andreroesti@anroesti-macbook: ~/UCI/2021-Winter/CS241/smpl-compiler ⌥⌘1

```
→  smpl-compiler git:(master) ✗ ./main.py --help
usage: main.py [-h] [-o OUTPUT] [--ast] [--interpret] [--ir] [--no-ce]
               [--no-dce] [--no-cse] [--dlx] [--asm] [--run]
               infile

Compile a Smpl program to abstract syntax tree, SSA intermediate
representation graph, DLX assembly or DLX machine code.

positional arguments:
  infile

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
                        Output file (stdout by default).
  --ast                 Produce only AST graph.
  --interpret           Run interpreter on the input program instead of
                        compiling.
  --ir                  Produce only IR graph.
  --no-ce               Disable constant elimination.
  --no-dce              Disable dead code elimination.
  --no-cse              Disable common subexpression elimination.
  --dlx                 Produce DLX machine code.
  --asm                 Output assembly instead of machine code.
  --run                 Run byte code in DLX emulator.
```

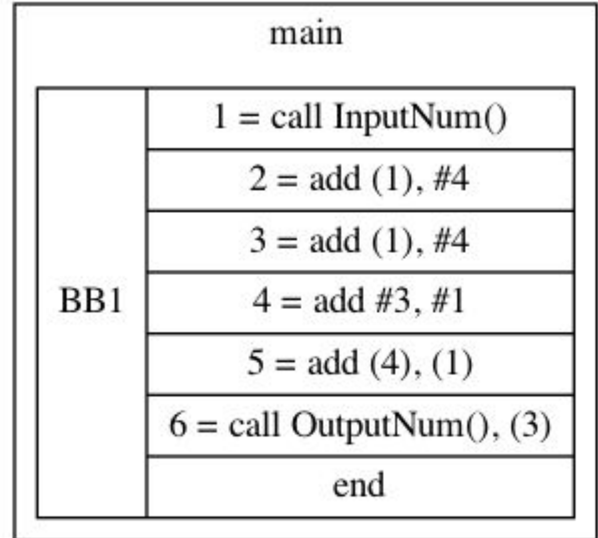
CS244 Advanced Compiler Design

Final Presentation

André Rösti

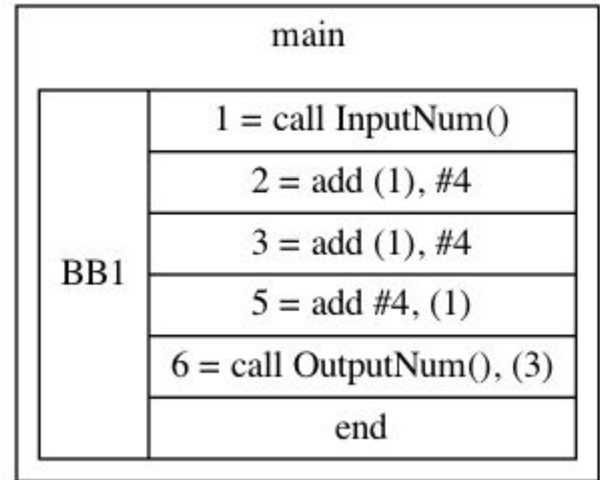
Unoptimized

```
main
var input, constant, redundant,
output, unused;
{
   let input <- call InputNum();
   let constant <- 4;
   let redundant <- input + constant;
   let output <- input + constant;
   let unused <- 3 + 1 + input;
   call OutputNum(output);
}.
```



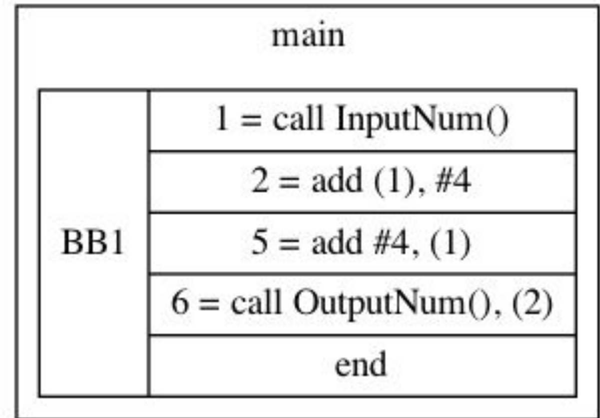
Constant Elimination

```
main
var input, constant, redundant,
output, unused;
{
   let input <- call InputNum();
   let constant <- 4;
   let redundant <- input + constant;
   let output <- input + constant;
   let unused <- 3 + 1 + input;
   call OutputNum(output);
}.
```



