Review Lecture

Introduction

Lexical analysis

Parsing

Semantic analysis

(Intermediate) Code generation

Intermediate Representations

Optimization

Instruction selection

Register Allocation

Introduction

Compilers translate code from one langauge (generally a high-level one) into another (generally a low-level or machine language).

Compilers are organized into **phases**. The phases of a typical compiler are:

1. Lexical Analysis (scanning, tokenizing)
2. Syntax Analysis (parsing)
3. Semantic Analysis (typechecking+)
4. Intermediate Code Generation
5. Optimization
6. Target Code Generation

Compilers live in an environment with other related programs, such as:

- linkers
- loaders
- assemblers
- debuggers
- editors
- Integrated Development Environments

Compilers are required to make code that is above all correct, but also efficient, compact, and light on power usage. Sometimes these cannot all be achieved together.

Lexical Analysis

Lexical analysis is the dividing up of the input character stream into **tokens**. This is often done by characterizing the tokens as **regular expressions**. Regular expressions can be found in the input stream by **DFAs** or **NFAs**.

A **deterministic finite automaton** (DFA) is a mathematical machine $(Q, \Sigma, \delta, q_0, F)$

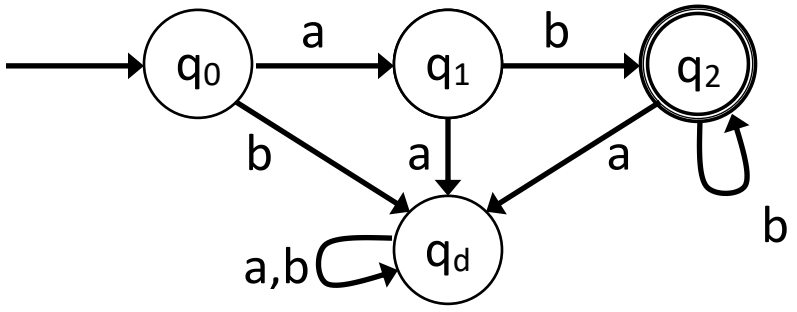
Q: Finite set of states

Σ: Finite alphabet

δ: Transition function $Q \times \Sigma \rightarrow Q$

q_0: Start state $q_0 \in Q$

F: Final states $F \subseteq Q$



Nondeterministic Finite Automaton (NFA)

$(Q, \Sigma, \delta, q_0, F)$

Q: Finite set of states (text uses S)

Σ: Finite alphabet

δ: Transition function $Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$

q_0: Start state $q_0 \in Q$

F: Final states $F \subseteq Q$

NFAs are like DFAs, but…

- You can have more than one outgoing arrow from a state that is labelled with the same character.
- There are transitions on (arrows with label) ε, which can be taken at any time during the operation of the machine.
- The machine **accepts** if **there exists** a path labelled with the input (and epsilons) from the start state to a final state.

We can convert an NFA into a DFA by the **subset construction.** We can convert a regular expression into an NFA by **Thompson's construction**. We can convert a DFA into a regular expression by **Kleene's construction.**

Thus, DFAs, NFAs, and regular expressions all characterize the same set of languages. They are equivalent in power.

Parsing

Program syntax is most often described using a **context-free grammar (CFG).**

A CFG is

$G = (N, T, S, P)$

N: Set of nonterminals (often capital letters)

T: Set of terminals (often lower-case letters and symbols)

S: Start symbol (element of N)

P: Productions of the form

$A \rightarrow \alpha$

where α is a string on $N \cup T \cup \{\varepsilon\}$.

This is the BNF form of productions (Backus-Naur Form). The checkpoints use the EBNF (Extended BNF) form, where α can be a regular expression on $N \cup T \cup \{\varepsilon\}$. This is just a shorthand and does not change which languages have a grammar.

There are standard conventions for giving a grammar. We only list the productions. N is taken to be the set of all symbols on the left-hand side of any production. T is taken to be the set of all other symbols appearing in productions. S is taken to be the left-hand symbol of the first production.

