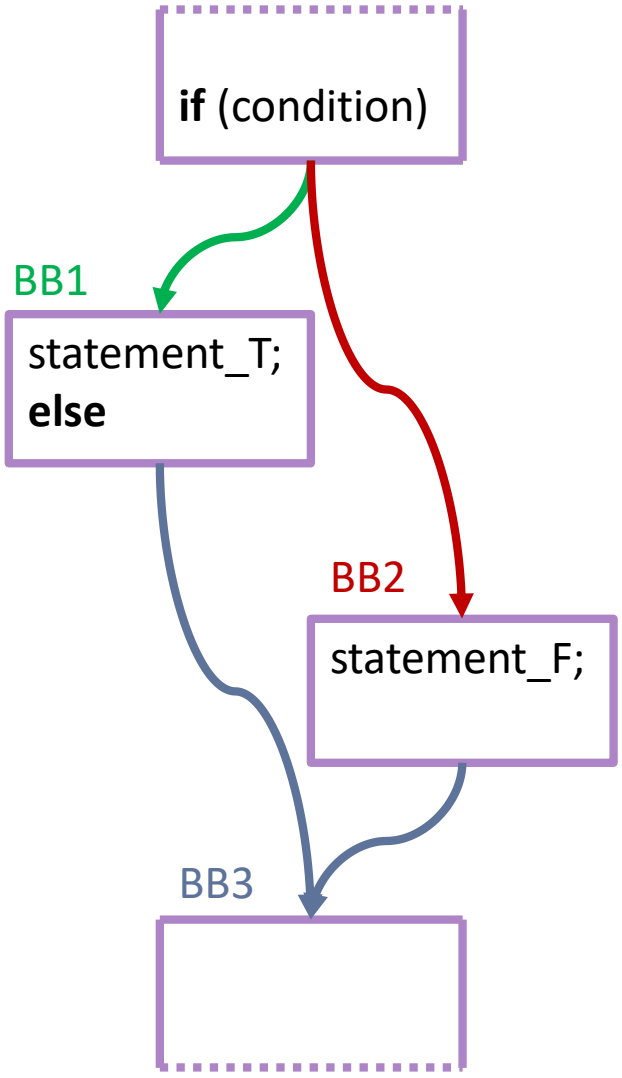
Cecko

Assignment 5 – Generating code for statements

Control flow in LLVM IR

- Instructions are created using llvm::IRBuilder
  - accessible as ctx->builder()->CreateXYZ(...)
- the builder appends instructions at an **insertion point** into a **basic block**
  - it may also be in unbound state, unable to emit instructions
  - preset by the framework upon entering a function body (ctx->enter_function())
- any control-flow statement requires creation of new basic blocks
  - basic blocks are created by calling ctx->create_basic_block(name)
  - a basic block is identified by CKIRBasicBlockObs
- a basic block is needed to
  - create a branch instruction pointing to that BB
    - a BB may be a target of more than one branch instruction
  - set builder insertion point before generating instructions to that BB
    - this can be done repeatedly, producing a BB from more than one batch of instructions
    - the order of instructions in a BB is determined by the order of Create... calls
    - this order usually corresponds to the order statements/expressions in the source code
    - if a different order is required, more than one BB must be used
- if a branch or return instruction is generated into a basic block
  - adding further instructions to the same BB will produce invalid code
  - it is a good idea to signalize it by calling ctx->builder()->ClearInsertionPoint()
    - check the insertion point (GetInsertBlock()!=nullptr) before generating implicit returns or unconditional branch instructions

IF-ELSE statement



- IF-ELSE statement
  - the condition is generated into the previous BB
  - the Boolean result is held in LLVM Value of type **i1**
  - at the end of the condition
    - create two new BBs
    - append a conditional branch

```
ctx->builder()->CreateCondBr(cond,BB1,BB2);
```

    - switch the builder to BB1

```
ctx->builder()->SetInsertPoint(BB1);
```

    - the then-statement is generated into BB1
  - at the **else** keyword
    - create a third BB
    - append an unconditional branch
      - only if the builder still has an insertion point

```
ctx->builder()->CreateBr(cond,BB3);
```

