# ECE 351 Lecture Notes

## Types of Exam Questions

1. Prove a theorem
   1. Properties of classes of languages
2. Give examples of DFAs, NFAs, regular expressions, regular languages, undecidable, CFL, PDA
3. Counter example
4. NFA -> DFA
5. Prove non-regularity

## 2. Basics of Formal Languages and Computability

### Formal Languages

* **Alphabet** – a finite set of symbols/characters
  + E.g.
* **String** – a sequence (concatenation) of symbols from the alphabet of a language
  + / represent the **empty string**
* **Language** – a possibly infinite set of finite-length strings over a (finite) alphabet
  + **Universal language** – is the set of all strings over an alphabet
    - Every language is a subset of
  + **Finite languages** – the cardinality of the set of words in the language is a finite number

### Typical Operations on Strings

* **Concatenation(s1, s2)**
  + Non-communitive –
  + Associative –
* **Prefix(s1) / Suffix(s1)**
  + For the string :
    - is a prefix
    - is a suffix
  + is a prefix and suffix of every string
  + Every string is a prefix and suffix of itself
* **Reverse(s1)** 
  + The reverse of is
  + The reverse of is
* **Length(s1)**
  + Computes the length of a string

### Typical Operations on Languages

* **Union:** 
  + Set of strings in either/both or
* **Intersection**:
  + Set of strings in both and
* **Difference**:
  + Strings in but not
* **Complement**:
  + Set of all strings except those in
* **Reverse**:
  + Set of all reversed strings in
* **Concatenation**:
  + Set of strings that has a prefix from and suffix from
* **Kleene Star**:
  + The set union of concatenations of strings from

### Reverse of a Language

Definition:   
i.e. the reverse of a language is the set of the reverse of every word in the language

Example: For ,

### N-ary Concatenation of Two Languages

Definition:

Special case: For any language ,

Example: For

### Star Closure / Kleene Star

Definition:   
i.e. the set of all strings that can be constructed from a language

Example: For ,

### Positive Closure

Definition:   
i.e. the star closure, without the empty string

### Star and Positive Closures on Alphabets

is the set of all possible strings from the alphabet

is the set of all possible strings from the alphabet , except the empty string

### Two Special Languages

* **Empty language** –
* **Language with the empty string** –

### Formal Languages and Computation

* All computational problems can be described as a language/set membership problem, in the form: “given a string and a language , does there exist an algorithm to determine if belongs to ?”
* The solution to the problem is an algorithm that takes input and correctly decides if belongs to

### Languages and Turing Machines

* **Turing machine**: a finite state machine with a potentially infinite read-write tape
* Any computable algorithm can be implemented as a program on a Turing machine
  + We say that a computational problem has a solution if there exists a Turing machine that correctly decides for any string , if belongs to
  + The Turing machine can:
    - Correctly say that is in and halt
    - Correctly say that is not in and halt
    - Loop forever

### Decidability of Languages

* A language is **Turing-acceptable** if there exists a Turing machine that determines if any input exists in a language :
  + If – halts in the accept state
  + If – either halts in a non-accept state or loops forever (never terminates)
* A language is **decidable** if there exists a Turing machine that determines if any input exists in a language and always halts:



* + If – halts in the accept state



* + If – halts in a non-accept state



* A computational problem is decidable, and thus solvable, if its corresponding language is decidable
* Every decidable language is Turing-acceptable, but not vice versa

### Halting and the Halting Problem

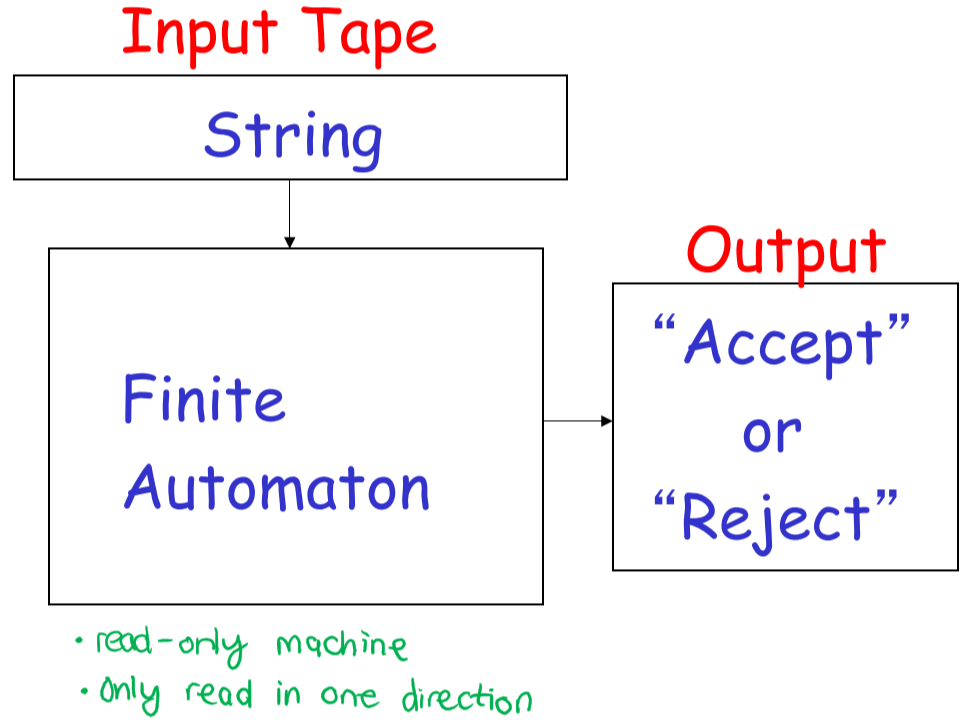
* A Turing machine will **halt** in a state if:
  + There are no transitions out of the state, or
  + There is no corresponding transition out of the current state based on the next input
* **Halting problem**: for a given input to a Turing machine , does halt while processing ?
  + By the diagonalization proof, the halting problem is undecidable, and thus unsolvable
  + However, the problem is Turing-acceptable

### Static Program Analysis is Undecidable

* Static program analysis is the analysis of a program without actually executing it
* We can show that static analysis, and code optimization by extension, is Turing-acceptable but not decidable:
  + Assume that static analysis is decidable
  + It then follows that we can easily analyze any program and determine if it will halt on an input
  + If this is the case, we have solved the halting problem, so this cannot be possible

## 3. Deterministic Finite Automata and Regular Languages

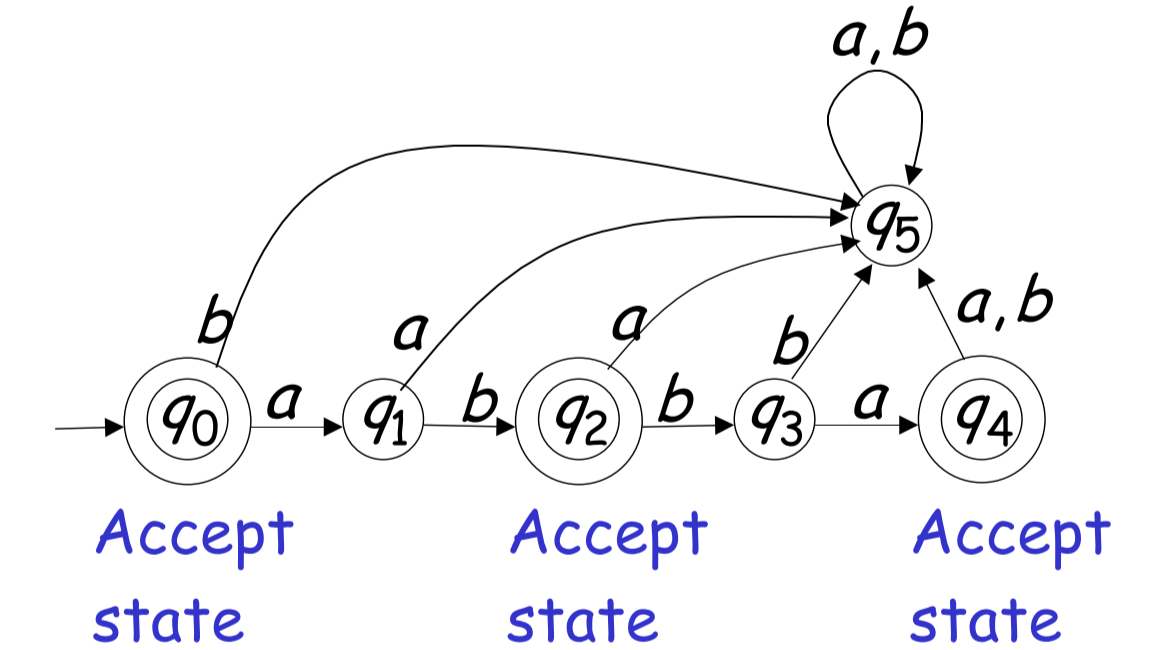
### Deterministic Finite Automaton (DFA)



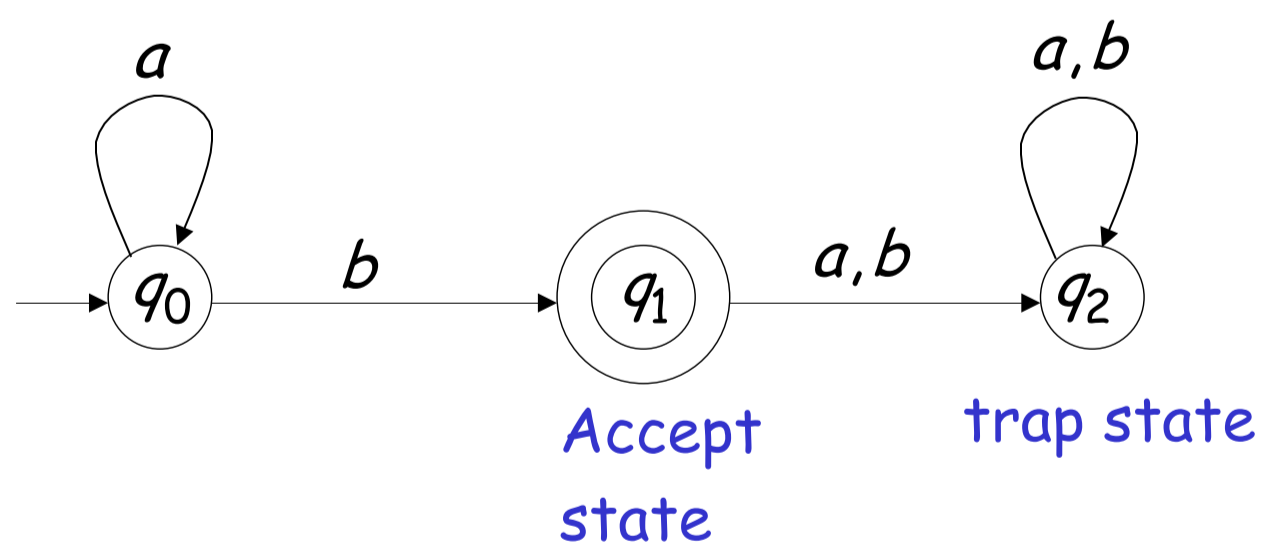
* A state machine that accepts or rejects a string of characters from an alphabet
  + For every state, there is a transition from every symbol in the alphabet
  + To accept a string, every character in the input is scanned, and the last state is accepting
  + To reject a string, every character in the input is scanned, and the last state is non-accepting



* Example: In the above DFA, the only language that is accepted is ; all other strings will end up in a non-accept state
* Example: The following DFA accepts
  + In order for a DFA to accept the empty string, q0 must be an accept state



