Roll your own compiler

Easy IR generation

Kai Nacke

3 February 2019

LLVM dev room @ FOSDEM‘19

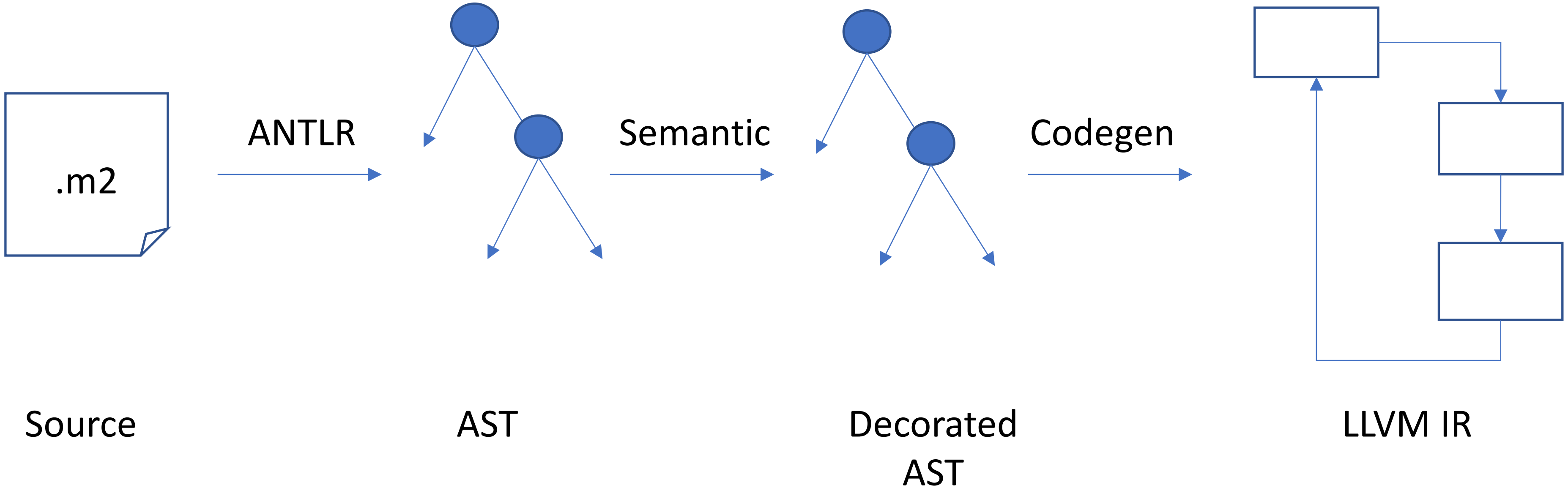
What are the obstacles of IR generation?

Modula-2

- First implementation in 1979 for the PDP-11
- Complete language
  - Carefully designed syntax
  - Module concept
  - Low-level facilities and procedure types
- Large code base available
  - The Lilith operating system
  - The GMD compiler toolbox (“cocktail”)
- Later standardized as ISO 10514

```
(* Taken from PIM4, page 25. *)
MODULE gcdlcm;
FROM InOut IMPORT ReadInt, WriteLn,
                  WriteString, WriteInt;
VAR x, y, u, v: INTEGER;
BEGIN
  WriteString("x = "); ReadInt(x); WriteLn;
  WriteString("y = "); ReadInt(y);
  u := x; v := y;
  WHILE x # y DO
    IF x > y THEN
      x := x - y; u := u + v
    ELSE
      y := y - x; v := v + u
    END
  END;
  WriteInt(x, 6); WriteInt((u+v) DIV 2, 6); WriteLn
END gcdlcm.
```

m2lang – The LLVM-based Modula-2 compiler



- Modula-2 grammar provided by ANTLR
- Semantic phase and IR generation hand-coded
- Semantic phase uses hand-coded AST
- Goal: replace ANTLR with RD-parser