    - switch the builder to BB2

```
ctx->builder()->SetInsertPoint(BB2);
```

    - the else-statement is generated into BB2
  - at the end of the else-statement
    - append an unconditional branch
      - only if the builder still has an insertion point

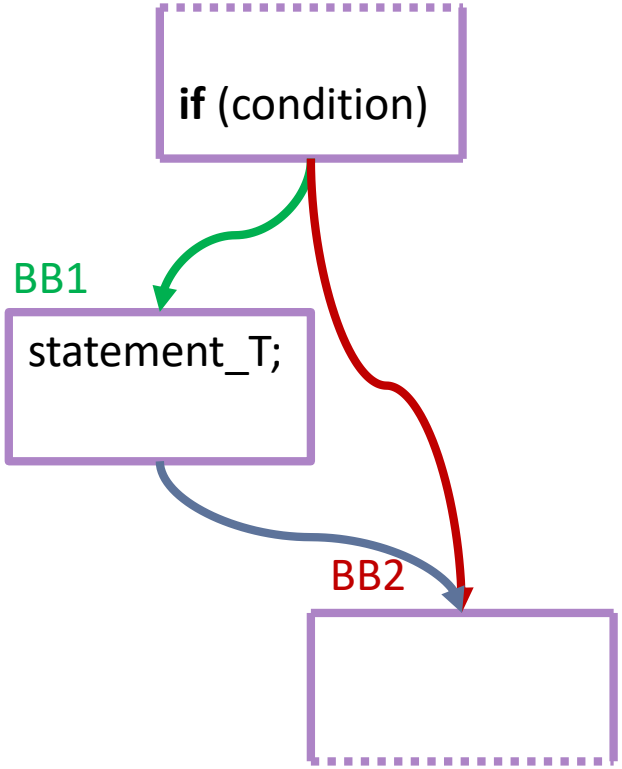
```
ctx->builder()->CreateBr(cond,BB3);
```

    - switch the builder to BB3

```
ctx->builder()->SetInsertPoint(BB3);
```

    - the following statements will be generated into BB3
      - BB3 not needed if no branch to BB3 was generated

Incomplete IF statement



- Incomplete IF statement
  - the condition is generated into the previous BB
  - the Boolean result is held in LLVM Value of type i1
  - at the end of the condition
    - create two new BBs
    - append a conditional branch

```
ctx->builder()->CreateCondBr(cond,BB1,BB2);
```

    - switch the builder to BB1

```
ctx->builder()->SetInsertPoint(BB1);
```

    - the then-statement is generated into BB1
  - when there is no **else** keyword
    - append an unconditional branch
      - only if the builder still has an insertion point

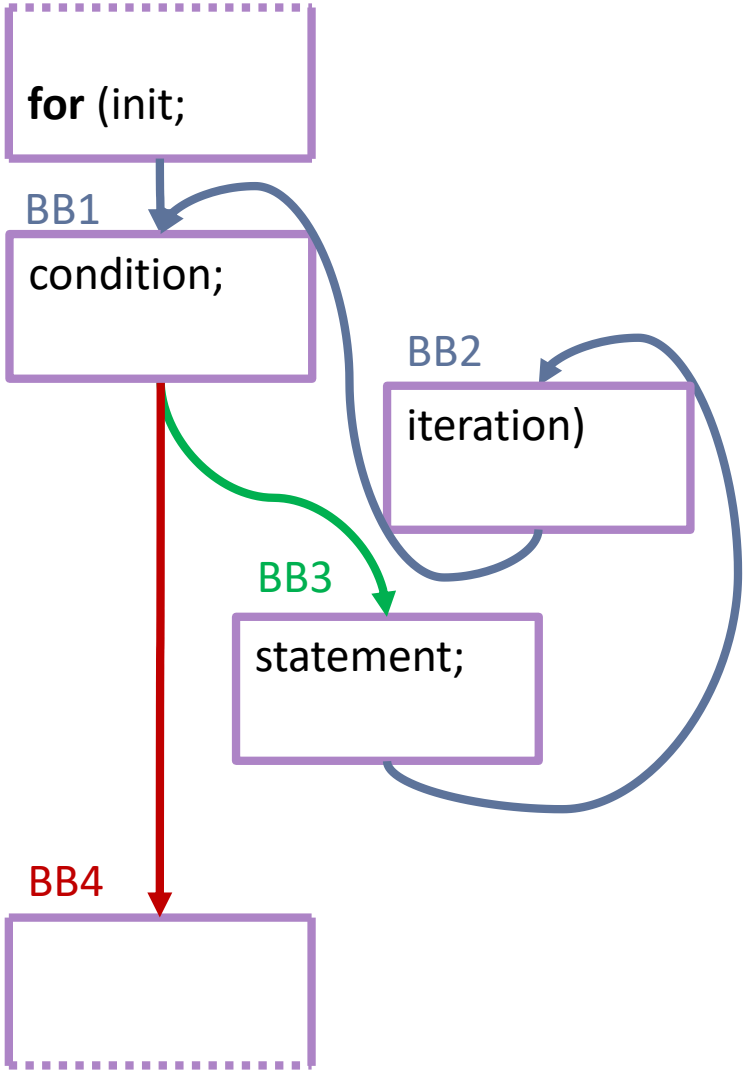
```
ctx->builder()->CreateBr(cond,BB2);
```