* Example: The following DFA accepts , i.e. any number of a’s ending with a b



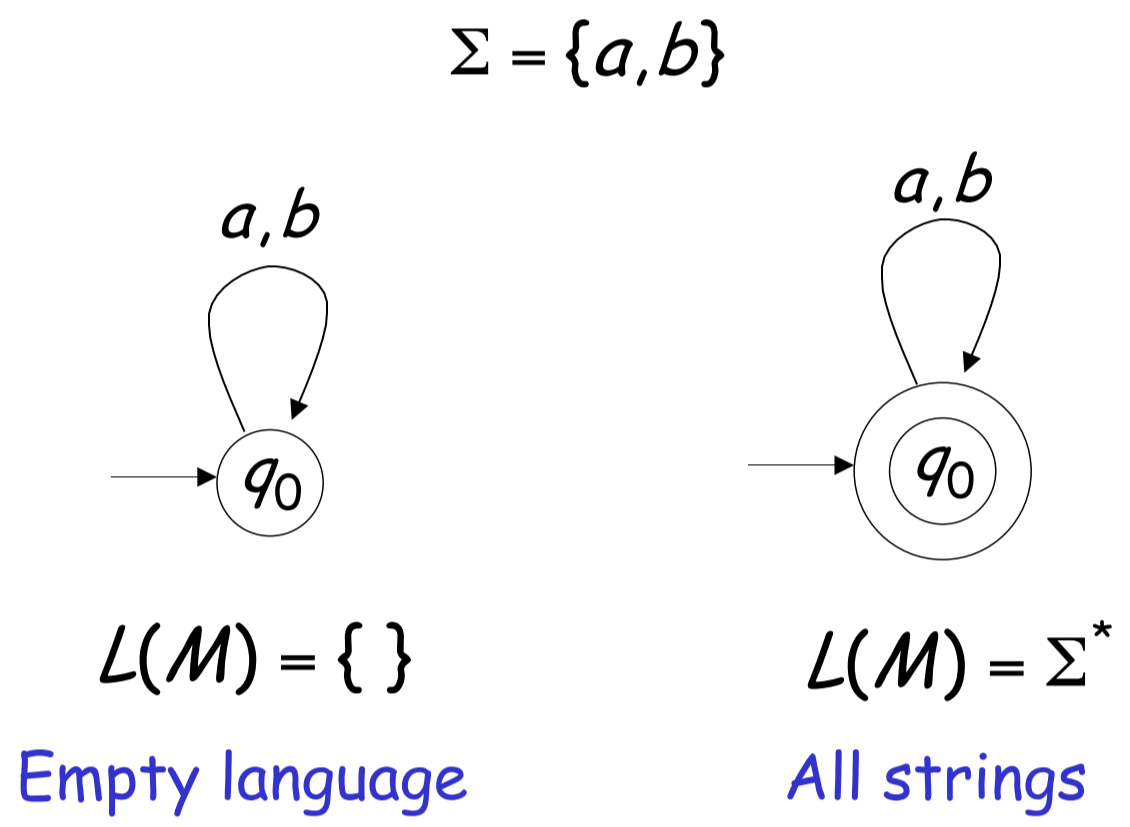
### Formal definition of DFA



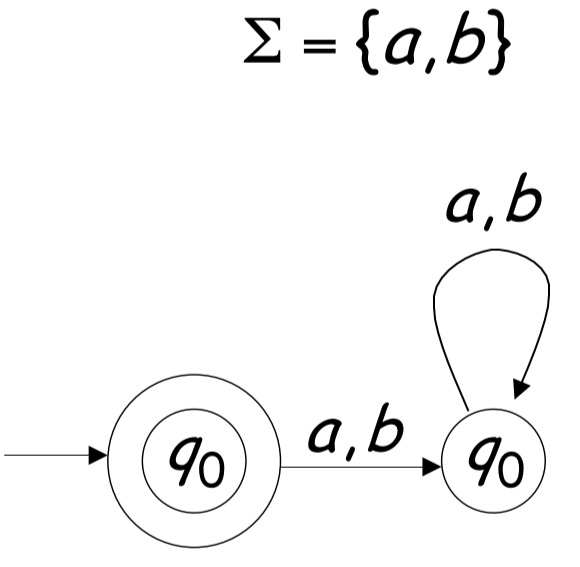
* – set of states
* – input alphabet
  + Must satisfy (the alphabet doesn’t contain the empty string)
* – transition function
  + describes a transition between states and , on the character
  + Extended transition function: describes the resulting state after scanning a word from state (i.e. a walk of transitions)
* – initial state
* – the set of accepting states (a subset of )

### Language Accepted by a DFA

* The language accepted by a DFA is denoted , and is the set of all strings accepted by
* For a DFA ,   
   i.e. the set of all strings that result in an accepting state
* Example: Any DFA with no accepting states accepts the empty language (). Any DFA with only accept states () accepts the universal language ().





* Example:
  + 

### Non-Deterministic Finite Automata (NFA)

* NFAs have two additional properties:
  + Can transition **on an empty string** from one state to another
  + Can transition to **multiple states on the same character**
* All NFAs can be transformed to have a **single accept state**
* All DFAs are NFAs by definition

### Regular Languages

* **Regular languages** are languages that have a DFA or NFA that accepts it –
* The family of regular languages contain the set of all languages that are accepted by DFAs and NFAs
  + Theorem: The set of languages accepted by DFAs can be accepted by NFAs, and vice versa
* Languages can be **non-regular**, such as ; no DFAs/NFAs can accept these languages

### Properties of Regular Languages



* Regular languages are closed under:



* 1. Union



* 1. Concatenation



* 1. Star operation



* 1. Reversal



* 1. Complement



* 1. Intersection



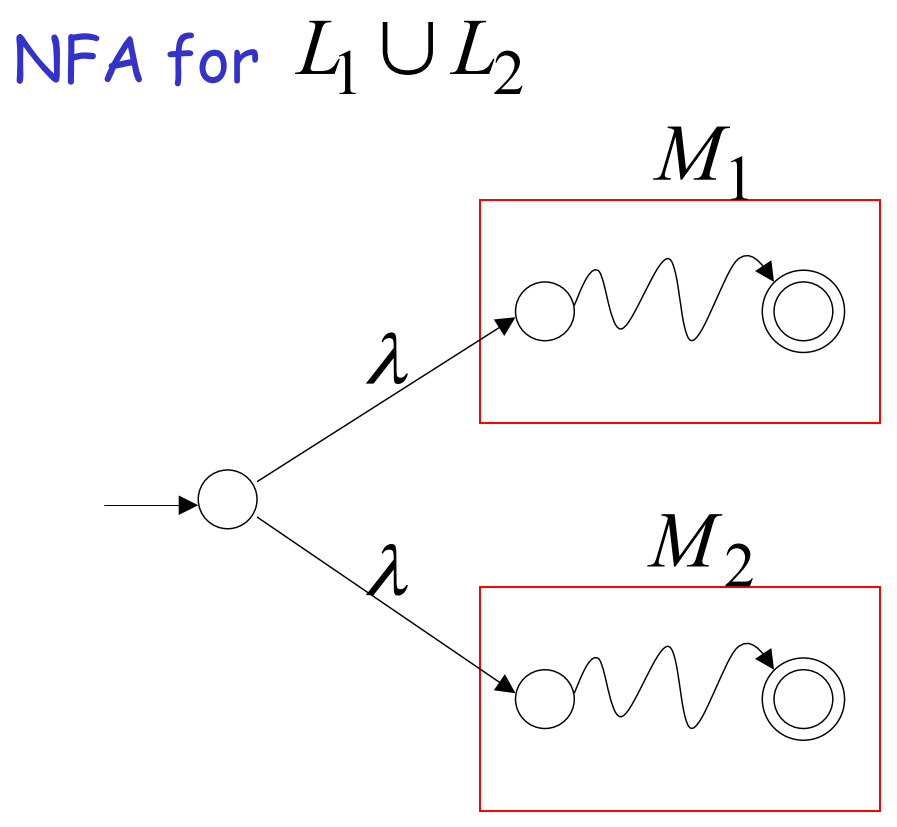
* This means that the resulting language from those operations is a regular language

For two regular languages and , with corresponding NFAs and :



#### Union

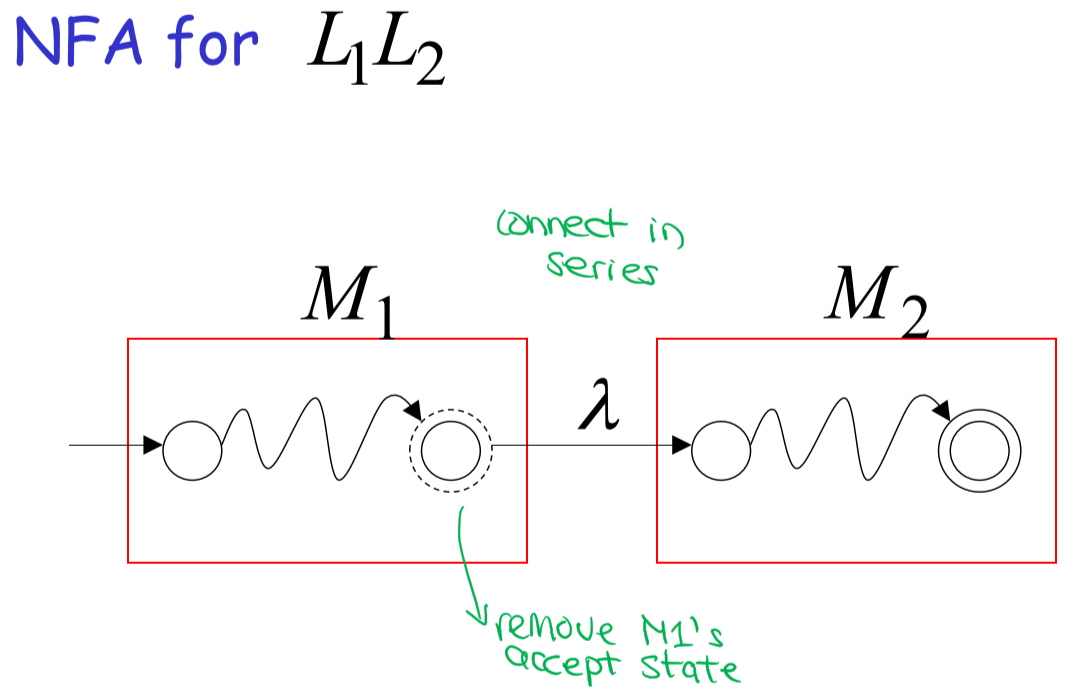
* Create a new start state with empty string transitions to the start states of and





#### Concatenation

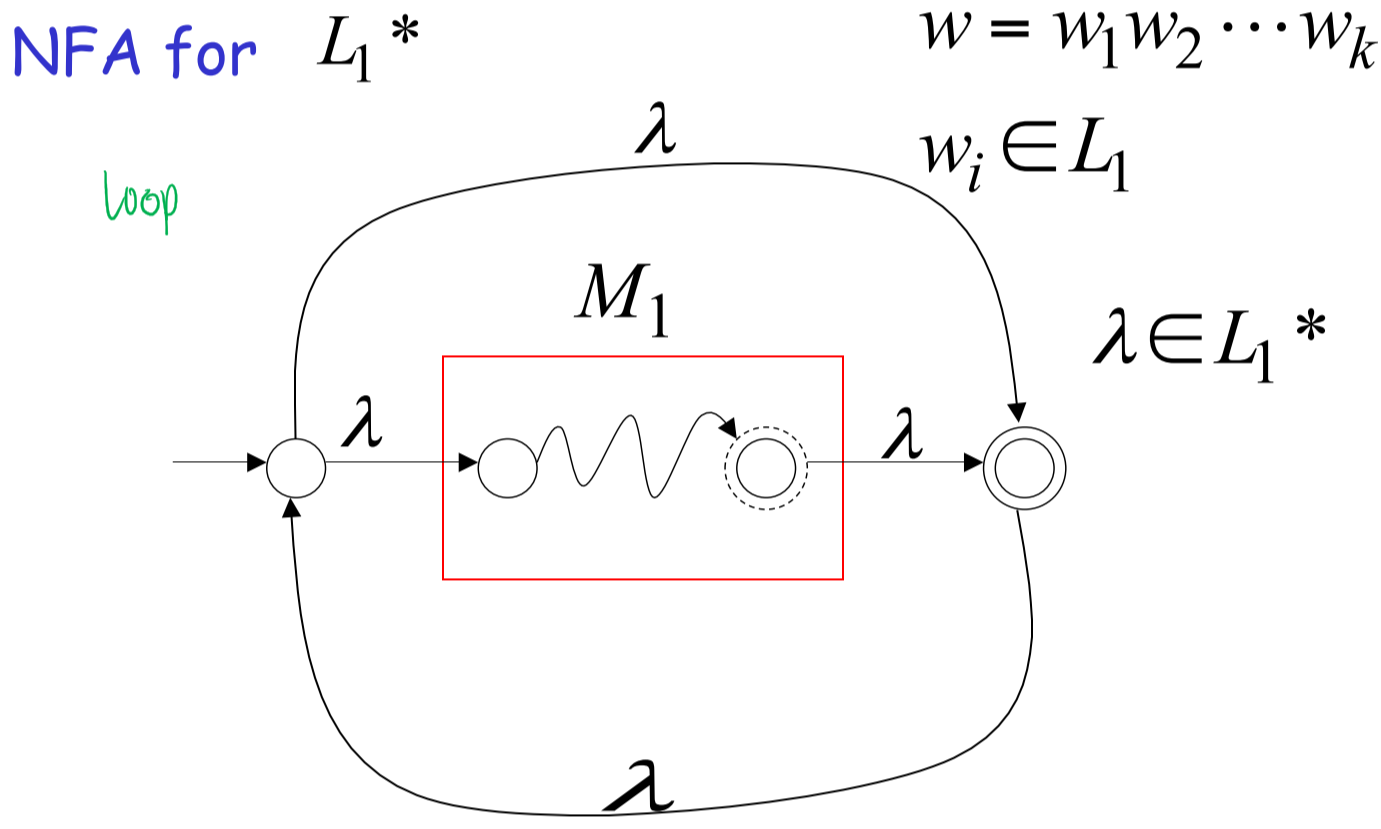
* Connect and in series, removing ’s accept state