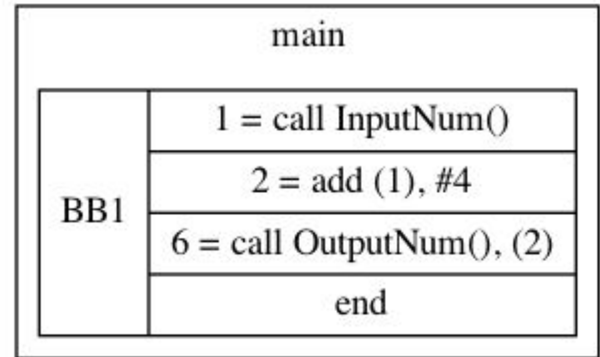
Common Subexpression Elimination

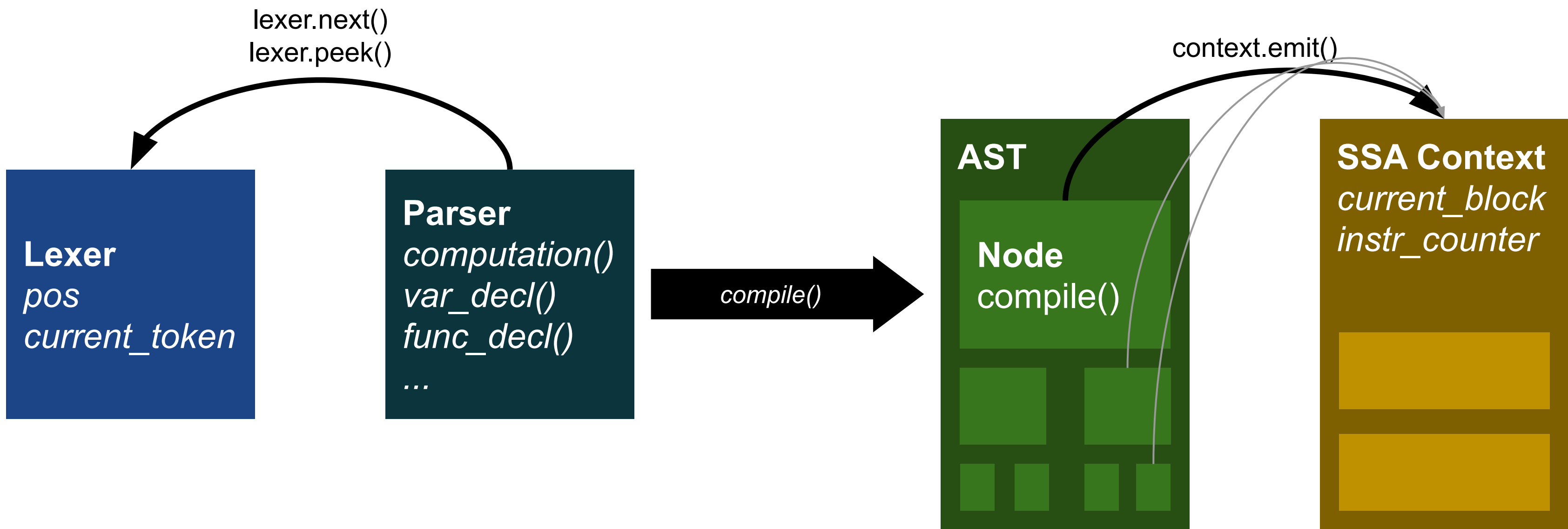
```
main
var input, constant, redundant,
output, unused;
{
   let input <- call InputNum();
   let constant <- 4;
   let redundant <- input + constant;
   let output <- input + constant;
   let unused <- 3 + 1 + input;
   call OutputNum(output);
}.
```