    - switch the builder to BB2

```
ctx->builder()->SetInsertPoint(BB2);
```

    - the following statements will be generated into BB2

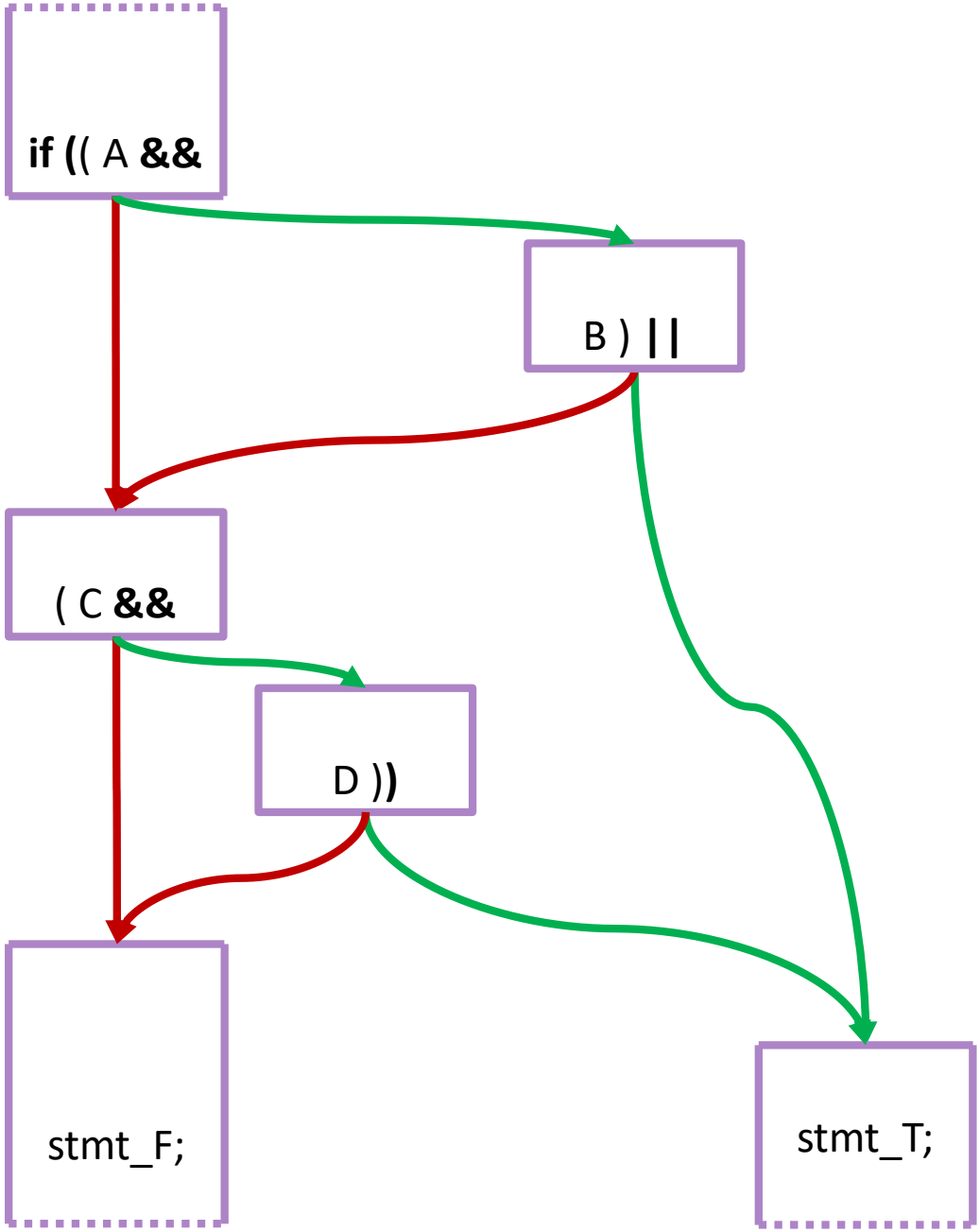
FOR statement



- FOR statement
  - The *condition* expression requires a new BB because we need to jump there after each iteration
  - The *iteration* expression and the body *statement* must be executed in different order than they are parsed – two BBs required
  - The grammar may need rewriting into something like this:

```
for1: FOR LPAR expr SEMIC
  { $$.bb1=...; }
;
for2: for1 expr SEMIC
  { $$.bb1=$1.bb1;
    $$.bb3=...;
    $$.bb4=...;
    $$.bb2=...; }
;
for3: for2 expr RPAR
  { $$.bb2=$1.bb2; $$.bb4=$1.bb4;
    ... $1.bb1 ... $1.bb3 ... }
;
for_statement: for3 statement
  { ... $1.bb2 ... $1.bb4 ... }
;
```