#### Star Operation

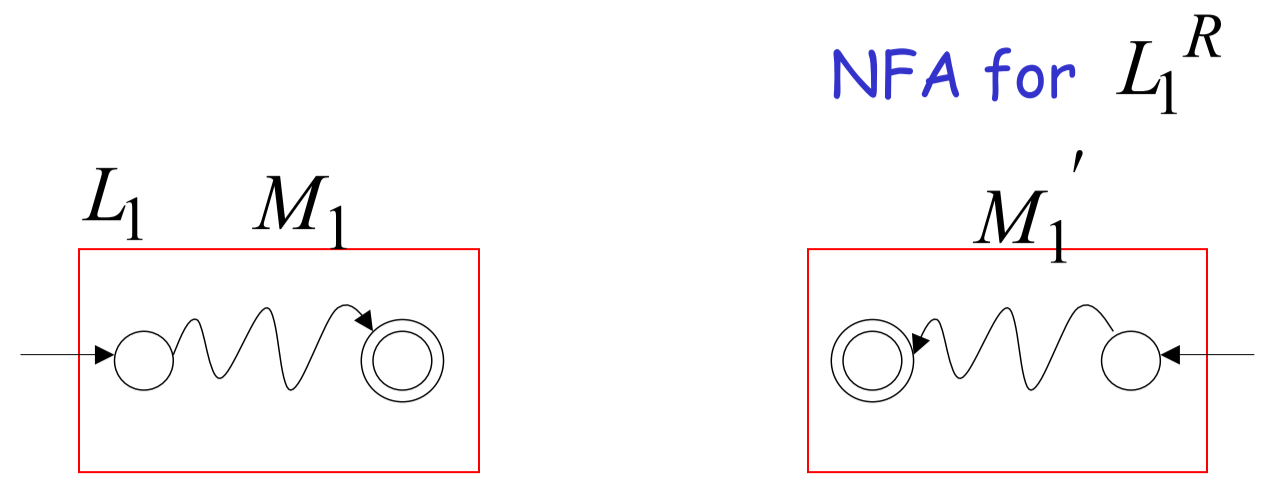
* Create new start and accept states, and make a loop of empty string transitions





#### Reverse

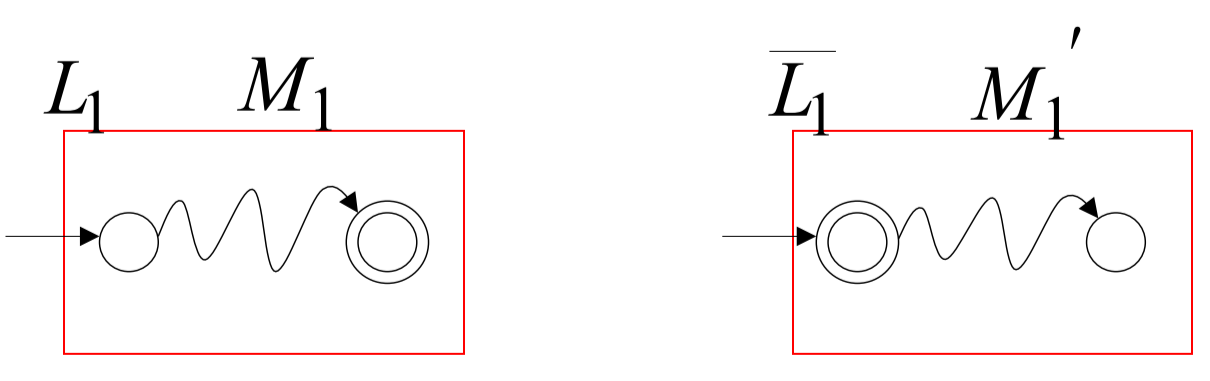
* Reverse all transitions, and swap the accepting and start states





#### Complement

* Change all accepting states to non-accepting, and all non-accepting to accepting states





#### Intersection

* By **DeMorgan’s Law**,
* Since regular languages are closed under the union and complement operations, it is also closed under intersection
* Alternatively, can construct a new DFA that simulates and in parallel
  + Only accepts a string if both and accepts it



### Regular Expressions

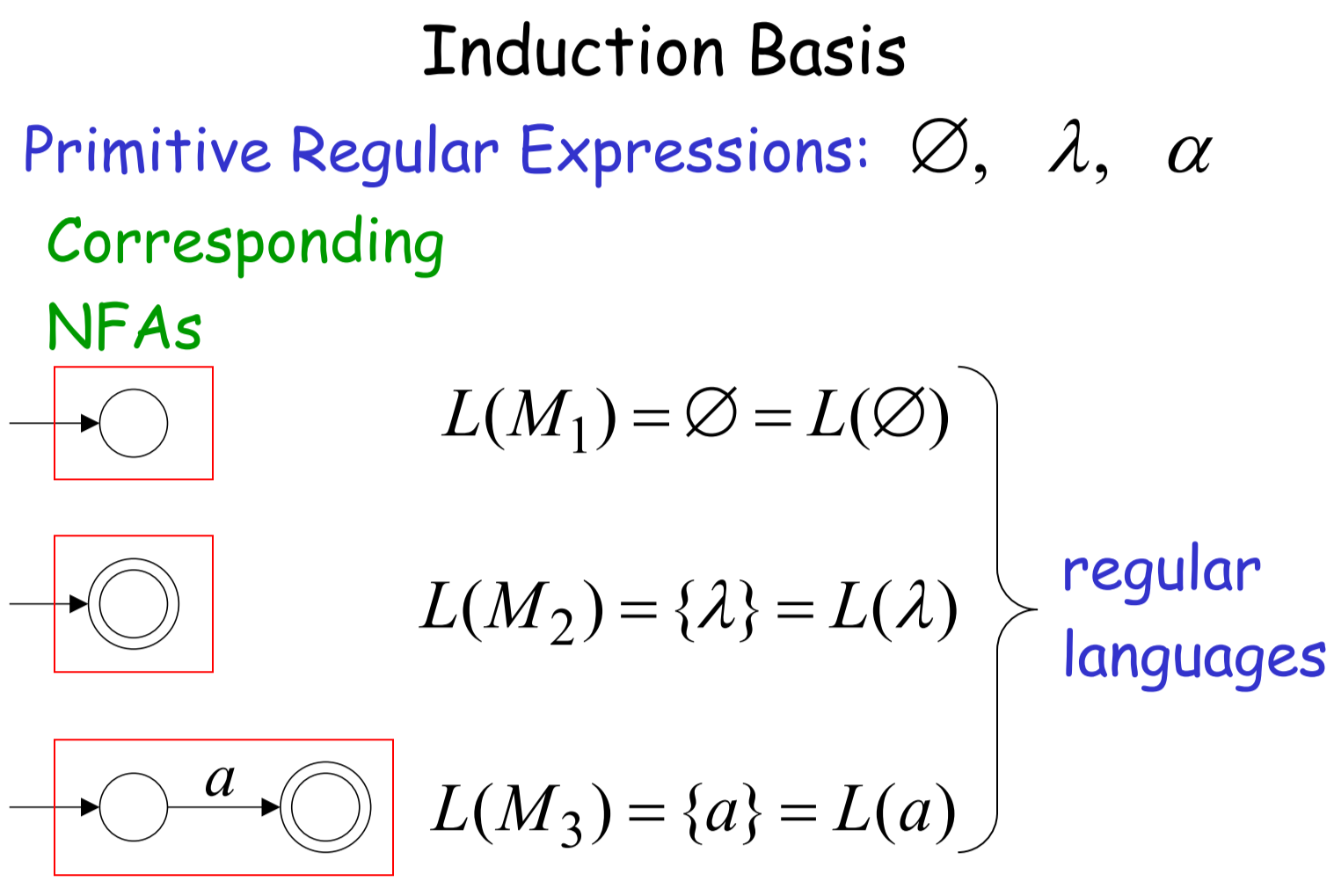
* **Regular expressions** describe regular languages; for every regular expression, there exists a regular language, and vice versa
* Primitive regular expressions:
  + – empty set
  + – empty string
  + Single character
* The language of a regular expression is denoted
* Example:
* Given regular expressions and , the following are regular expressions:
  + :
  + :
  + :
  + :
* Example:
* Example:
* Example:

### Equivalent Regular Expressions

* Two regular expressions and are **equivalent** if

### Regular Expressions and Regular Languages

* Theorem: The set of all languages generated by regular expressions is equivalent to the set of all regular languages





* **Part 1**: Languages generated by regular expressions Regular languages
  + Suppose that for regular expressions and , and are regular languages.



* + - so, it is a regular language (closed under union)
    - so, it is a regular language (closed under intersection)
    - so, it is a regular language (closed under complement)
  + Thus, any regular expressions can be created from these operations on the primitive regular expressions, and will result in a regular language, since from these closure properties, we can construct a corresponding NFA that accepts it



* **Part 2**: Regular languages Languages generated by regular expressions
  + For any regular language , there is an NFA that accepts it



* + From , construct the **generalized transition graph**, where all transitions are changed into regular expressions



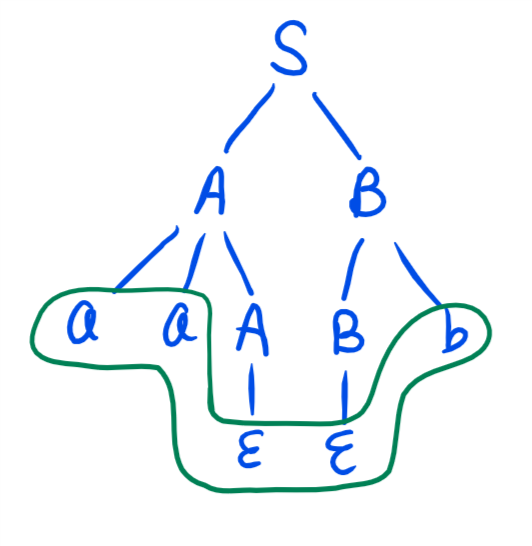
* + Thus, since any NFA can be reduced down to a regular expression, any regular language must be a language generated by a regular expression

## 4. Context-Free Grammars and Languages

* + set of variables/non-terminals
  + set of terminals
  + start variable
  + set of production rules
* A language is context-free if there is a context-free grammar such that
* Example:
* Example:
  + – palindromes
* Example:
  + Describes matched parentheses

### Derivation Order and Derivation Trees

* **Leftmost derivation order**: at each step, substitute for the leftmost variable first
* **Rightmost derivation order**: at each step, substitute for the rightmost variable first
* Example:
  + Derive the string .