Source will be published here: https://github.com/redstar/m2lang

Basic blocks

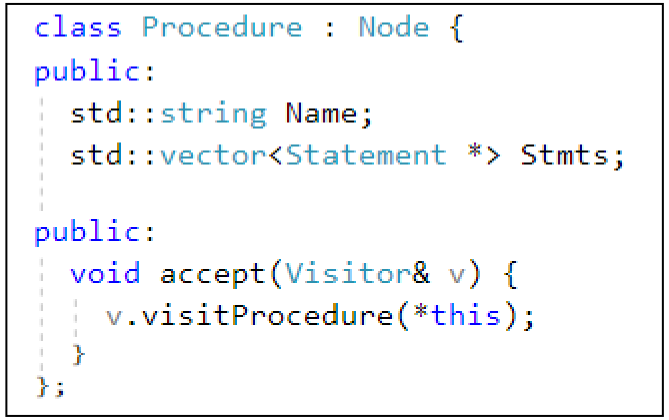
- IR instructions go into a basic block
- A basic block is a single entry single exit section of code
  - Entry is with first instruction, usually marked with a label
  - Ends with a terminating instruction, e.g. conditionally/unconditionally branch, return

```
if:
  %6 = load i32, i32* %3, align 4
  %7 = load i32, i32* %4, align 4
  %8 = icmp ne i32 %6, %7
  br i1 %8, label %then, label %else
```

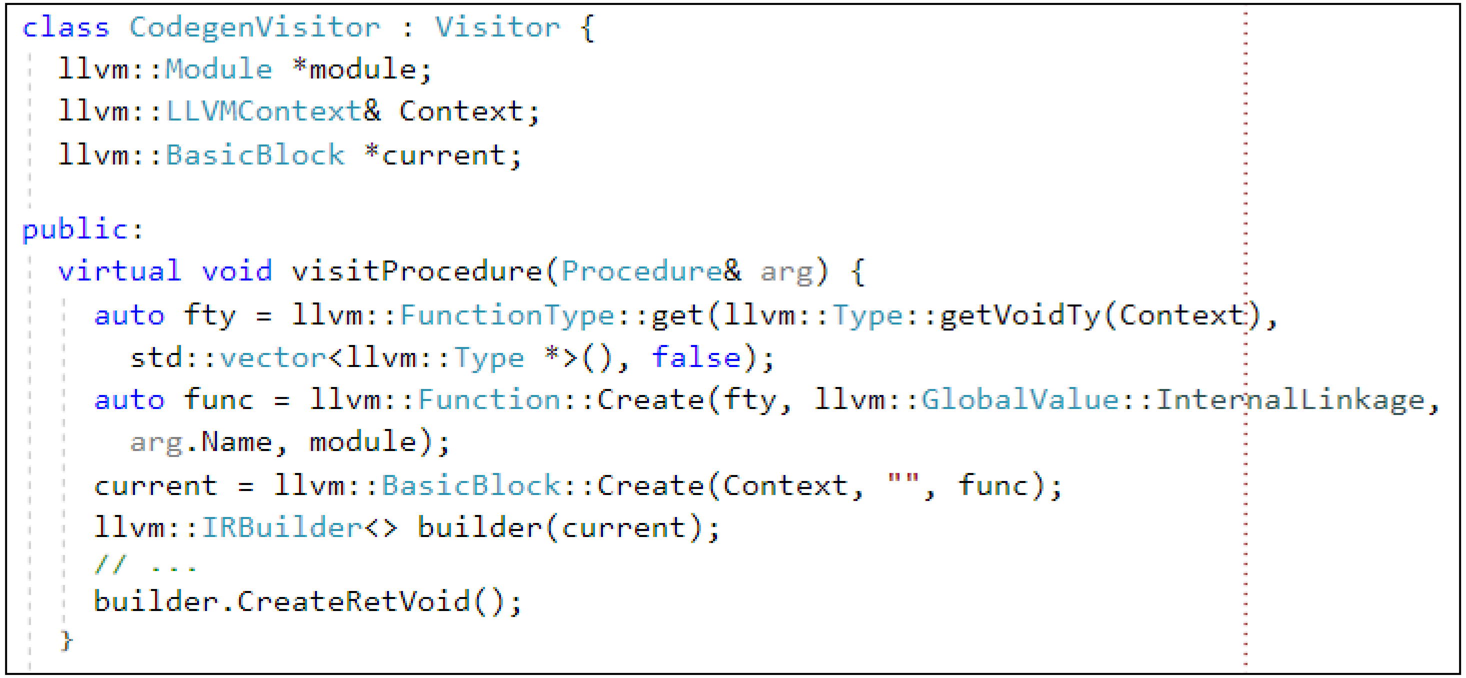
- Optimization is usually applied to basic blocks
- All basic blocks of a function form a control flow graph (CFG)