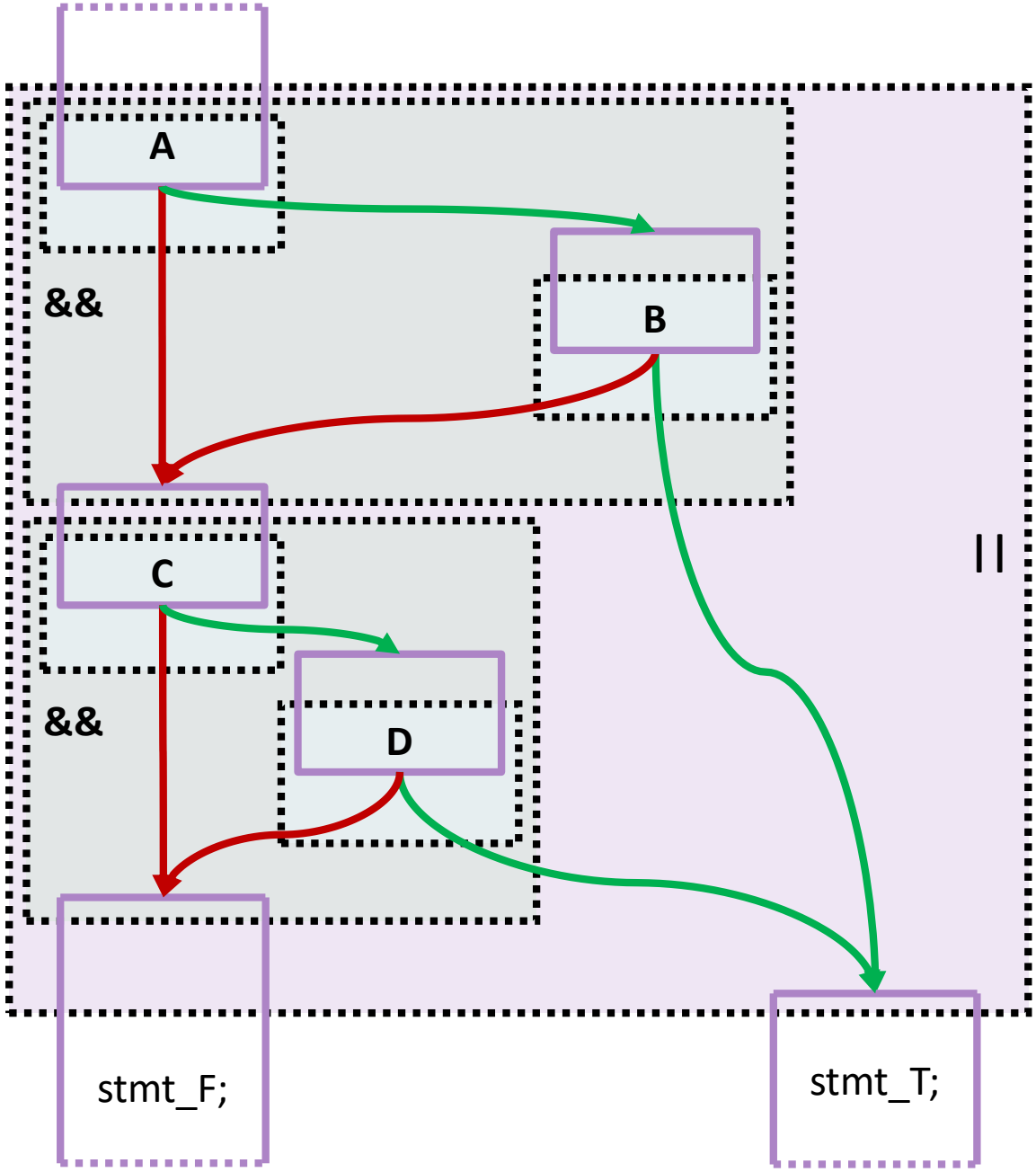
Shortcut evaluation of &&, ||



- This is the ideal implementation of

```
if ((A && B) || (C && D))
  stmt_T;
else
  stmt_F;
```