### Ambiguity

* A context-free grammar is **ambiguous** if there is a string that has:
  + Two different derivation trees
  + Two different leftmost derivations



* Example:
  + Ambiguous grammar
  + Non-ambiguous equivalent grammar

### Precedence and Associativity Declarations



* Instead of rewriting the grammar, most tools use associativity and precedence declarations to disambiguate grammars



* + E.g. always use left associativity: int + int + int 🡪 (int + int) + int



* + E.g. has precedence over : int + int \* int 🡪 int + (int \* int)



### Properties of Context-Free Languages

* CFLs are closed under:
  1. Union



* 1. Concatenation



* 1. Star operation



* They are **not** necessarily closed under:



* + Intersection



* + Complement

#### Union

* Introduce a new start state
* Example:



#### Concatenation

* Introduce a new start state
* Example:



#### Star Operation

* Example:



### Regular Closure Lemma

* The intersection of a context-free language and a regular language is a context-free language



* + For a context-free language with NPDA and a regular language with DFA



* + Construct a new NPDA that runs and in parallel, and only accepts a string if both and accept it



* Example: Prove that is context-free.
  + We know that is context-free
  + We know that is a regular language; since regular languages are closed under complement, is also a regular language
  + By regular closure, is context-free
* Example: Prove that is not context-free.
  + We know that is a regular language
  + If is context-free, then is context-free
  + However, since it isn’t, this means that cannot be context-free

## 6. Non-Regular Languages and Pumping Lemma

### Pigeonhole Principle and DFAs

* Consider a DFA with states and a string
* By pigeonhole principle, if , at least one state must be repeated in the walk through the DFA

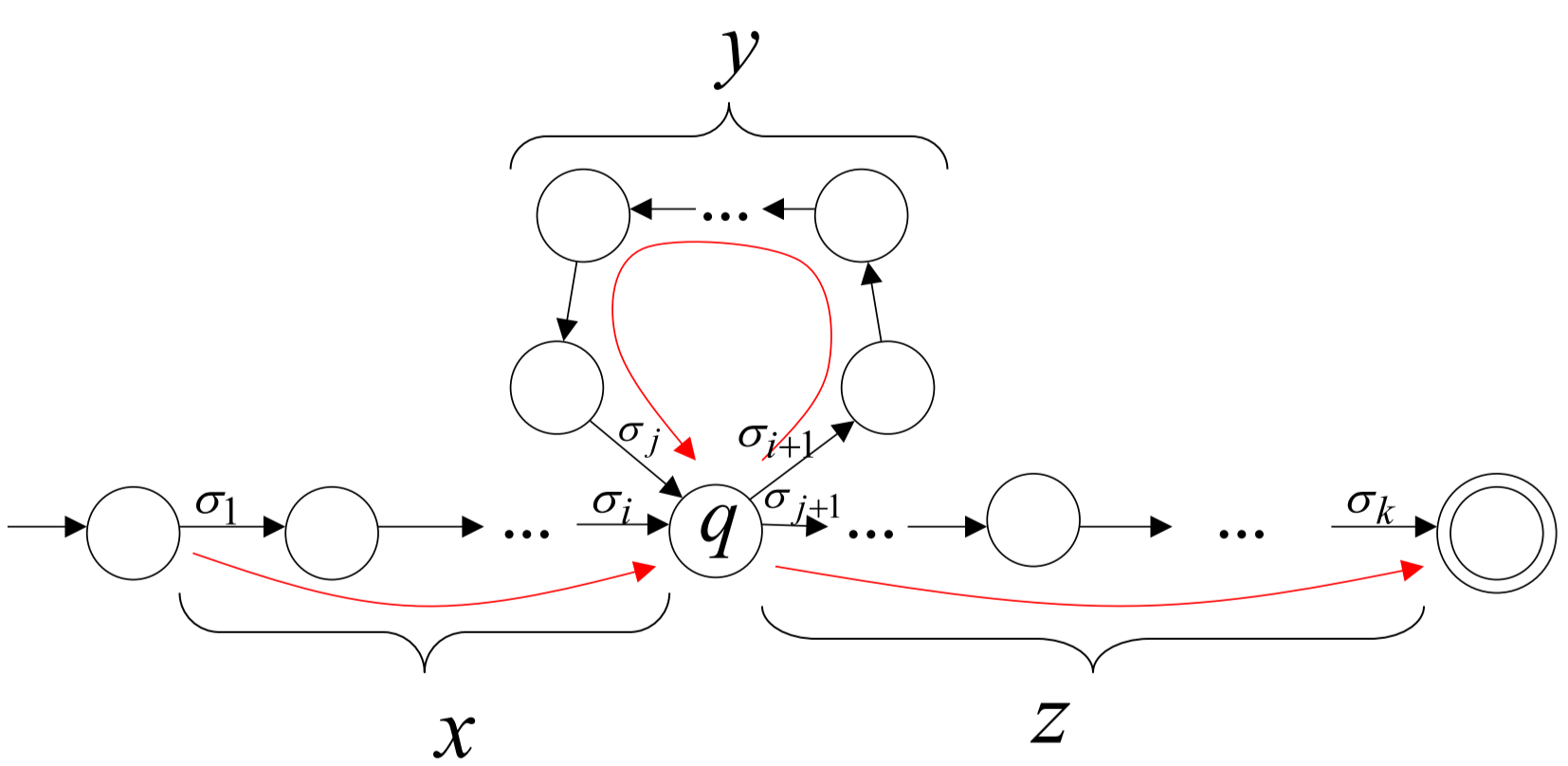
### Pumping Lemma

* Consider an infinite language , with a DFA that accepts it containing states
* For any string with , at least one state in the DFA is repeated in the walk of



* + Say that is the first repeated state
* If we write , where is the substring between the first and second occurrence of , then number of transitions in must be less than or equal to the number of states in the DFA:



* + 
* Furthermore, since there is at least one transition in the loop (from to )
* In general, is accepted by the DFA; thus,

**Example:** Prove that the language is not regular.

Assume that is a regular language; then, there exists a DFA with states, which accepts it. Since is an infinite language, apply pumping lemma.



Pick a string such that :



Divide ; we can say that



By pumping lemma, if is a regular language, then . This means that , which means that .



But , hence . This proves by contradiction that is not a regular language.

## 7. Lexical Analysis

* **Tokens**: sets of strings that classify program substrings according to their role
  + Identifiers
  + Integers
  + Keywords
  + Whitespace
* Lexical analysis outputs a stream of tokens that are the input to the parser, which needs to know the distinction between types of tokens
* Designing a lexical analyzer:
  + Define a finite set of tokens
  + Describe which strings apply to each token



* A lexical analyzer does two things:
  + Recognize substrings corresponding to tokens



* + - Need to partition the string, recognizing tokens one at a time



* + - Lookahead may be required, to decide where one token ends and the next begins



* + Return the **lexeme** – string value of the token



### Regular Languages and Tokens

* Regular languages are a popular way to specify tokens
* Atomic regular expressions
  + Single character:
  + Epsilon:
* Compound regular expressions
  + Union:
  + Concatenation:
  + Iteration:
* Example: Phone numbers
* Example: Email addresses

## 8. Lexer Implementation

### Regular Expressions in Lexical Specification

1. Write a regular expression for the lexemes of each token
2. Construct a regular expression , which matches the lexemes of all tokens
3. For an input string with characters , check if for all
4. If success, then we know that , where is a regular expression for a token
5. Remove from the input and go back to step 3

### Ambiguities

* Always pick the **longest possible string** in that is a match
* Always pick the **rule listed** **first** – order of rules matter
* Need a rule to match all “bad” strings, placed at lowest priority

### Lexer Implementation

* Regular expressions are lexer specifications
* Finite automata are lexer implementations
* Regular expression NFA DFA Table-driven implementation of DFA

### Powerset Construction (NFA DFA)

1. For the initial state of the NFA , and any walks on the empty string , the corresponding initial state in the DFA is
2. For every DFA state and symbol in the alphabet ,   
   compute in the NFA   
   and add a transition to the DFA
3. For every DFA state where some is an accepting state in the NFA, make an accepting state in the DFA

## 9. Introduction to Parsers and Recursive Descent Parsing

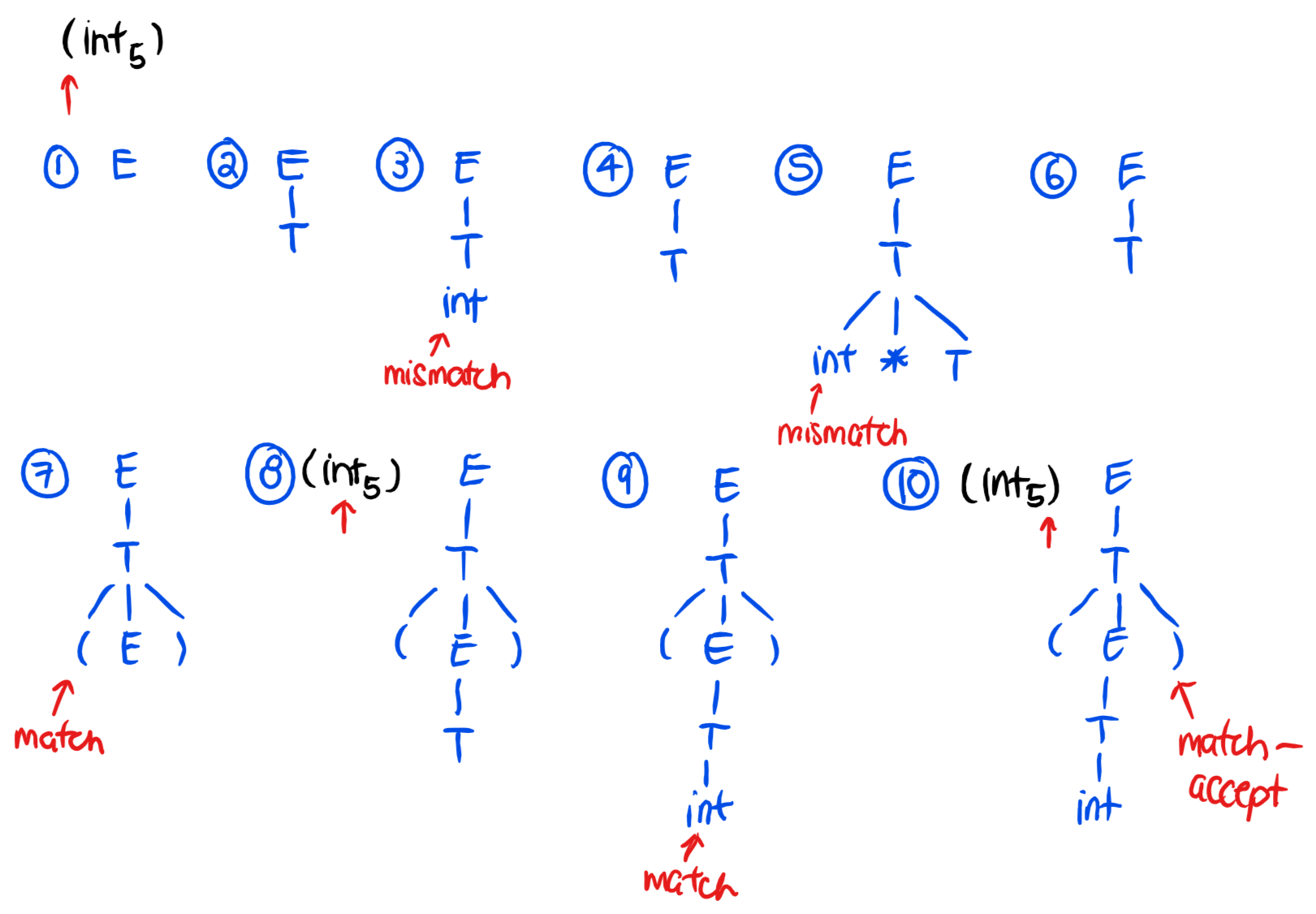
### Bottom-Up vs. Top-Down

* **Top-down parsers** parse from start symbol of grammar to string
* **Bottom-up parsers** parse from the string to the start symbol of the grammar

### Recursive Descent Parsing

Example:

Input: – integer of value 5



### Left Recursion

* A **left-recursive grammar** has for some
* Recursive-descent doesn’t work in this case, because trying to replace would go into an infinite loop



* To eliminate left-recursion, rewrite the grammar using right-recursion



* + Example: is left-recursive



It can be rewritten using right-recursion:



* In general, for a left-recursive grammar , it can be rewritten as:



### \*\*Not on exam \*\* Error Handling

* The compiler has two purposes:
  + Detect non-valid programs
  + Translate valid programs
* There are many types of errors:
  + **Lexical** detected by Lexer
  + **Syntax** detected by Parser
  + **Semantic** detected by Type checker
  + **Correctness** detected by User

#### Syntax Error Handling

* Panic mode
  + Discard tokens until one with a clear role is found
  + Continue from there
  + Synchronizing tokens: tokens that can be discarded
* Error productions
  + Promotes common errors to alternative syntax, by writing them into the grammar
  + This is useful for catching common mistakes, but complicates the grammar
* Local and global correction
  + Try to insert/delete tokens to find a correct “nearby” program
  + Results in exhaustive search for what to insert/delete
    - Hard to implement
    - Slows down parsing
    - The “nearby” program isn’t necessarily the intended one

## 10. Top-Down Parsing: Predictive and LL(k) Parsers

* **Parser**: a program that accepts strings and returns parse trees
* **Predictive** **recursive-descent parser**
  + Follows recursive-descent template – requires backtracking



* + Every LL(k) language has a corresponding predictive recursive-descent parser
* **LL(k) parser**
  + Table-driven



* + Uses lookahead tokens – doesn’t require backtracking
  + Every LL(k) language has a corresponding LL(k) parser



* **LL(k) grammar**: a context-free language that can be parsed by a LL(k) table-driven parser or predictive recursive-descent parser

### LL(k) Table-Driven Parser

* **Predictive** – given a non-terminal to expand and the next input token, predicts which grammar rule to use to parse
* Consists of:
  + Input buffer to hold input string
  + Stack to store terminals and non-terminals that need to be parsed
  + Parsing table that is used to determine which grammar rule to apply, given the symbol at the top of the stack and the next input token

### Left Factoring

* Factor out **common prefixes** of grammars
* Example:

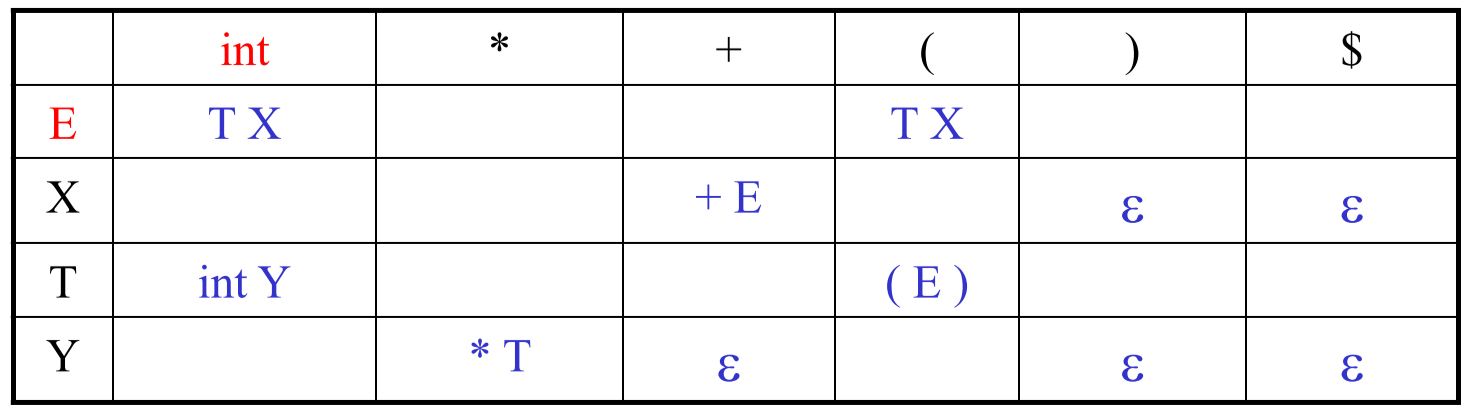


Left-factored:



### LL(1) Parsing Table

* For the leftmost non-terminal and the next input token , choose the rule at
* Reject if an error state is reached (empty table entry)
* Accept on end of input if the stack is empty
* Example:



* + : when the top of the stack is and the next input token is , expand
  + : when the top of the stack is and the next input token is , remove from the stack
* If the stack is and the input is

|  |  |  |
| --- | --- | --- |
| **Table entry used** | **Stack** | **Input buffer** |
|  |  |  |
|  |  |  |
|  |  |  |
| Remove from stack, increment input pointer |  |  |
|  |  |  |
| Remove from stack, increment input pointer |  |  |
|  |  |  |
| Remove from stack, increment input pointer |  |  |
|  |  |  |
|  |  |  |
| Accept |

### Constructing a LL(1) Parse Table

* Consider a production rule , with non-terminal , non-terminal/terminal , and input token
* When can we replace with , i.e. when does ?
  + If ( can derive a in the first position)
    - This is the case where
  + If and ( cannot derive ) and ( follows in at least one derivation)
    - This is the case where

#### Computing First Sets

* For terminals,
* if or if and
* if and

Example:



#### Computing Follow Sets

* For the start symbol ,
* for each production rule
* for each production rule where
  + If then and
  + If then

Example:

#### Constructing the Parse Table

* For each production rule :
  + For each terminal ,
  + If , for each terminal ,
  + If and ,



## 11. Bottom-up Parsing

### Bottom-up Parsers

* Don’t need grammars to be **left-factored**
* Work by reducing a string to the start symbol, by inverting production rules

Grammar:

Example: int \* int + int

* + int \* int + int
  + int \* T + int
  + T + int
  + T + T
  + T + E
  + E

#### Important Fact #1

Bottom-up parsers trace the **rightmost derivation in reverse**

* If is a step in the bottom-up parse and the next reduction is , then  **must be a string of terminals**, because is a step in the rightmost derivation

### Shift-Reduce Parsing

* Split string into two substrings, divided by |



* + Left: terminals and non-terminals
  + Right: string of terminals yet to be examined



* **Shift**: Move | one place to the right – shifts a terminal to the left string
* **Reduce**: Apply the inverse of a production rule at the right end of the left string



* The left string can be implemented as a stack
  + Shift pushes a terminal on the stack
  + Reduce pops 0+ symbols off of the stack and pushes a non-terminal onto it



Grammar:



Example: int \* int + int

* + | int \* int + int
  + int | \* int + int
  + int \* | int + int
  + int \* int | + int
  + int \* T | + int
  + T | + int
  + T + | int
  + T + int |
  + T + T |
  + T + E |
  + E |

#### Conflicts

* **Shift-reduce conflict**: in a given state, it is legal to shift or reduce
* **Reduce-reduce conflict**: in a given state, it is legal to reduce by two different production rules



We only want to reduce when the result can be reduced to the start symbol



* **Handle**: strings that can be reduced, and allow reductions back to the start symbol – we only want to reduce at handles



* + Handles represent the **RHS** of a production rule; if is a production rule, then the is a handle of



#### Important Fact #2

In shift-reduce parsing, handles always appear at the **top of the stack**



* Handles are never to the left of the rightmost non-terminal – thus, the | never has to move to the right
* Bottom-up parsing algorithms are based on recognizing handles
  + There are no known efficient algorithms for doing so

## 12. Semantic Analysis

* Parsers alone cannot catch many errors
* Semantic analysis checks:
  + Identifiers are declared
  + Types
  + Inheritance
  + Classes
  + Methods
  + Reserved identifiers aren’t misused

### Scopes

* **Scope**: the portion of the program in which an identifier is accessible
* The same identifier can refer to different things throughout the program, as long as the different scopes don’t overlap
* Most languages have **static scope** – depends on program text, not run-time behaviour



* Some languages have **dynamic scope** – depends on program execution
  + Dynamically-scoped variables refer to the closest enclosing binding during program execution



#### Symbol Tables

* Used to track the current bindings of identifiers
* E.g. let x: Int = 0 in e
  + Before processing e, add definition of x to symbol table, overriding any previous definition of x
  + After processing e, remove current definition of x and restore its old definition
* Simplest implementation – stack
  + Doesn’t work well for multiple variable definitions in the same scope
* More complex implementation
  + Methods to start a new nested scope and exit the current scope

### Types

* **Type**:
  + Set of values
  + Set of operations on those values
* Certain operations are legal for each type
* **Type checking**: process of verifying fully typed programs – ensures that operations are used with the correct types
* For the different types of languages:



* + Statically typed: almost all type checking done as part of **compilation**



* + Dynamically typed: almost all type checking done during **program execution**
  + Untyped: no type checking



#### Type Inference and Inference Rules

* **Type inference**: process of filling in missing type information



* Inference rules have the form “if hypothesis is true, then conclusion is true”



* **Compact form**:



* + : and
  + : if-then
  + : e has type T

E.g. “if e1 has type Int and e2 has type Int, then e1+e2 has type Int”

Compact form:

* **Traditional form**:
  + : it is provable that

E.g. “if e1 has type Int and e2 has type Int, then e1+e2 has type Int”

Traditional form:

* Rules for constants:

#### Soundness



* A type system is **sound** if whenever , evaluates to type



* Type checking proves the facts of , through a bottom-up pass of the AST (Abstract Syntax Tree)

## 13. Code Generation

* Two main goals:
  + Correctness
  + Speed
* Makes some assumptions about execution:
  + Execution is sequential
  + When a procedure is called, control eventually returns to the point right after the call

### Activations



* **Activation**: the invocation of a procedure *P* is called the activation of *P*
* **Lifetime** **of an activation** of *P*: all the steps required to execute *P*
  + Includes all the steps of any procedures that *P* activates



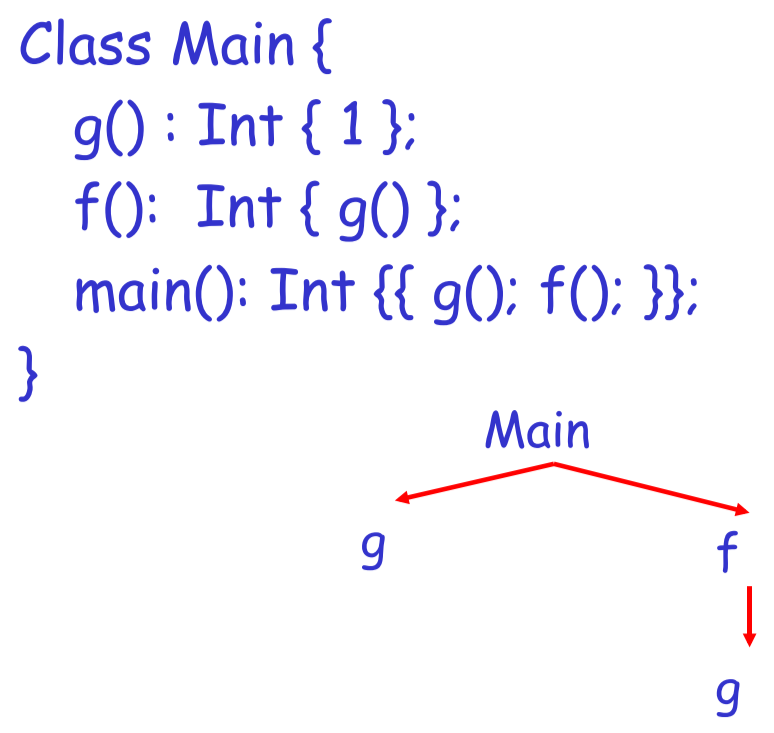
#### Variable Lifetimes



* **Lifetime of a variable**: the portion of the program where the variable is defined
  + Lifetime is a dynamic/run-time concept
  + Scope is a static/compile-time concept

#### Activation Trees

* Activation lifetimes can be depicted as a tree
* Assume that when a procedure *P* calls a procedure *Q*, *Q* finishes and returns before *P* does
  + This ensures that the lifetimes of activations are properly nested
  + Since they are properly nested, can use a stack to trace active procedures
* Activation tree depends on run-time behaviour – can be different depending on program input
* Example:





### Activation Records



* **Activation record (AR)/Frame**: the information needed to manage one procedure’s activation
* If procedure *F* calls procedure *G*, then *G*’s activation record contains a mixture of *F* and *G*’s information
  + *G*’s AR needs to contain information about how to resume execution of *F*
* A typical AR for G will contain:



* + Space for G’s return value (needed by F)



* + Parameters to *G* (supplied by *F*)



* + **Control link**: pointer to the AR of the caller of *G* (in this case, *F*)



* + **Return address**: where execution should resume after the procedure finishes



* + Machine status prior to calling *G*: contents of registers and program counter, and any local variables



* + Other temporary values



* Example:





Stack after two calls to *f*:

|  |  |
| --- | --- |
| main |  |
| f | <return value> |
| 3 |
| control link |
| (\*) |
| f | <return value> |
| 2 |
| control link |
| (\*\*) |

Stack after second call to *f* returns:

|  |  |
| --- | --- |
| main |  |
| f | <return value> |
| 3 |
| control link |
| (\*) |
| f | 1 |
| 2 |
| control link |
| (\*\*) |

* + The advantage to putting the return value first in the frame is that the caller can find it at a fixed offset from its own frame
* The compiler needs to determine the layout of ARs at compile-time, to generate code that correctly accesses the locations in the AR
* Thus, the AR layout and code generator must be designed together

#### Global Variables

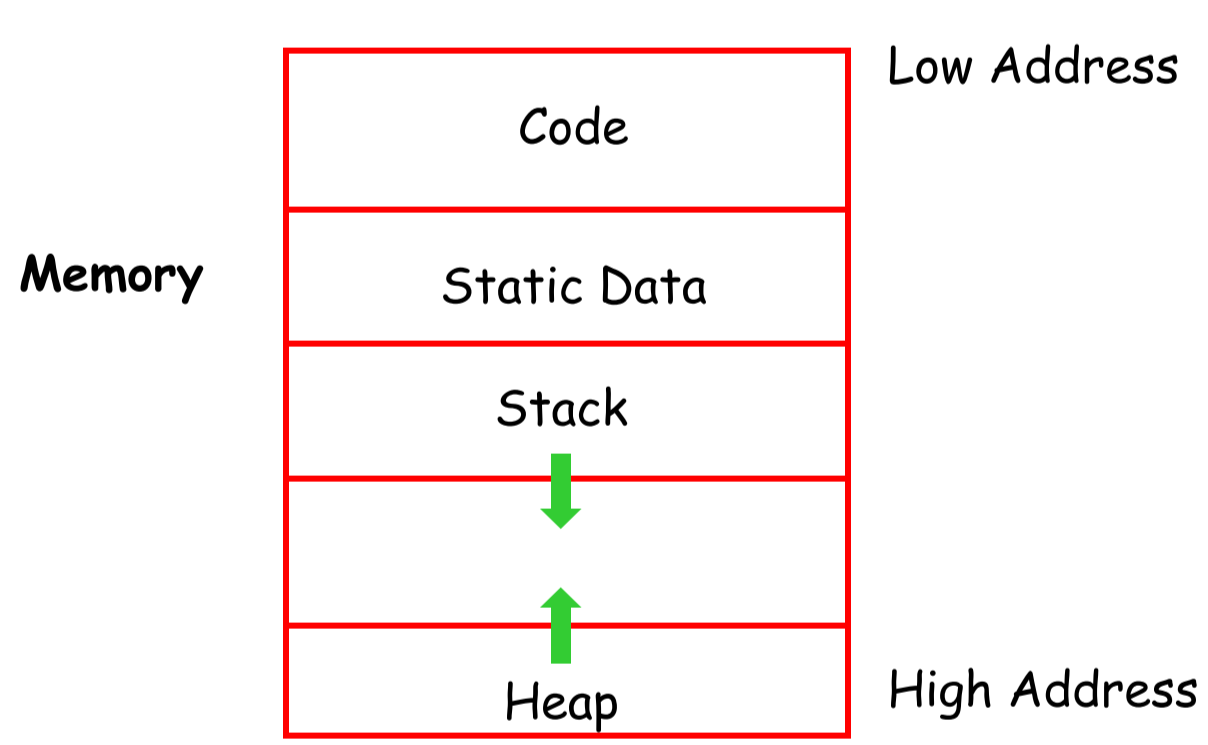
* All references to global variables refer to the same object, which means they can’t be stored in the AR
* Instead global variables are statically allocated to a fixed address once