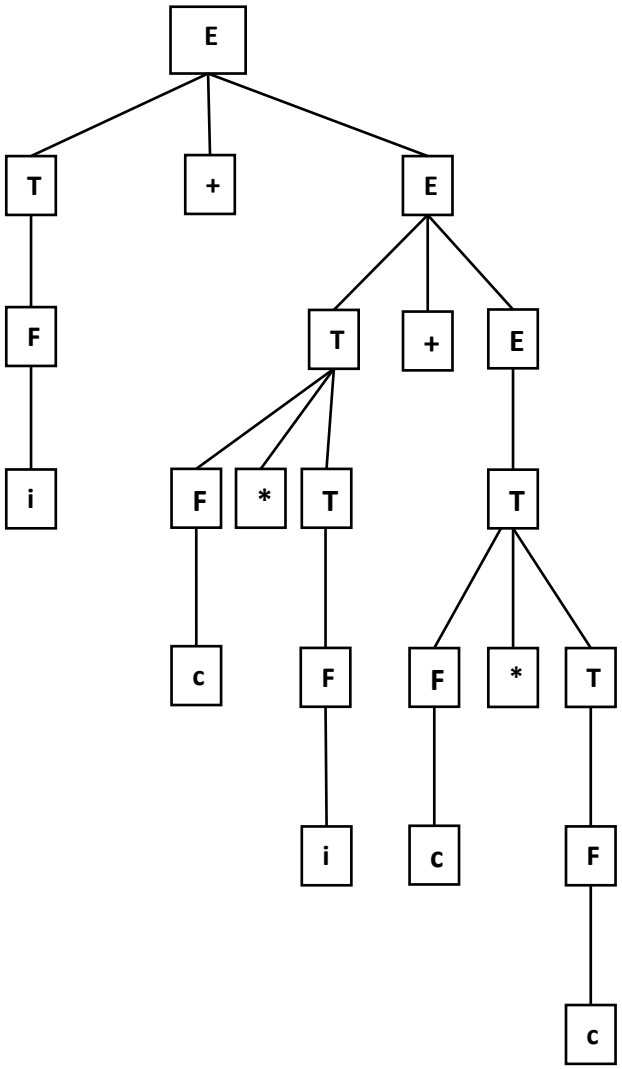
A **derivation** in a grammar starts with α_0 being the start symbol, and in each step k it derives α_k from α_{k-1} by finding a nonterminal in α_{k-1} and replacing it with the RHS of a production for that nonterminal. The derivation halts if at some step k there are no nonterminals remaining in α_{k-1}.

$\underline{E} \Rightarrow T + \underline{E} \Rightarrow T + T + \underline{E} \Rightarrow \underline{T} + T + T \Rightarrow \underline{F} + T + T \Rightarrow$

$\ldots \Rightarrow i + c * i + c * c$

The **language of a grammar** is all possible sentences that you can derive from the grammar.

A **parse tree** of a derivation is the tree one gets by placing the RHS of a production used as a step in the derivation below the LHS and as children of it.



Different derivations can lead to the same parse tree, but different parse trees must lead to different derivations.

It is the job of the **parser** to look at an input stream (of tokens) and decide which derivation or parse tree of the language that input stream belongs to, if any.

To be suitable for **predictive parsing,** a grammar must be **left-factored.**

$E \rightarrow \mathbf{T} + E \mid \mathbf{T}$ **Not left-factored**

$T \rightarrow \mathbf{F} * T \mid \mathbf{F}$

$F \rightarrow \mathbf{(}\ E\ \mathbf{)} \mid c \mid i$

$E \rightarrow T\ Y$ **left-factored**

$Y \rightarrow + E \mid \varepsilon$

$T \rightarrow F\ Z$

$Z \rightarrow * T \mid \varepsilon$

$F \rightarrow \mathbf{(}\ E\ \mathbf{)} \mid c \mid i$

It must also have no **left-recursion**. This is a situation where some $A\ \alpha \Rightarrow \ldots \Rightarrow A\ \beta$

To remove **direct left-recursion,** we look at two (left-factored) productions of the form:

$A \rightarrow A\ \alpha$

$A \rightarrow \beta$

and convert them to:

$A \rightarrow \beta\ A'$

$A' \rightarrow \alpha\ A'$

$A' \rightarrow \varepsilon$

Removing **indirect left-recursion** involves an algorithm that systematically visits the nonterminals.

There are **top-down** and **bottom-up parsers.** Our **recursive descent** parser is an example of top-down. Bottom-up parsers are based on large tables and are more flexible, but typically require a parser generator to create.

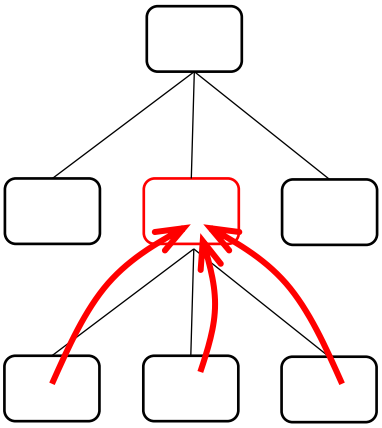
Semantic analysis

Mostly, we discussed ad-hoc semantic analysis, but we also saw **attributed grammars.** These are CFG's with rules for each production that define attributes of grammar symbols. For example,

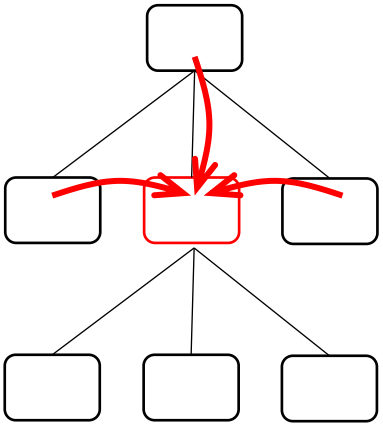
production	**rules**
$E_1 \rightarrow (E_2)$	E_1.type := E_2.type
	E_1.isConstant := E_2.isConstant

If a rule gives a grammar symbol an attribute, then each instance of that grammar symbol in the parse tree gets its own instance of that attribute. In the above grammar, all E's would get an attribute *type* and an attribute *isConstant*.