Shortcut evaluation of &&, ||

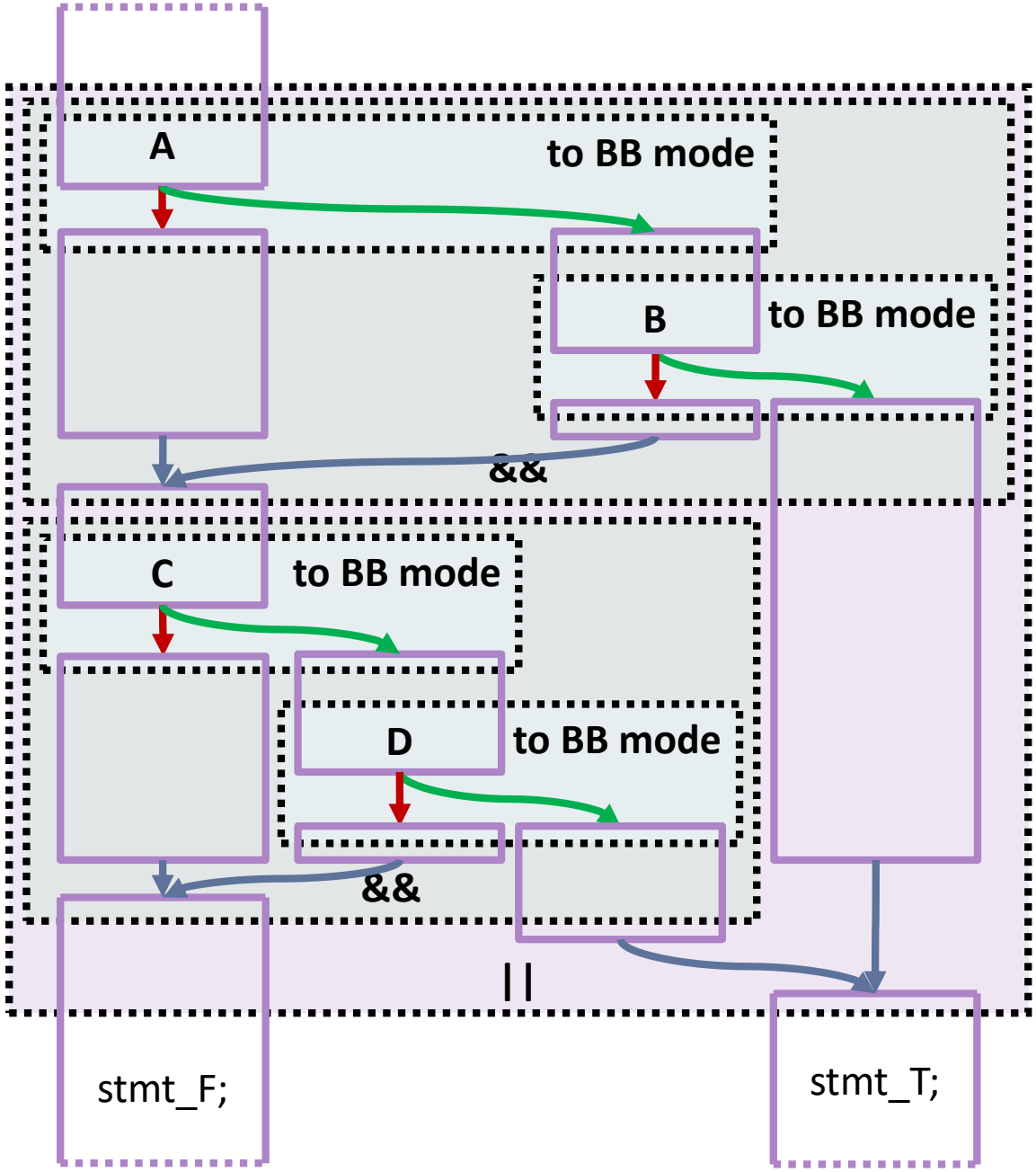


- This is the ideal implementation of

```
if ((A && B) || (C && D))
  stmt_T;
else
  stmt_F;
```

- It is impossible to generate this code in our environment:
  - The true branches of the B and D conditions must point to the same BB
  - The bottom-up parser does not allow coordination between the two subexpressions

Shortcut evaluation of &&, ||



- This is a possible implementation of

```
if ((A && B) || (C && D))
  stmt_T;
else
  stmt_F;
```

  - New **BB mode** of subexpression:
    - Represented by the pair [trueBB,falseBB]
    - Executing trueBB signalizes value 1
    - Executing falseBB signalizes value 0
  - Conversion to BB mode
    - Create two BBs
    - Generate CondBr
  - && operator
    - Enforce BB mode on the left operand
    - Set insert point to left.trueBB before the right operand
    - Create a new falseBB to merge left.falseBB with right.falseBB
  - || operator