#### Heap Storage

* Values that outlive the procedures that create them also can’t be stored in the AR
* Heaps are used to store dynamically allocated data

#### General Memory Layout



* **Code area**: object code
  + Usually fixed-size and read-only
* **Static area**: data with fixed addresses, e.g. global data
  + Usually fixed-size and readable/writable
* **Stack**: activation records for every currently active procedure
  + Each AR is usually a fixed size
* **Heap**: all other data
* To prevent the stack and heap from growing into each other, start them at opposite ends of memory and let them grow towards each other

#### Alignment

* Most modern machines are 32-bit: 4 bytes/32 bits per word
* Data is **word-aligned** if it begins at a word boundary
* Example: String “Hello” – to align it, add 3 padding characters to the string

### Stack Machines

* No variables or registers – stack keeps track of intermediate results



* Each instruction:
  + Remove operands from top of stack
  + Computes required operation
  + Pushes result on stack
* Each operation takes operands from the same place and puts the result in the same place
  + This means that the location of operands and results are implicit
* **Accumulator**: register that stores the value at the top of the stack
  + Registers allow for faster memory access



* + The result of a calculation is stored in the accumulator
  + For nested calculations, push the result onto the stack
* To evaluate an operation:
  + Push all operands except the last onto the stack
    - Store value in accumulator
    - Push accumulator to top of stack
  + Do the operation, using the value in the accumulator and the value at the top of stack – store result in accumulator
  + Pop top of stack
  + Push result to stack if it is an intermediate value

## 14. Code Generation

### From Stack Machines to MIPS

* The compiler generates code for a stack machine with accumulator
* Stack machine instructions can be simulated with MIPS instructions and registers
* MIPS:
  + Accumulator is stored in register $a0
  + Stack is stored in memory
    - MIPS convention: stack grows towards lower memory addresses
  + Stack pointer $sp points to the next location on the stack
    - Top of stack is at address $sp+4
  + $T1 is a temporary register

#### MIPS Instructions

* li <reg> <val>: load immediate value into register
* lw <reg> <addr>: load word from memory address into register
* sw <reg> <addr>: store value of register to memory address
* addiu <addr> <addr> <val>: adds intermediate value to address
* add <reg3> <reg1> <reg2>: adds reg1 and reg2, stores in reg3
* sub <reg3> <reg1> <reg2>: subtracts reg2 from reg1, stores in reg3
* b <label>: branch (jump) to label
* b <reg1> <reg2> <label>: branch to label if reg1 = reg2

|  |  |
| --- | --- |
| * + ***Stack Machine Code*** | * + ***MIPS Code*** |
| * + acc <- <val> | * + li $a0 <val> |
| * + push acc | * + sw $a0 0($sp)   + addiu $sp $sp -4 |
| * + pop | * + addiu $sp $sp 4 |
| * + acc <- acc + top\_of\_stack | * + lw $T1 4($sp)   + add $a0 $a0 $T1 |
| * + acc <- top\_of\_stack - acc | * + lw $T1 4($sp)   + sub $a0 $T1 $a0 |

#### Activation Record

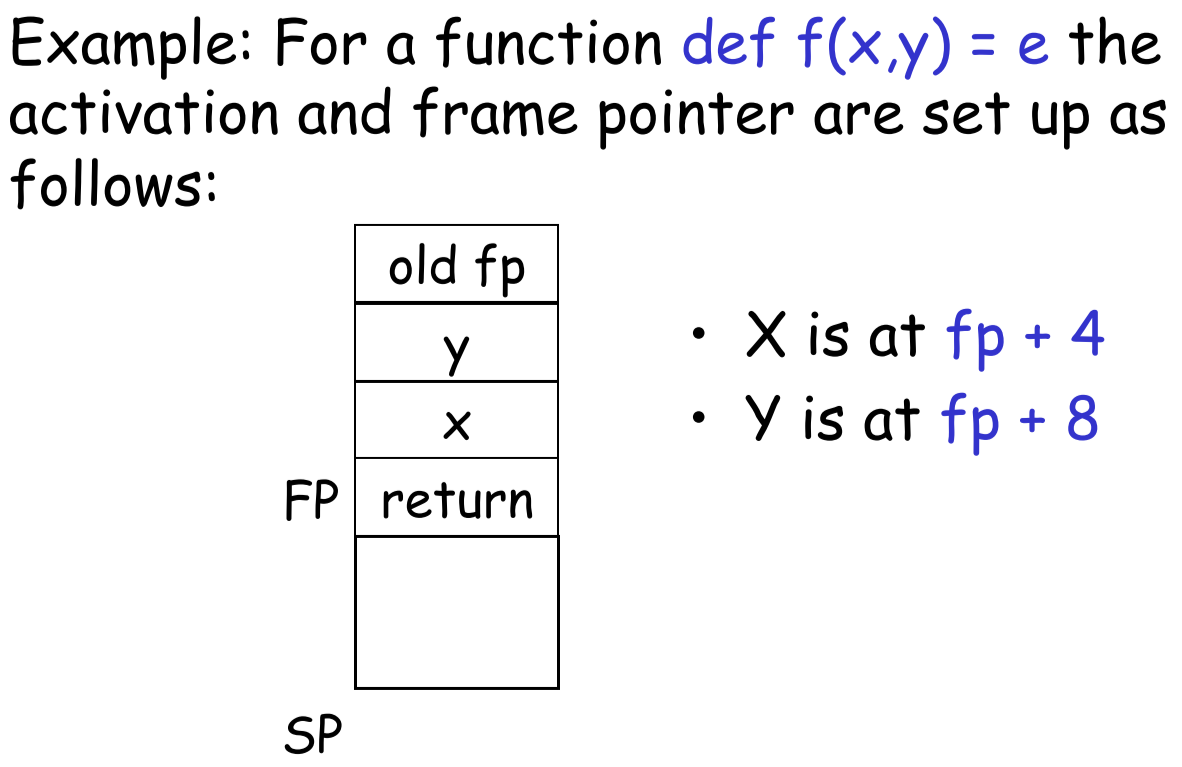
* The result is always stored in the accumulator, so it doesn’t need to be stored in the AR
* Very simple AR suffices, which holds:
  + Parameters to the function
  + Return address



* + **Frame pointer** ($fp): pointer to the current activation
* The stack guarantees that on function exit, the stack pointer $sp is in the same place it was on function entry



* Because the stack grows when intermediate results are saved, function parameters aren’t at a fixed offset from the stack pointer
* Solution: frame pointer
  + Always points to the return address on the stack – since it doesn’t move, it can be used to find the addresses of variables



## Code Generation for OO Languages

### Object-Oriented Languages

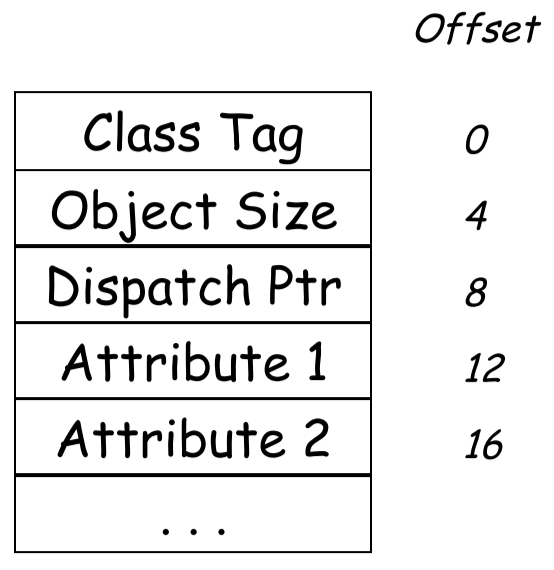
* If B is a subclass of A, then an object of class B can be used wherever an object of class A is expected
* This means that any code for class A works unmodified on an object of class B

### Object Layout

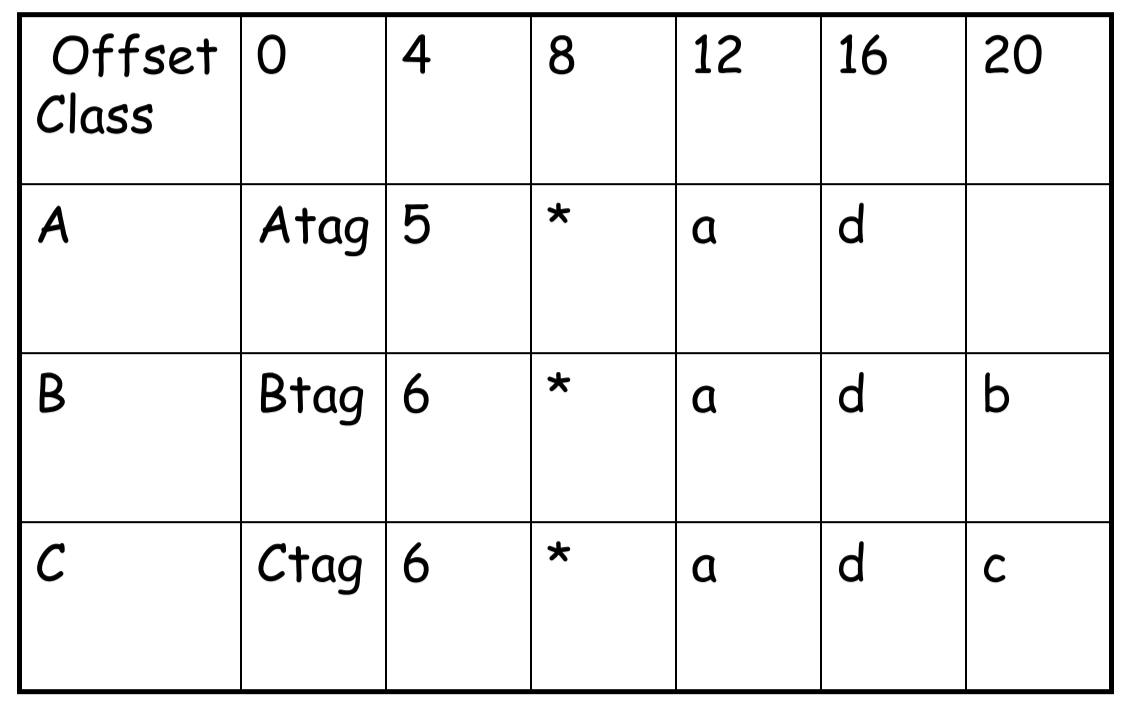


* Attributes a and d are inherited by classes B and C
* For A’s methods to work properly for A, B, and C objects, a must be in the same “place” in each object
  + Objects are typically laid out in contiguous memory
  + Each attribute is stored at a fixed offset in an object
  + All references to a method are an index into the object at an offset corresponding to the method