If an attribute is computed from attributes of symbols below it in the tree, it is called a **synthesized** or **synthetic** attribute. If it is computed using attributes of its siblings and its parents, it is called an **inherited** attribute.

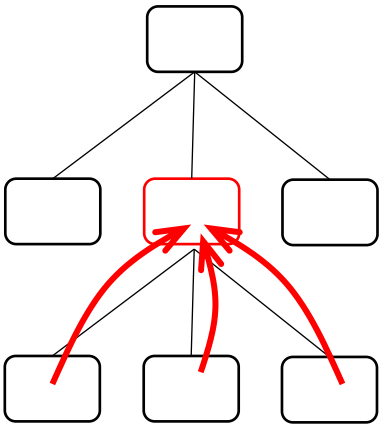


Synthesized

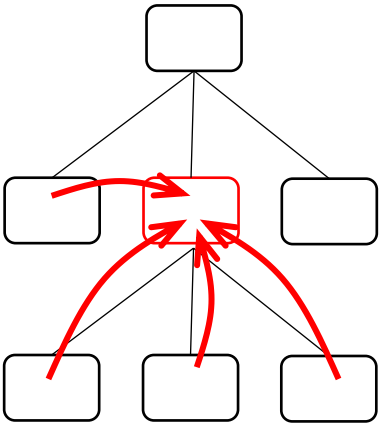


Inherited

An attributed grammar is called **L-attributed** if every attribute at a node is computable if one knows the value of all the attributes of the node's children and all of the attributes of the node's *left* siblings.



S-attributed



L-attributed

Intermediate Representations

- **Graphical IR**
  - **AST, high- or low-level**
  - **Expression DAGs**
- **Linear IR**
  - **1-address code**
  - **3-address code**
  - **basic blocks**
  - **Static Single-assignment form**
- **Hybrid IR**
  - **control-flow graphs**
  - **call graphs**

- **One-address code.** These model the behaviour of accumulator or stack machines. The resulting code is quite compact. Our ASM code is a one-address code. Java, Smalltalk, and Scala all compile to one-address code.
- **Two-address code.** These model a machine with destructive operations (operations take two operands and write the result over one of them). Not popular or useful nowadays.
- **Three-address code.** These model a machine where most operations take two operands and produce a result. The resulting code resembles code for a RISC machine. Very popular.

In three-address code (3AC) most operations have the form

i = j **op** k

A record that stores this information is often called a **quadruple.**

A **basic block** is a maximal set of consecutive linear instructions that must be executed together. A basic block starts with a **leader** that is either the first instruction in a subroutine, a labelled instruction (which can be the target of a jump or branch), or the statement after a branch. A basic block ends at the first branch, jump, or subroutine call, return or other leader after its leader.