    - Same as && but swap trueBB with falseBB
  - Produces empty BBs with unconditional branches

Instructions needed for Cecko

Instruction	Asgn 4	Asgn 5	Note
GlobalString	STRLIT		produces llvm::Constant*
ConstInBoundsGEP2_32	Array to ptr		use two 0 indexes
ICmpNE	char/int to _Bool	!=	in _Bool= and conditions
IsNotNull	ptr to _Bool		in _Bool= and conditions
ZExt	_Bool/char to char/int		in most operators
Trunc	int to char		in char=
Add,Sub,Mul,SDiv,SRem	int+int,-,*,/,%		
GEP	ptr+int,ptr-int,ptr[int]		
Neg	-int,ptr-int		
PtrDiff	ptr-ptr		
StructGEP	str.name,ptr->name		use get_idx()
ExtractValue	f().name		non-L-value before .name
Load	L-value to R-value		
Store	=		
Ret,RetVoid	return		incl. implicit
Call	Function call		incl. void

Instructions needed for Cecko

Instruction	Asgn 4	Asgn 5	Note
ICmpEQ		==	
ICmpSLT,SLE,SGT,SGE		int<int,<=,>,>=	
ICmpULT,ULE,UGT,UGE		ptr<ptr,<=,>,>=	
Not		!	or swap BBs
CondBr		if,while,for,&&,||	
Br		else,do,while,for,&&,||	