The naive approach to IR generation

- Define a visitor holding pointer to current basic block
- Traverse the AST and generate IR
  - Create a new basic block if needed



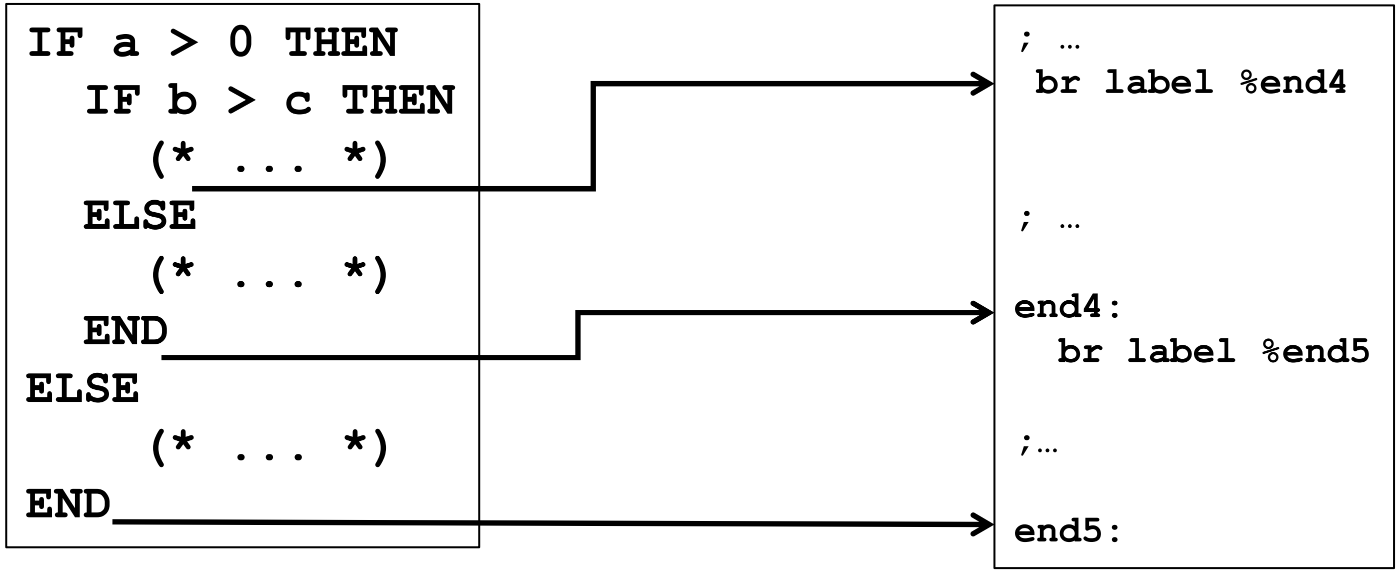
```
class Procedure : Node {
public:
  std::string Name;
  std::vector<Statement *> Stmts;

public:
  void accept(Visitor& v) {
    v.visitProcedure(*this);
  }
};
```



```
class CodegenVisitor : Visitor {
  llvm::Module *module;
  llvm::LLVMContext& Context;
  llvm::BasicBlock *current;

public:
  virtual void visitProcedure(Procedure& arg) {
    auto fty = llvm::FunctionType::get(llvm::Type::getVoidTy(Context),
      std::vector<llvm::Type *>(), false);
    auto func = llvm::Function::Create(fty, llvm::GlobalValue::InternalLinkage,
      arg.Name, module);
    current = llvm::BasicBlock::Create(Context, "", func);
    llvm::IRBuilder<> builder(current);
    // ...
    builder.CreateRetVoid();
  }
```

The trouble with naive approach

- Naive approach works well with simple arithmetic
- Now consider nested IF-THEN-ELSE-END structures