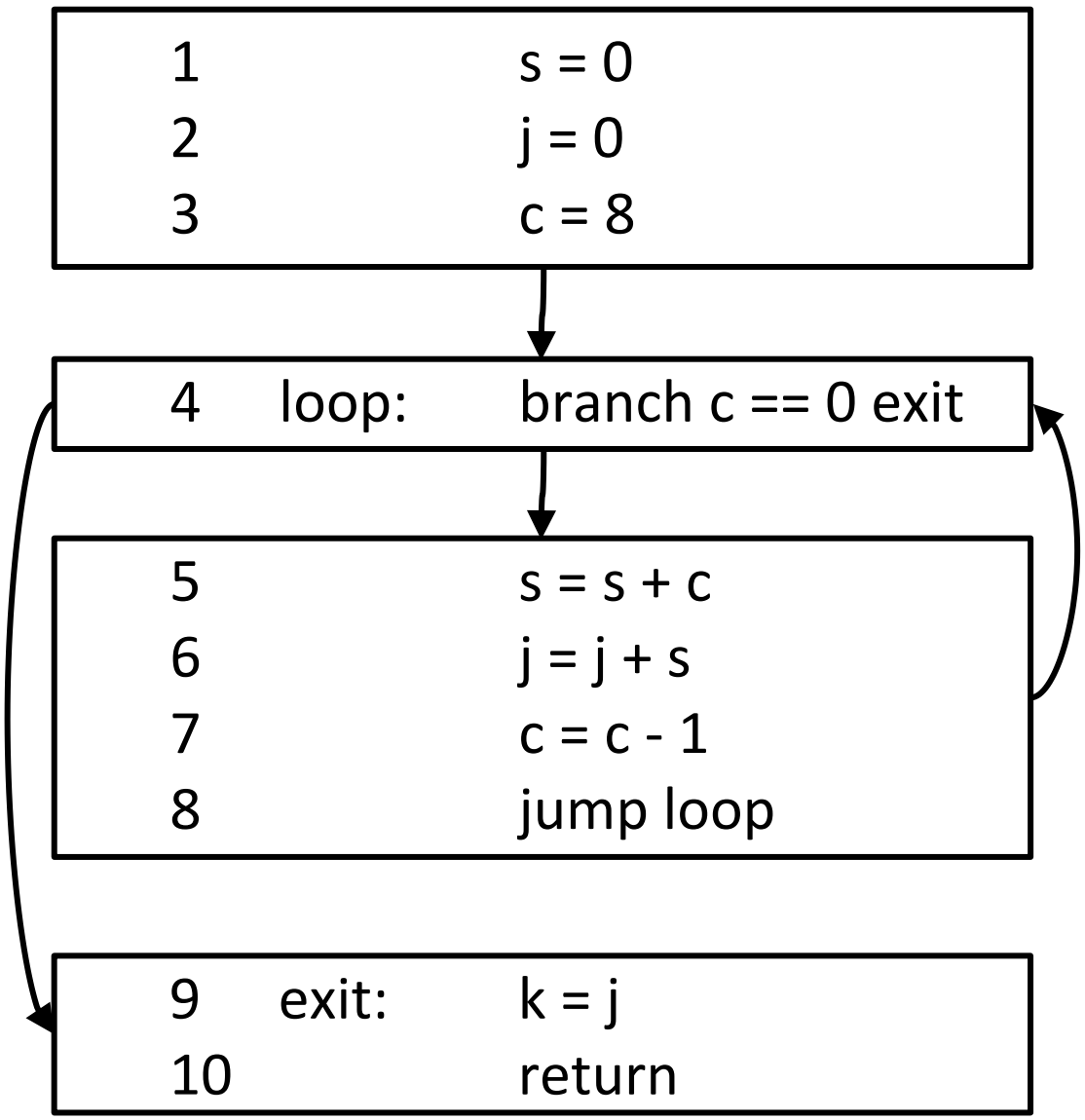
```
1                   s = 0
2                   j = 0
3                   c = 8
```

```
4     loop:         branch c == 0 exit
```

```
5                   s = s + c
6                   j = j + s
7                   c = c - 1
8                   jump loop
```

```
9     exit:         k = j
10                  return
```

If we add directed edges to show where program control can go, we arrive at the **control flow graph** (CFG) of the subroutine.



The CFG is a very common **hybrid IR.**

Static Single-Assignment form

Static Single-Assignment (SSA) is a naming and encoding discipline that encodes information about the flow of control and flow of data. In SSA, a name is only assigned to at one point in the code; each name is defined by one operation.

SSA form contains special operations, called ϕ-functions, at points where control-flow paths meet. A ϕ-function selects whichever of its arguments was last assigned to.

Original	SSA	
x = 0		$x_0 = 0$
y = 0		$y_0 = 0$
while(x < 100)	loop:	$x_1 = \phi(x_0, x_2)$
x = x + 1		$y_1 = \phi(y_0, y_2)$
y = y + x		if $x_1 \geq 100$ goto next
		$x_2 = x_1 + 1$
		$y_2 = y_1 + x_2$
		goto loop
	next:	

Optimizations

There are two issues at the heart of every optimization: **safety** and **profitability.**

Optimizations done on a single basic block are called **local** optimizations.

Optimizations done on an area larger than a single block but smaller than an entire procedure are called **regional** optimizations.

Optimizations done on a single procedure are called **global** optimizations.

Optimizations done on a bigger scale are called **interprocedural** optimization.