**Typical Layout**



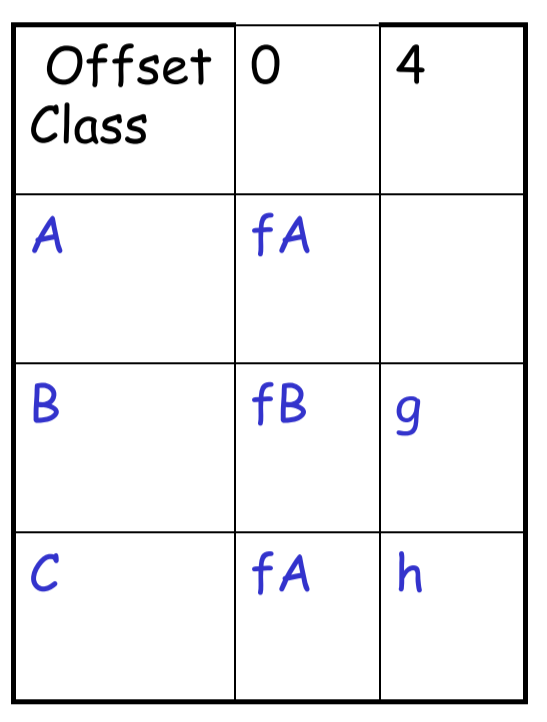
* **Class tag**: identifies class of object
* **Object size**: size of object in words
* **Dispatch pointer**: pointer to dispatch table of methods
* **Attributes**
* For subclasses, the layout of main class (A)’s attributes remain unchanged



### Dynamic Dispatch

* References to different attributes/methods depend on the class:
  + e.g(): refers to method g() in B if e is a B
  + e.f(): refers to method f() in A if e is an A or C, or f() in B if e is a B

#### Dispatch Tables

* **Dispatch table**: index of a class’s set of methods
  + Array of **method entry points**
  + Each method lives at a fixed offset in the dispatch table for a class and any of its subclasses
* Since methods can be overridden, the method f() isn’t the same for each class, but it is always at the same offset  
  
* The dispatch pointer for class X points to the start of the dispatch table for class X
* Each method is assigned an offset in the dispatch table at compile-time

## 15. Local Optimizations

### Intermediate Languages

* High-level assembly language
* Uses an **unlimited number** **of** **register names**
* Uses **control structures** like an assembly language
* Uses **opcodes**, but some are higher-level

#### Three-Address Intermediate Code

* Each instruction is in the form:

x := y op z

x := op y

* + y and z are either registers or constants
* **Example**: x + y \* z

t1 := y \* z

t2 := x + t1

* Generating intermediate code is very similar to generating assembly code, just with an unlimited number of registers to hold intermediate results
* **igen(e,t)** generates the code to compute the value of e and store it in register t
* **Example**: igen(e1+e2, t)

igen(e1, t1)

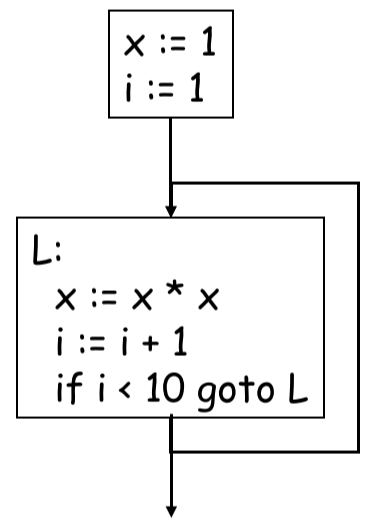
igen(e2, t2)

t := t1 + t2

### Basic Blocks

* Maximal sequence of instructions with **no labels** (except first instruction) and **no jumps** (except last instruction)
* Single-entry, single-exit, straight-line code segment – can only jump into/enter at beginning and jump out of/exit at end
* Used to find local optimizations

### Control-Flow Graph



* Directed graph
  + Nodes: basic blocks
  + Edges: if execution can pass from the last instruction in block A to the first instruction in block B, then there is an edge from A to B
* Used to find global optimizations

### Local Optimizations

**Algebraic Simplification**

* Some statements can be **deleted**

x := x + 0 delete

x := x \* 1 delete

* Some statements can be **simplified**

x := x \* 0 x := 0

y := y \*\* 2 y := y \* y

x := x \* 8 x := x << 3

x := x \* 15 t := x << 4; x = t – x;

**Constant Folding**

* **Operations on constants** can be computed at compile-time

x := 2 \* 2 x := 4

**Flow of Control Optimizations**

* Eliminate **unreachable basic blocks** – code that is unreachable from the initial block
* E.g. if an if condition always evaluates to false

if (2 < 0) then x := 5

x := 3

else

x := 5

**Common Subexpression Elimination**



* When two assignments have the **same RHS**, then they compute the same value

x := y + z x := y + z

w := y + z w := x

**Copy Propagation**

* If a variable assignment w := x exists in the block, replace subsequent uses of w with x
* Useful for enabling constant folding and dead code elimination

b := y + z b := y + z

a := b ~~a := b~~

x := 2 \* a x := 2 \* b

**Dead Code Elimination**

* A variable is **dead** if it does not contribute to the program’s result
* If a variable assignment exists but isn’t used anywhere else in the program, it can be eliminated

x := y + z b := y + z b := y + z

a := x a := b ~~a := b~~

x := 2 \* a x := 2 \* a x := 2 \* b

**Example**

a := x \*\* 2

b := 3

c := x

d := c \* c

e := b \* 2

f := a + d

g := e \* f

|  |  |  |  |
| --- | --- | --- | --- |
| **Algebraic simplification** | **Copy propagation & Dead code elimination** | **Constant folding** | **Copy propagation & Dead code elimination** |
| a := x \* x  b := 3  c := x  d := c \* c  e := b \* 2  f := a + d  g := e \* f | a := x \* x  ~~b := 3~~  ~~c := x~~  ~~d := x \* x~~  e := 3 \* 2  f := a + a  g := e \* f | a := x \* x  e := 6  f := a + a  g := e \* f | a := x \* x  ~~e := 6~~  f := a + a  g := 6 \* f |

Final code:



d := x \* x



f := a + a

g := 6 \* f

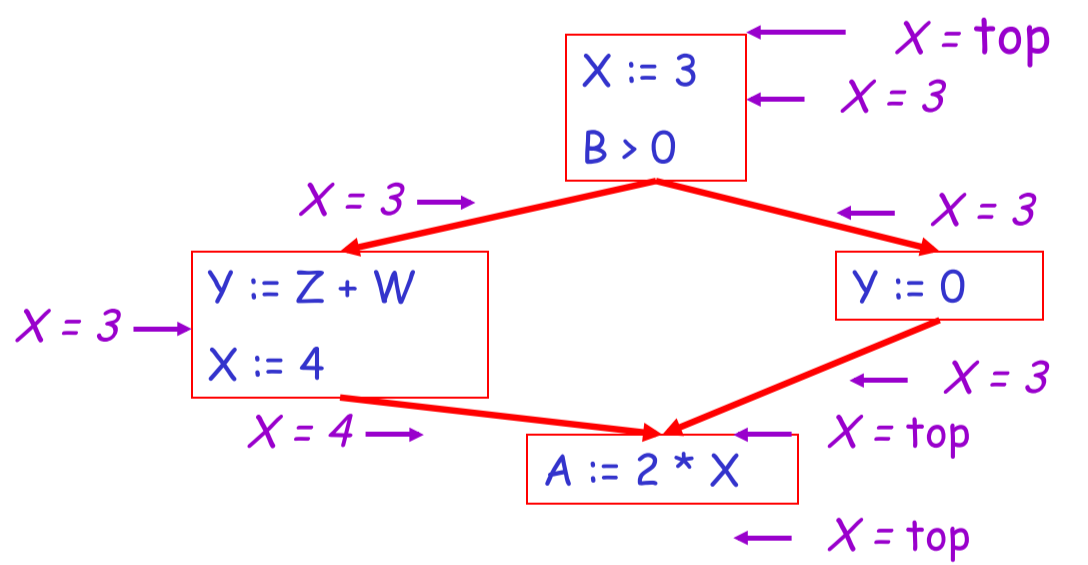
#### Peephole Optimizations on Assembly Code

* **Peephole optimizations**: optimizer replaces a short sequence of instructions with equivalent but faster ones
* Many basic block optimizations can be cast as peephole optimizations

## 16. Global Optimizations

### Global Constant Propagation

* Local optimizations can be applied to an entire control-flow graph
* **Invariant 1**: to replace a use of a variable x with a constant k, must ensure that on every path to the use of x, the last assignment of x is x := k
* In order to compute this, we associate x with a value at every program point:
  + **z**: control hasn’t reached that point – unknown
  + **c**: x is the constant c
  + **top**: x is definitely not a constant



* For each statement s, compute the value of x right before s
  + **Rule 1**: if any edge going into s has x = top, then C(x, s, in) = c
  + **Rule 2**: if two or more edges assign different constants to x, then C(x, s, in) = top
  + **Rule 3**: if all edges have x = c (same constant) or x = z, then C(x, s, in) = c
  + **Rule 4**: if all edges have x = z, then C(x, s, in) = z
* For each statement s, compute the value of x right after s
  + **Rule 5**: C(x, s, out) = z if C(x, s, in) = z (and s doesn’t assign x)
  + **Rule 6**: if s assigns x = c, then C(x, s, out) = c
  + **Rule 7**: if s assigns x = any non-constant expression, then C(x, s, out) = top
  + **Rule 8**: C(x, s, out) = c if C(x, s, in) = c (and s doesn’t assign x)
* Algorithm:
  1. For every entry into the program, set C(x, s, in) = top
  2. Set C(x, s, in) = C(x, s, out) = z everywhere else
  3. Make substitutions for x until all points satisfy rules 1-8 – pick statements s that don’t satisfy the rules and update constants of x at those points

### Liveness Analysis (Global Dead Code Elimination)

* Once constants have been globally propagated, we want to eliminate dead code
* A variable x is **live** at statement s if there exists a statement s’ that uses x and:
  + s precedes s’
  + There are no intervening assignments to x
* A variable assignment is **dead code** if x is dead after the assignment – value assigned doesn’t get used
* Liveness analysis is done bottom-up, according to the following rules:
  + **Rule 1**: if x is live from any edge going out of s, L(x, s, in) = true
  + **Rule 2**: if s refers to x on the RHS of an expression, L(x, s, in) = true
  + **Rule 3**: if s assigns a value to x, L(x, s, in) = false
  + **Rule 4**: L(x, s, in) = L(x, s, out) if s does not refer to x
* Algorithm:
  1. Let all L() = false initially
  2. Make substitutions for x until all points satisfy rules 1-4 – pick statements s that don’t satisfy the rules and update liveliness of x at those points
     + A value can change from false to true, but never the other way around
     + Each value can change only once
     + Once computed, dead code can be easily eliminated

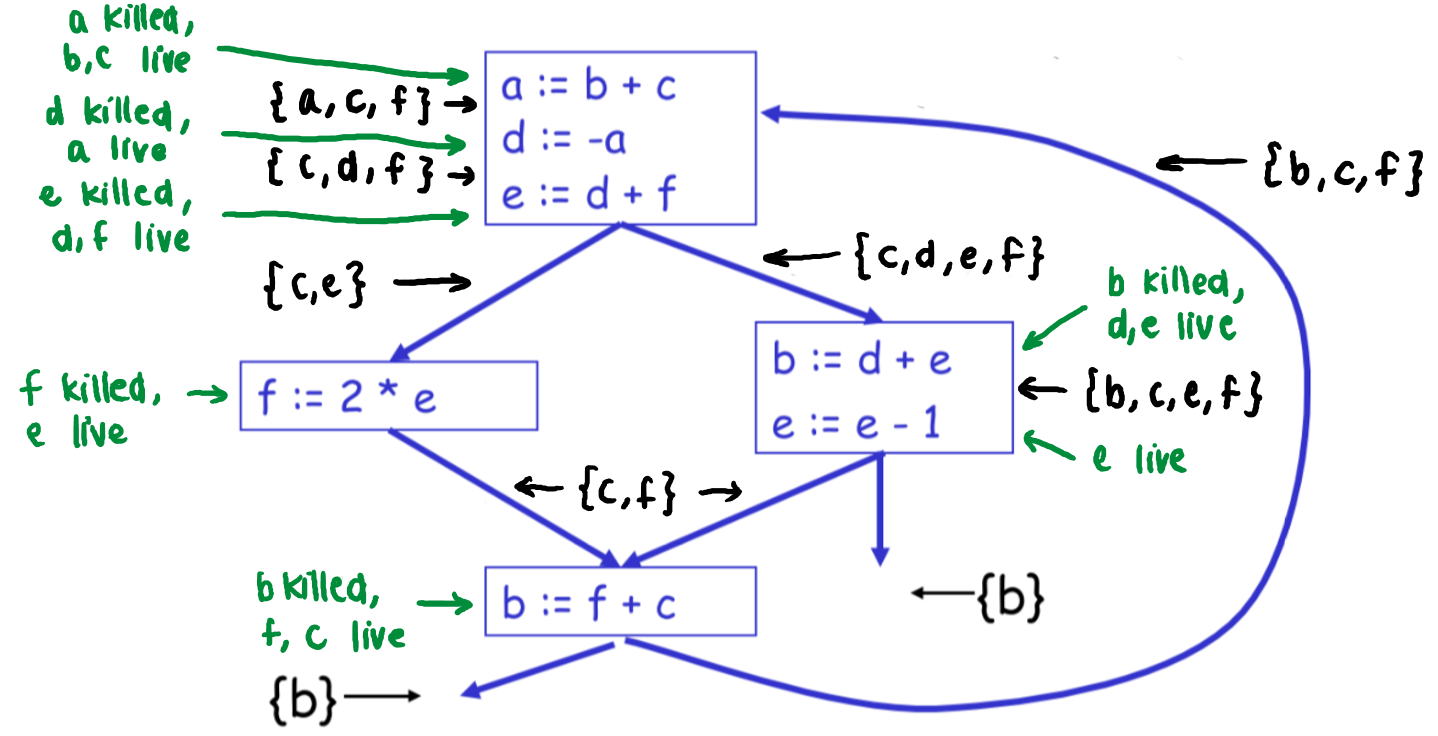
## 17. Register Allocation

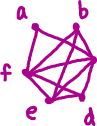
### Register Allocation Problem

* Intermediate code uses unlimited number of registers
* Typically, this means that it uses too many temporaries
* **Goal**: rewrite intermediate code to use no more temporaries than there are machine registers
* **Idea**: two temporaries can share the same register if, at any point in the program, at most one of them is live
  + This lets us assign multiple temporaries to each register without changing program behaviour