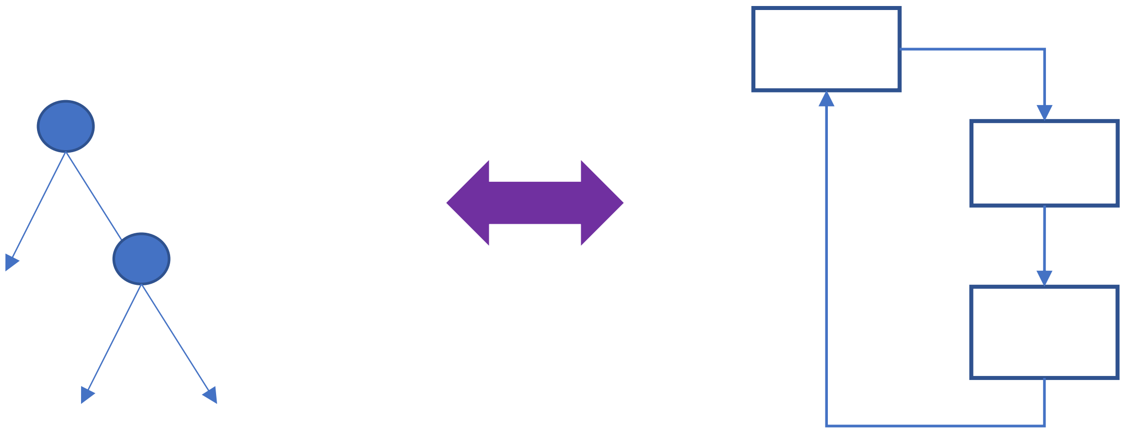
```
IF a > 0 THEN
  IF b > c THEN
    (* ... *)
  ELSE
    (* ... *)
  END
ELSE
    (* ... *)
END
```

```
; …
 br label %end4


; …

end4:
  br label %end5

;…

end5:
```

- Current block can be empty (“**END**”)
- Generates blocks with branch instruction only

The gap between AST and LLVM IR

- The AST is more closely to the textual representation
- The basic blocks form a control flow graph
- Generation of „branch only“ basic blocks is result of this mismatch

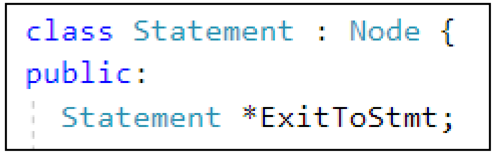


AST

CFG of basic blocks

How to resolve

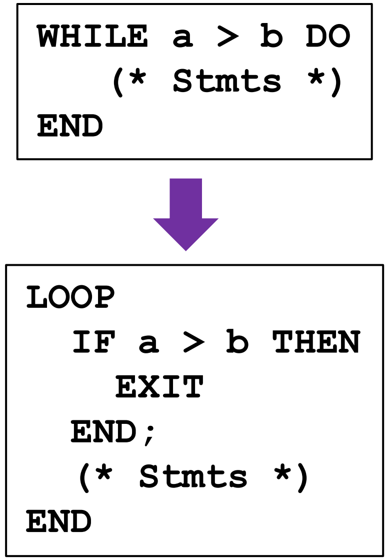
- Do not care – let LLVM optimize it away
  - Simple
- Induce the CFG on the AST
  - Just adds a pointer to the AST („next basic block“)
  - Can be constructed very fast with recursive visitor
- Explicitly construct the CFG
  - Costly if only done for construction of IR



```
class Statement : Node {
public:
  Statement *ExitToStmt;
```

Transform AST into high-level CFG

- Goal is to transform the AST into a representation closer to a CFG
- Lower high-level constructs in low-level constructs
  - Replace FOR, WHILE, REPEAT with LOOP/EXIT
  - Replace AND/OR with nested IF
- Introduce GOTO
  - Lowering every implicit jump into a GOTO creates a CFG
- Think how to preserve debug metadata!
- You now have created your own IR!



```
WHILE a > b DO
    (* Stmts *)
END
```

```
LOOP
  IF a > b THEN
    EXIT
  END;
  (* Stmts *)
END
```

When is another IR needed?

- Creating another IR can be helpful
  - Elaborate type checking
  - Scope checking
  - Generating synthetic code (e.g. cleanup handlers)

- Do only when needed
  - Modula-2 seems to be simple enough to go without new IR

- Be careful
  - Do not replicate LLVM functionality at a higher level
  - Consider adding a new LLVM pass instead

Thank you!