We saw the regional optimization of **loop induction variable analysis.**

Local optimization

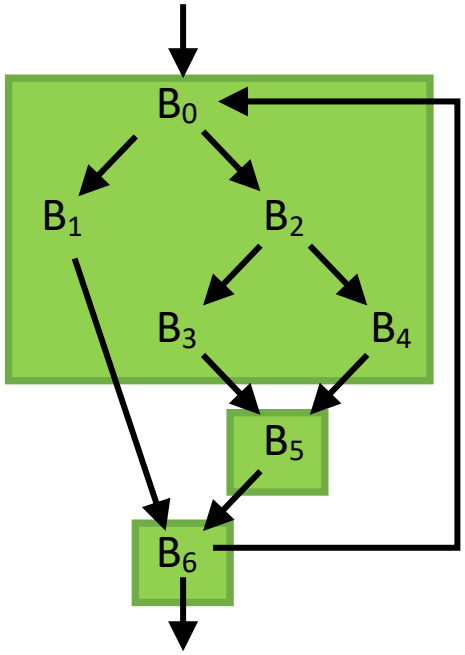
We examined **local value numbering (LVN).**

$a^2 = b^0 + a^1$
$b^4 = a^2 – d^3$
$c^5 = b^4 + a^2$
$d^4 = a^2 – d^3$

And extended it with commutative operations, constant folding, and algebraic identities.

Regional Optimization

Extended Basic Blocks (EBBs). An extended basic block is a set of basic blocks B_1, B_2, … B_n in the control flow graph (CFG), where B_1 may have multiple CFG predecessors and each other B_i has just one, which is some B_j in the set.



Superlocal value numbering. Uses EBBs as a starting structure. In a large EBB, each path from root to leaf is analyzed as a single basic block.

Loop unrolling. The oldest and best-known loop transformation. Here, the loop body is replicated and the logic controlling the loop is adjusted.

```
      do 80 i = 1, n
            sum = sum + a(i)
80    continue
```

could become:

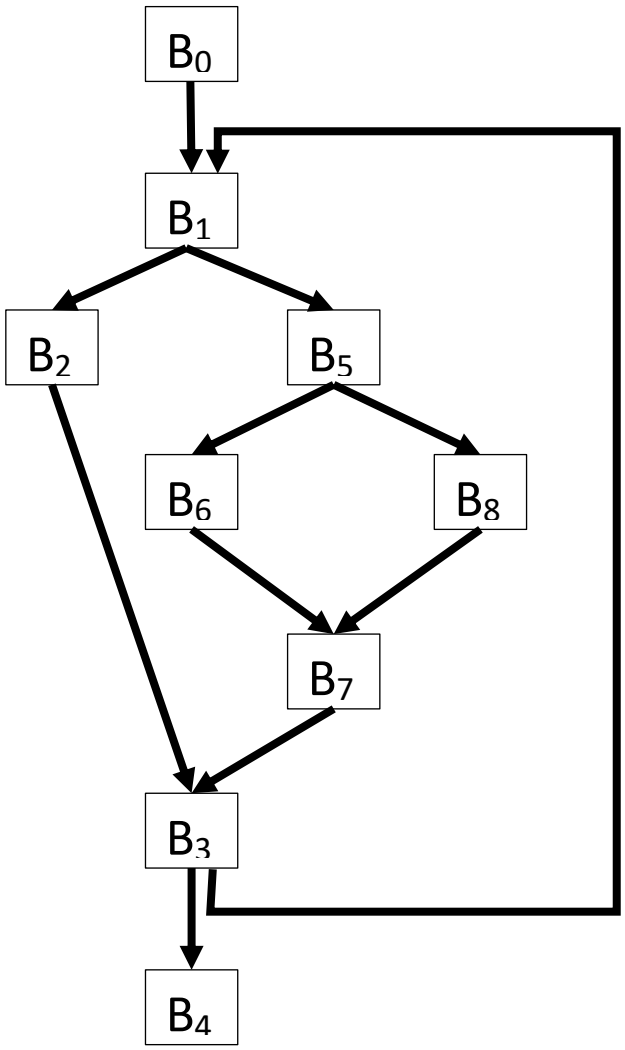
```
      n1 = mod(n, 2)
      if(n1 .eq. 1) then
            sum = sum + a(1)

      do 80 i = n1 + 1, n, 2
            sum = sum + a(i)
            sum = sum + a(i+1)
80    continue
```

Global Optimization

Live variable analysis. A variable **v** is live at point **p** if and only if there exists a path in the CFG from **p** to a use of **v** along which **v** is not redefined.

Dominance. A CFG node $\mathbf{B_i}$ dominates $\mathbf{B_j}$ iff $\mathbf{B_i}$ lies on every path from the entry node of the CFG to $\mathbf{B_j}$.