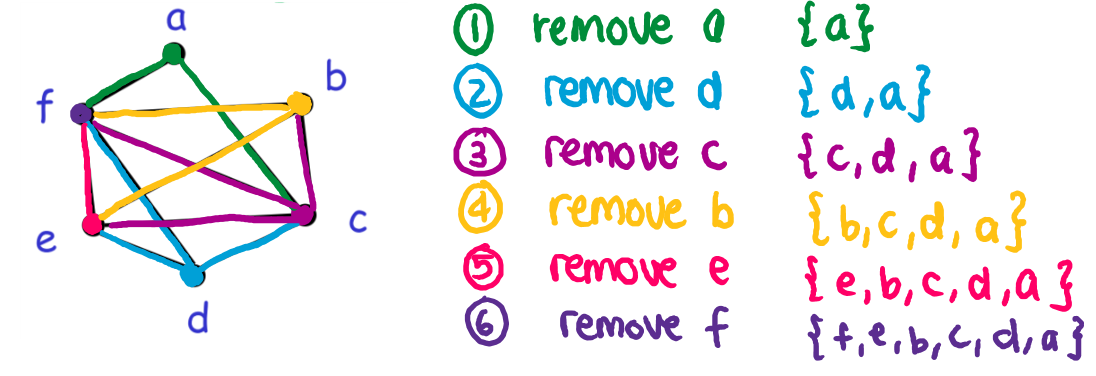
### Register Allocation Algorithm

1. Compute live variables at each point (liveness analysis)





1. Construct a **register interference graph (RIG)**
   * Each temporary is a node
   * Edge for each pair of temporaries that are live at the same point in the program
   * Two temporaries can be allocated to the same register if there isn’t an edge between them
2. Compute k-colouring of graph
   * Pick a node *n* with fewer than k neighbours in the RIG
   * Put *n* on the stack, remove *n* and its edges from the RIG
   * Repeat until one node remaining
   * Example: 4-colouring



1. Assign colours to nodes on the stack
   * At each step, pick a colour that is different from those assigned to already-coloured neighbours



#### Spilling



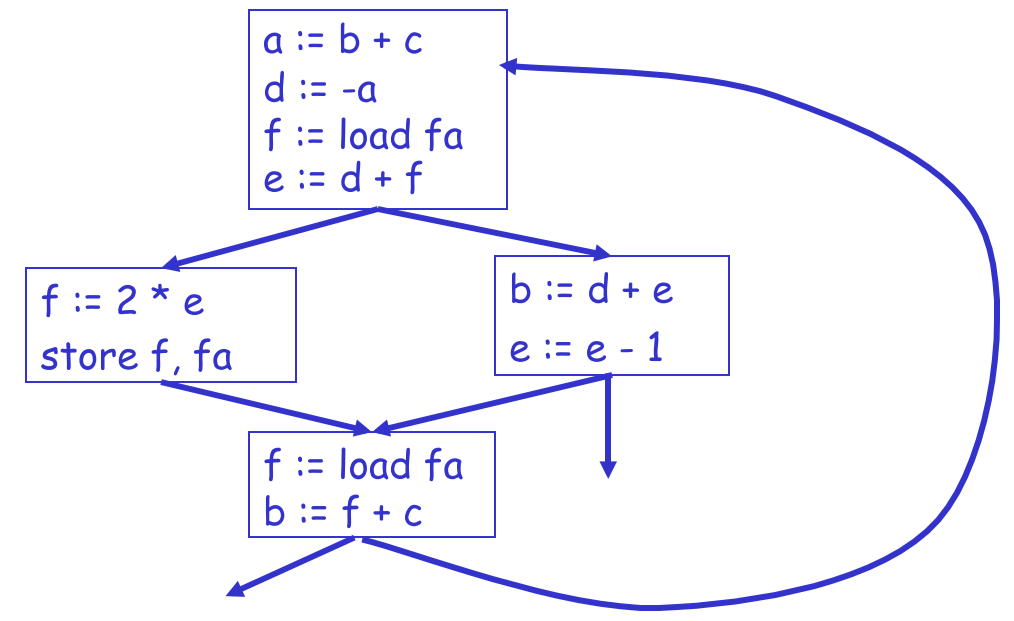
* What happens when k-colouring fails?



* + Pick a node *f* for **spilling** – store temporary in memory

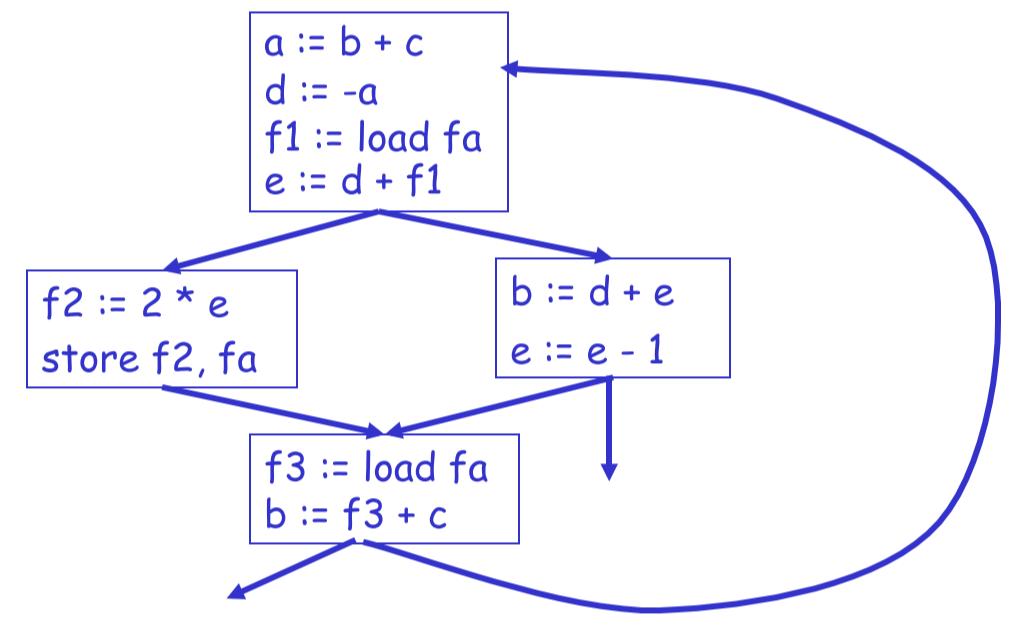


* + Remove *f* and continue the simplification
* **Optimistic colouring**: hope that we may be able to colour *f* after assigning colours to all other nodes
* If optimistic colouring fails, then we must spill *f*:
  + Allocate a memory location for it, e.g. address fa
  + Before each operation that reads *f*, insert f := load fa
  + After each operation that writes *f*, insert store f, fa
* New code:

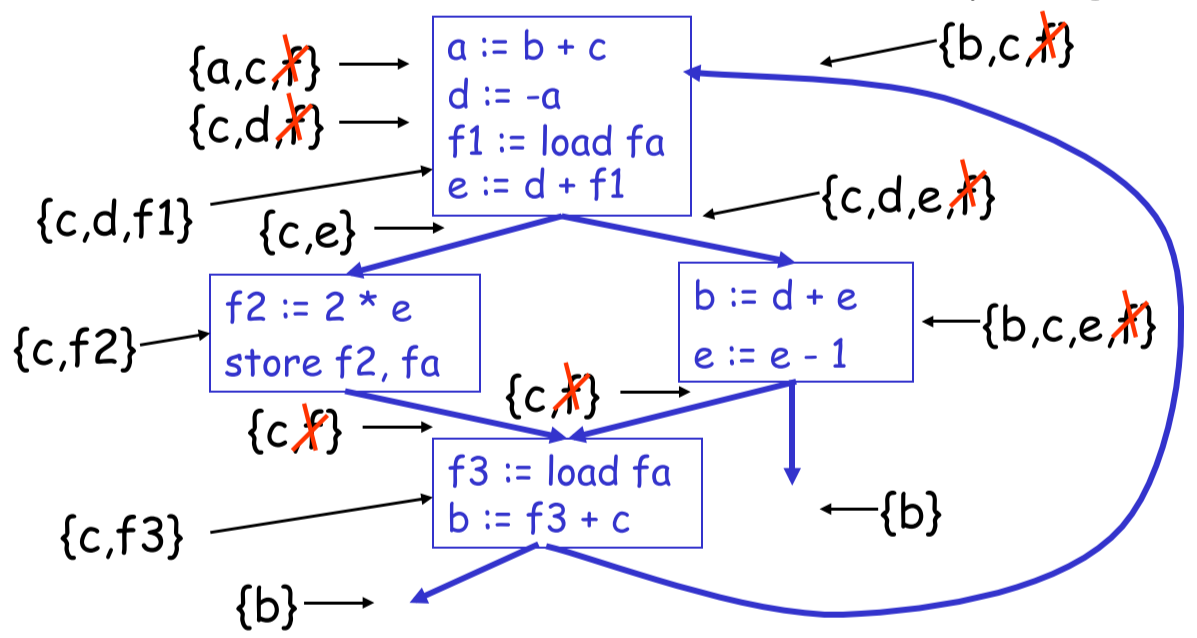




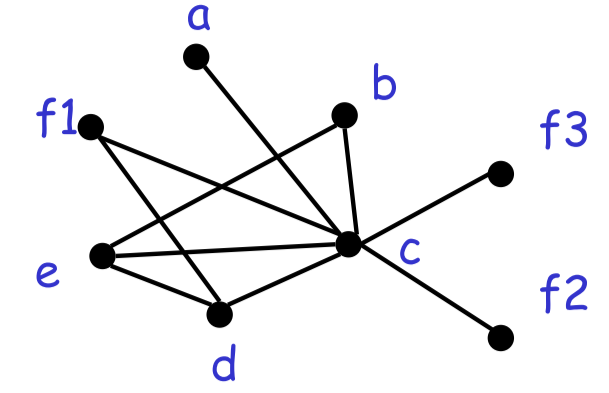
* Should use **distinct register names** whenever possible:



* Re-compute liveness analysis:



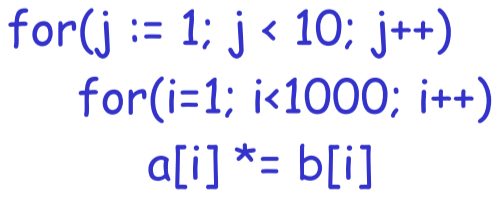
* + *f* is only live:
    - Between f := load fa and the next instruction
    - Between store f, fa and the preceding instruction
  + Spilling reduces the live range of *f*, which means fewer RIG neighbours
* Re-compute RIG after spilling – graph is now 3-colourable:



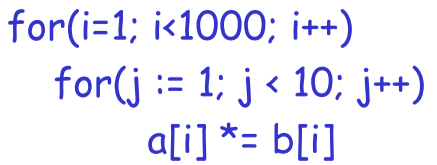
* Which node should be spilled?
  + Temporaries with **most conflicts**
  + Temporaries with **few definitions/uses**
* Multiple nodes may need to be spilled before a colouring is found

### Caches

* Compilers are good at managing registers but not caches – up to programmers
* They can, however, perform some cache optimizations
* **Loop interchange**:



* + Terrible cache performance, because inner loop is accessing 1000 locations, and LRU cache will forget those locations



* Performs same operation, but much better cache behaviour – up to 10x faster