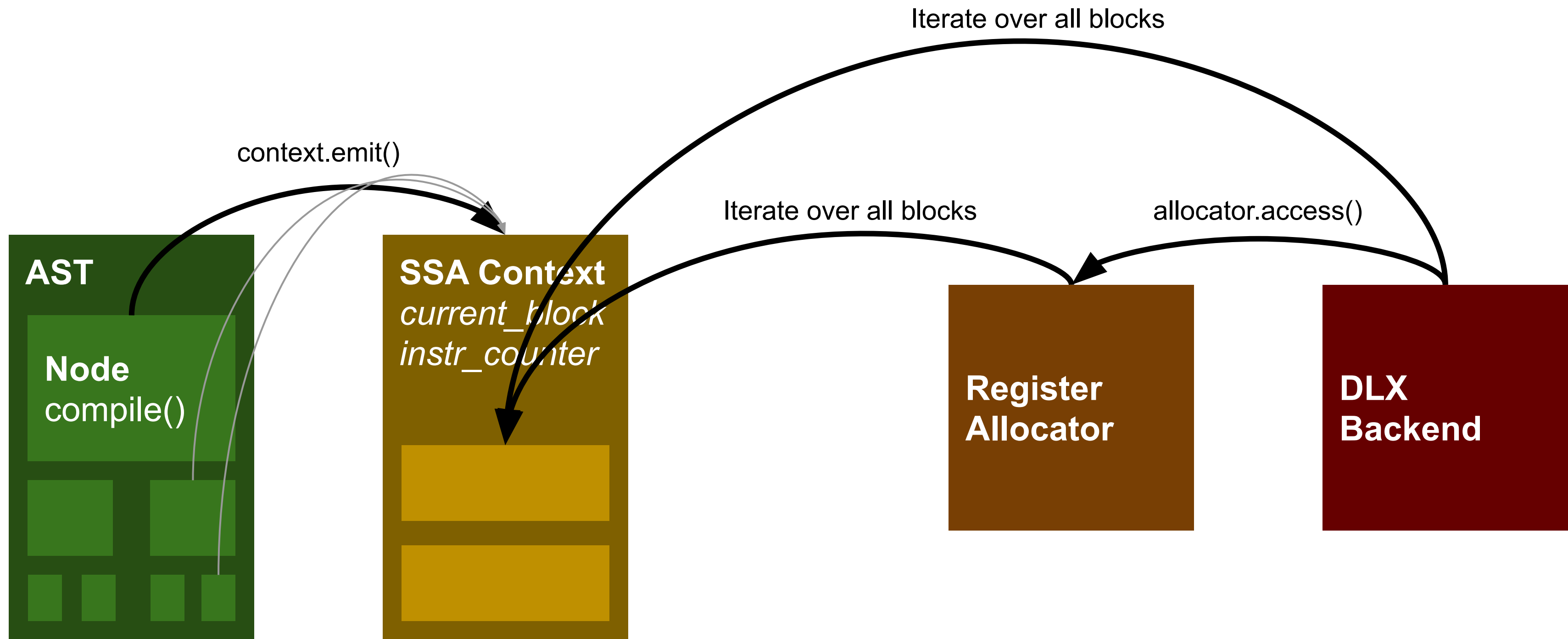


Dead Code Elimination

```
main
var input, constant, redundant,
output, unused;
{
   let input <- call InputNum();
   let constant <- 4;
   let redundant <- input + constant;
   let output <- input + constant;
   let unused <- 3 + 1 + input;
   call OutputNum(output);
}.
```




lexer.next()
lexer.peek()
Lexer
pos
current_token
Parser
computation()
var_decl()
func_decl()
...
compile()
context.emit()
AST
Node
compile()
SSA Context
current_block
instr_counter


Iterate over all blocks
context.emit()
Iterate over all blocks
allocator.access()
AST
Node
compile()
SSA Context
current_block
instr_counter
Register
Allocator
DLX
Backend

Compiling AST to SSA: example

```python
def compile(self, context):

    # Recursively compile instructions for RHS value of assignment
    val_op = self.rhs.compile(context)
    if isinstance(self.lhs, Identifier):
        # Update local context s.t. identifier refers to new value
        context.current_block.set_local_op(self.lhs.name, val_op)

    elif isinstance(self.lhs, ArrayAccess):
        # Emit array store to memory
        name = self.lhs.identifier.name
        if name not in context.current_block.locals_op:
            raise Exception(f"Assignment to undeclared array '{name}'")

        addr_op = self.lhs.compile_addr(context)
        context.emit("store", val_op, addr_op, produces_output=False)

    return None
```

One AST class for every syntactic element

Context keeps track of emitted instructions, blocks and variable assignments

Value-producing syntactic elements may return a SSA operand upon compilation

If-Then-Else Control Flow / Phi Nodes

1. ***context.get_new_block_with_same_context()***
   produces a child blocks
2. Compile condition *(parent block)*
3. Emit branch/jump *(parent block)*
4. Recursive compile bodies *(child blocks)*
5. **Scan child contexts**
   a. If child variable value ≠ parent variable value, emit **phi node** for that variable *(join block)*

While Control Flow / Phi Nodes

1. Produce body block
2. **Wrap variables in body block**
    a. Every operand is wrapped into a **PossiblyPhiOp**
3. Compile body
4. **Scan body context** and **update loop head context variables**
    a. If value changed (i.e. not wrapped)
        i. emit phi in head block,
        ii. rename operand to new phi operand *(recursively)* in **both** head and body block(s)
    b. else, simply unwrap *(recursively)*
5. Compile branch/jump in head block

```
andreroesti@anroesti-macbook: ~/UCI/2021-Winter/CS241/smpl-compiler
→ smpl-compiler git:(master) ✗ ./main.py --help
usage: main.py [-h] [-o OUTPUT] [--ast] [--interpret] [--ir] [--no-ce]
               [--no-dce] [--no-cse] [--dlx] [--asm] [--run]
               infile

Compile a Smpl program to abstract syntax tree, SSA intermediate
representation graph, DLX assembly or DLX machine code.

positional arguments:
  infile

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
                        Output file (stdout by default).
  --ast                 Produce only AST graph.
  --interpret           Run interpreter on the input program instead of
                        compiling.
  --ir                  Produce only IR graph.
  --no-ce               Disable constant elimination.
  --no-dce              Disable dead code elimination.
  --no-cse              Disable common subexpression elimination.
  --dlx                 Produce DLX machine code.
  --asm                 Output assembly instead of machine code.
  --run                 Run byte code in DLX emulator.
```

CS244 Advanced Compiler Design

Final Presentation

André Rösti

Unoptimized

```
main
var input, constant, redundant,
output, unused;
{
   let input <- call InputNum();
   let constant <- 4;
   let redundant <- input + constant;
   let output <- input + constant;
   let unused <- 3 + 1 + input;
   call OutputNum(output);
}.
```