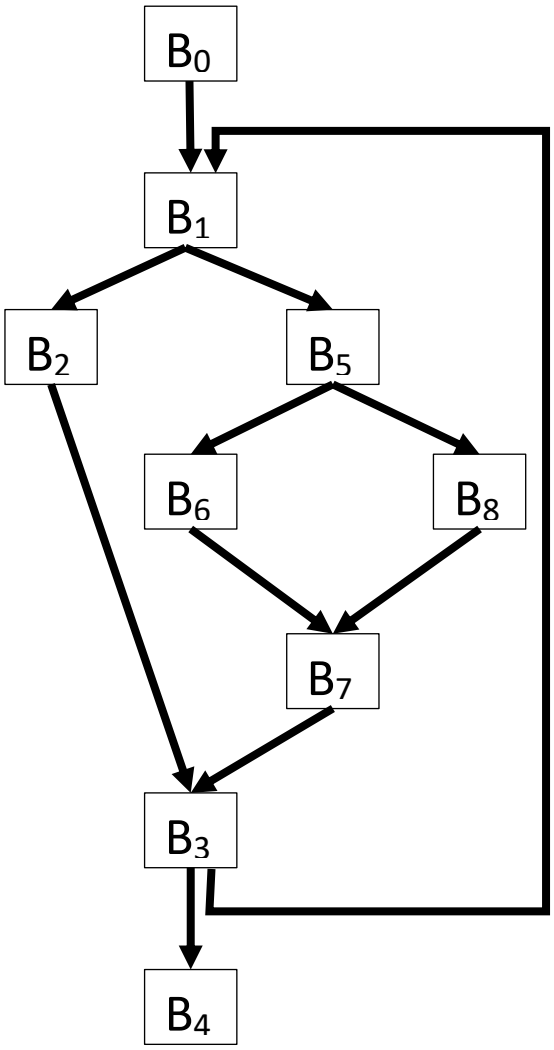
node	DOM(n)
B_0	{0}
B_1	{0,1}
B_2	{0,1,2}
B_3	{0,1,3}
B_4	{0,1,3,4}
B_5	{0,1,5}
B_6	{0,1,5,6}
B_7	{0,1,5,7}
B_8	{0,1,5,8}

$$\boldsymbol{DOM}(n) = \{n\} \cup \left(\bigcap_{m \in \text{preds}(n)} \boldsymbol{DOM}\,(m) \right)$$

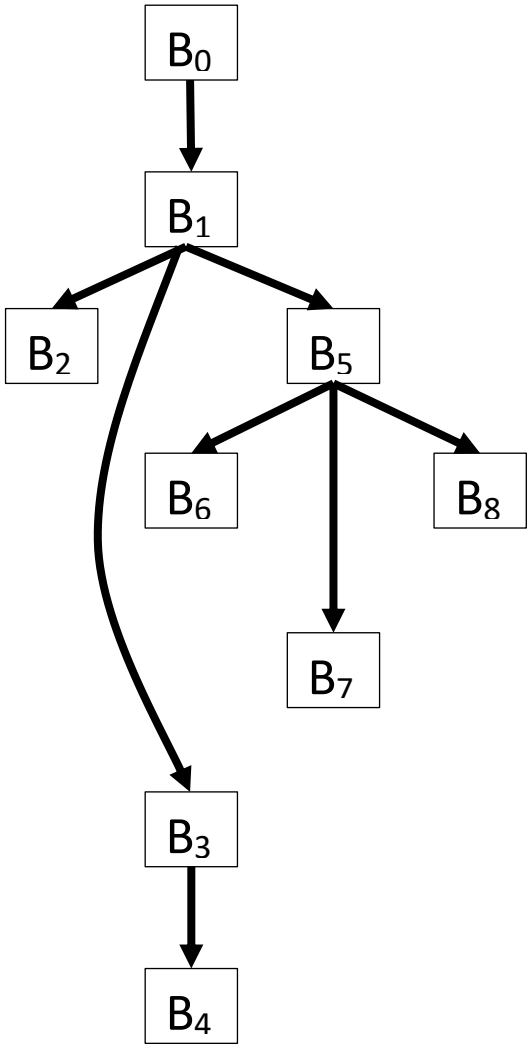
with the initial conditions that **DOM**(n_0) = {n_0} and, for all $n \neq n_0$, **DOM**(n) = N. (N is the set of all nodes of the CFG.)

Dominator Trees. The nodes that strictly dominate *n* are **DOM**(*n*) – *n.* The closest strict dominator to *n* is called its **immediate dominator**, and denoted **IDOM**(*n*). The entry node has no immediate dominator.

The **dominator tree** compactly encodes **IDOM** and **DOM** information. It consists of edges (**IDOM**(*n*), *n*) for each node n.



Example CFG



Dominator tree

Dominance Frontiers. The dominance frontier **DF**(*n*) of a node *n* is all nodes *m* where

1. *n* dominates a predecessor of *m*, and
2. *n* does not strictly dominate *m*.

These are the nodes "just outside" the dominance of *n*.

Static Single-Assignment Form

We saw how to translate normal 3AC into Maximal SSA and into Semipruned SSA forms. Both involve

1. Placing ϕ-functions
2. Renaming

We also saw how to translate from SSA back to normal 3AC. This involved **splitting** edges and inserting new basic blocks inside them.

