalg Range Op

```
def Linalg_RangeOp :
    Linalg_Op<"range", [NoSideEffect]>,
    Arguments<(ins Index:$min, Index:$max, Index:$step)>,
    Results<(outs Range)> {
  let summary = "Create a `range` type value, used to create `view`s";
  let description = [{
    Example:
    ```mlir
    %3 = linalg.range %0:%1:%2 : !linalg.range
    ````
let verifier = ?;
  let assemblyFormat = "$min `:` $max `:` $step attr-dict `:` type(results)";
}
```

ODS: Generated class/functions

- Linalg Range Op

```
class RangeOp : public ::mlir::Op<RangeOp, OpTrait::ZeroRegion, OpTrait::OneResult, OpTrait::ZeroSuccessor,
OpTrait::NOperands<3>::Impl, ::mlir::MemoryEffectOpInterface::Trait> {
public:
  using Op::Op;
  using Adaptor = RangeOpAdaptor;
  static ::llvm::StringRef getOperationName();
. . . . .
  ::mlir::Value min();
  ::mlir::Value max();
  ::mlir::Value step();
. . . . .
  static void build(OpBuilder &builder, OperationState &result, Value min, Value max, Value step);
. . . . .
  ::mlir::LogicalResult verify();
  static ::mlir::ParseResult parse(::mlir::OpAsmParser &parser, ::mlir::OperationState &result);
  void print(OpAsmPrinter &p);
};
```

MLIR vs LLVM IR: Similarities

- SSA based
- Typed
- Round-trippable textual form
- Syntactically similar

MLIR vs LLVM IR: Differences

MLIR

- Extendable via dialects
- Blocks have arguments
- Operations
- Operations can have regions
- Some dialects have higher level constructs like the loop operation
- High quality source locations as part of every operation

LLVM IR

- Fixed IR
- Phi values use to merge control flow
- Instructions
- No region support. Outlining needed
- No loops. Loop information is generally recreated in LLVM
- Source location is an after-thought via debug

FIR: Do Loop Conversion (FIR -> LLVM MLIR)

```
func @_QPsumn(%arg0: !fir.ref<i32>) -> i32 {
    // init and allocate ((%2 is 1 and %4 is N)
    %5 = fir.do_loop %arg1 = %2 to %4 step %c1 ->
index {
    %8 = fir.convert %arg1 : (index) -> i32
    fir.store %8 to %0 : !fir.ref<i32>
    %9 = fir.load %1 : !fir.ref<i32>
    %10 = fir.load %0 : !fir.ref<i32>
    %11 = addi %9, %10 : i32
    fir.store %11 to %1 : !fir.ref<i32>
    %12 = addi %arg1, %c1 : index
    fir.result %12 : index
  }
  // convert and store %5 into var i
  // load and return value of var res
}
```

```
llvm.func @_QPsumn(%arg0: !llvm.ptr<i32>) -> i32
  // init and allocate (%2 is 1 and %8 is N)
  llvm.br ^bb1(%2, %8 : i64, i64)
^bb1(%11: i64, %12: i64):  // 2 preds: ^bb0, ^bb2
  %13 = llvm.icmp "sgt" %12, %1 : i64
  llvm.cond_br %13, ^bb2, ^bb3
^bb2:  // pred: ^bb1
  %14 = llvm.trunc %11 : i64 to i32
  llvm.store %14, %4 : !llvm.ptr<i32>
  %15 = llvm.load %6 : !llvm.ptr<i32>
  %16 = llvm.load %4 : !llvm.ptr<i32>
  %17 = llvm.add %15, %16  : i32
  llvm.store %17, %6 : !llvm.ptr<i32>
  %18 = llvm.add %11, %2  : i64
  %19 = llvm.sub %12, %2  : i64
  llvm.br ^bb1(%18, %19 : i64, i64)
^bb3:  // pred: ^bb1
  // convert and store %11 into var I
  // load and return value of var res
}
```

FIR: Do Loop Description

```
def fir_DoLoopOp : region_Op<"do_loop", [DeclareOpInterfaceMethods<LoopLikeOpInterface>]> {
  let summary = "generalized loop operation";
  let description = [{
    Generalized high-level looping construct. This operation is similar to MLIR’s `scf.for`.
  }];
  let arguments = (ins
    Index:$lowerBound,
    Index:$upperBound,
    Index:$step,
    Variadic<AnyType>:$initArgs,
    OptionalAttr<UnitAttr>:$unordered,
    OptionalAttr<UnitAttr>:$finalValue
  );
  let results = (outs Variadic<AnyType>:$results);
  let regions = (region SizedRegion<1>:$region);
  let skipDefaultBuilders = 1;
  …
}
```