Finally, we saw Sparse Simple Constant Propagation (SSCP), an optimization which gains a great benefit from using SSA form. SSCP is an **optimistic** algorithm.

Interprocedural Optimization

We looked at **call graph computation**, which is difficult in the presence of indirect function calls.

We also had an overview of **interprocedural constant propagation (ICP).**

Scalar Optimizations

Scalar optimization is optimization of code for a single thread of control. We looked at these with an eye towards five objectives:

- Eliminate useless and unreachable code
- Code motion
- Specialization
- Redundancy elimination
- Enable other transformations

Eliminating Useless and Unreachable Code

An operation can be **useless**, meaning that its result has no externally visible effect. Alternatively, the operation can be **unreachable**, meaning that it cannot execute. If an operation falls into either category, it can be eliminated. We use the term **dead code** to refer to such code.

We saw a **mark-and-sweep** algorithm for determining and eliminating dead code. We also saw an algorithm **Clean** for eliminating useless control flow.

Code Motion

Compilers perform code motion for two primary reasons. Moving an operation to a point where it executes fewer times than it would in its original position should reduce execution time. Moving an operation to a point where one instance can cover multiple paths in the CFG should reduce code size.

We examined **partial redundancy elimination (PRE),** which moves code to where it is executed less often.

Specialization

Compiler front-ends produce general code that works in any context the running code might encounter. With analysis, it can learn enough to narrow the contexts in which the code must operate. This creates the opportunity for the compiler to **specialize** the sequence of operations in ways which capitalize on this knowledge of contexts.

We looked in particular at **tail-call optimization** and **leaf-call optimization**.

Redundancy Elimination

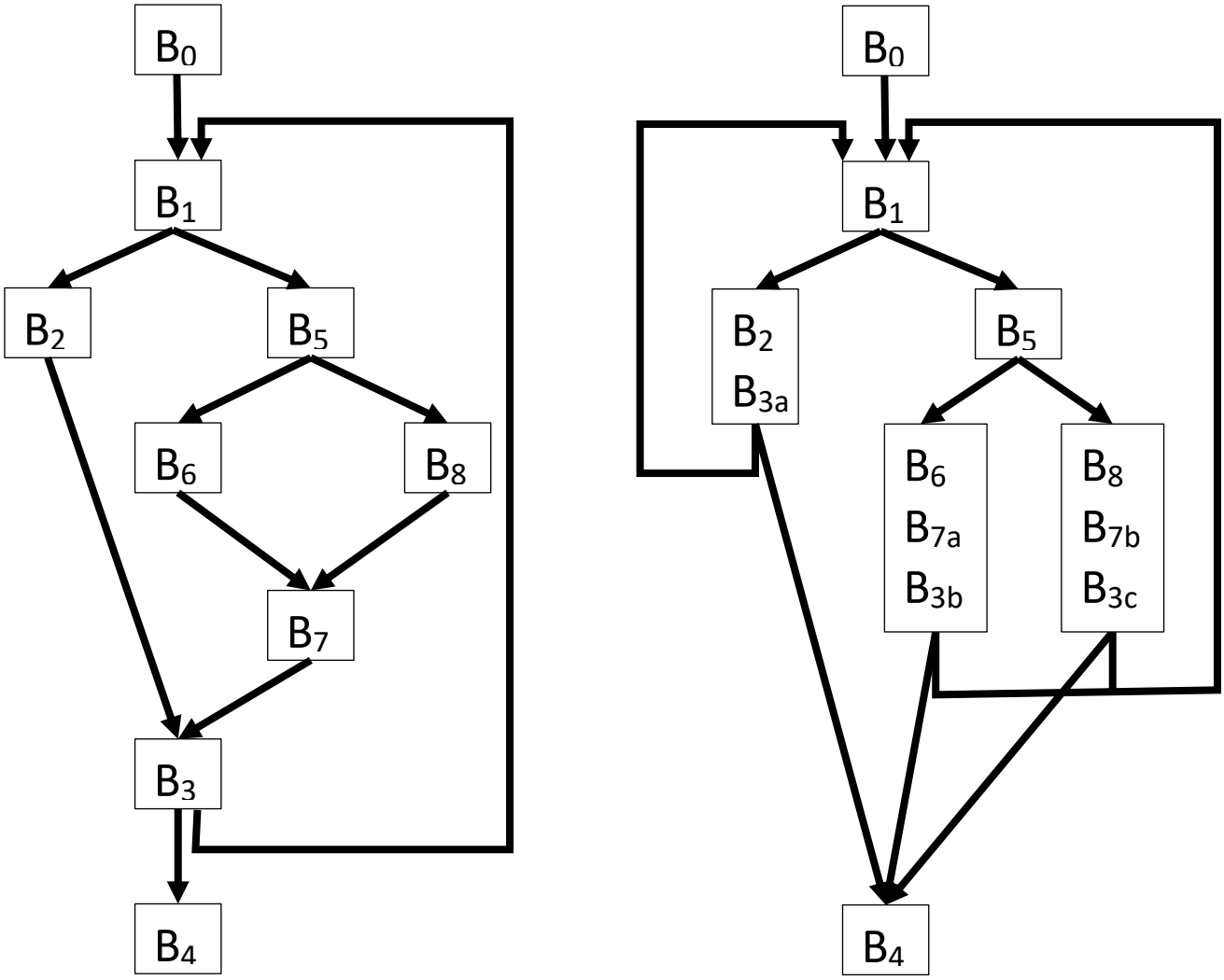
A computation x + y is redundant at some point *p* in the code if, along every path that reaches *p*, x + y has already been evaluated and x and y have not been modified since the evaluation. Redundant computations typically arise as artifacts of translation or optimization.

We saw three techniques that eliminate redundancy: LVN (Local value numbering), SVN (Superlocal value numbering), and PRE (partial redundancy elimination).

Enabling other transformations

Often, a compiler includes passes whose primary purpose is to expose opportunities for other transformations. Loop unrolling and inline substitution are such transformations. We studied another, namely **superblock cloning.**

Superblock cloning transforms the CFG on the left to the CFG on the right.



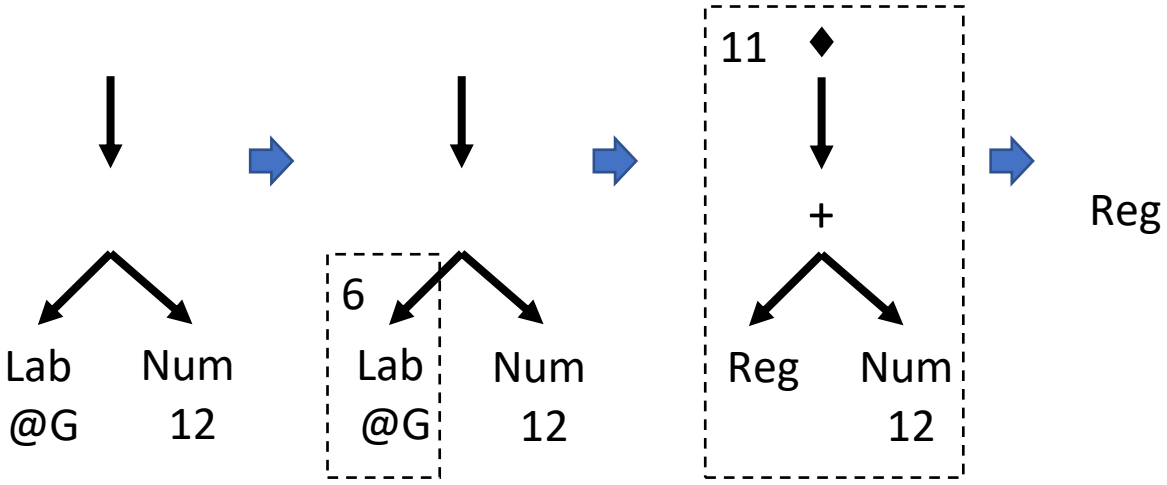
The back end

Our text uses the term “back end” to refer to the part of the compiler that must know about the target machine architecture. Increasingly, modern compilers are dividing this back end into three phases, dealing with three concerns. These phases are

1. Instruction selection.
2. Instruction scheduling.
3. Register allocation.

We studied **instruction selection** and **register allocation**.

The particular type of instruction selection we saw was **instruction selection via tree-pattern matching**. It relies on a set of **rewrite rules** which capture the function and cost of different machine instructions.



Instruction selectors are most often generated by a program that takes the rewrite rules or other architecture description as its input. There are several strategies available for writing instruction selectors.

Register Allocation

The register allocator determines which values reside in registers and which register will hold each of these values. If the allocator cannot keep a value in a register throughout its lifetime, the value must be stored in memory for some or all of the time. This is called **spilling** a value.

Most processors have distinct classes of registers for different kinds of values. For example, most modern processors have both **general-purpose registers** and **floating-point registers**. Some processors have classes of **condition code registers**, or **branch-target registers**.

We saw two algorithms for **local register allocation: top-down** and **bottom-up**. In **top-down**, the allocator ranks variables by their number of uses in the block, assigning higher-ranked variables to registers. In **bottom-up**, the choices of what to keep and what to spill are made locally in a front-to-back pass through the block.

For **global register allocation**, we spoke of one algorithm. It is based on the **live ranges** of SSA names and their **interference graph.** Basically, it assigns registers by coloring the vertices of the interference graph.