Overview of Compilation

Readings: EAC2 Chapter 1

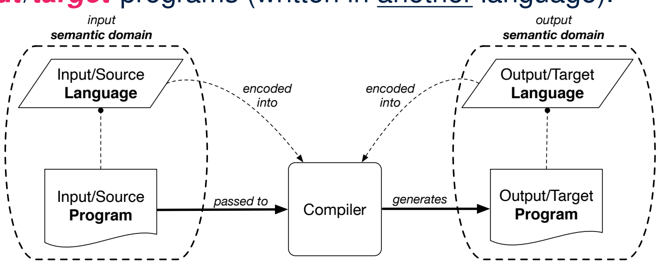
YORK UNIVERSITÉ UNIVERSITY

EECS4302 A:
Compilers and Interpreters
Fall 2022

CHEN-WEI WANG

What is a Compiler? (1)

A **software system** that **automatically** ***translates/transforms*** ***input/source*** programs (written in one language) to ***output/target*** programs (written in another language).



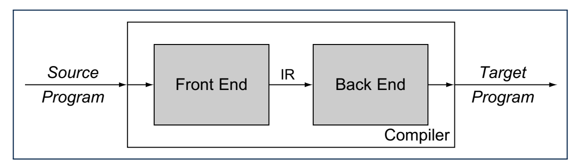
- *Semantic Domain*: Context with its own vocabulary & meanings
  e.g., OO (EECS1022/2030/2011), database (3421), predicates (1090)
- ***Source*** and ***target*** may be in **different** ***semantic domains***.
  e.g., Java programs to SQL relational database schemas/queries
  e.g., C procedural programs to MISP assembly instructions

What is a Compiler? (2)

- The idea about a ***compiler*** is extremely powerful:
  You can turn anything to anything else,
  as long as the following are ***clear*** about these two things:
  - SYNTAX [***specifiable*** as CFGs]
  - SEMANTICS [***programmable*** as mapping functions]

  Mental Exercise. Let's consider an A+ challenge.
- A compiler should be constructed with good *SE principles*.
  - ***Modularity*** [interacting components]
  - ***Information Hiding*** [hiding unstable, revealing stable]
  - ***Single Choice Principle*** [a change only causing minimum impact]
  - ***Design Patterns*** [polymorphism & dynamic binding]
  - ***Regression Testing*** [e.g., unit-level, acceptance-level]

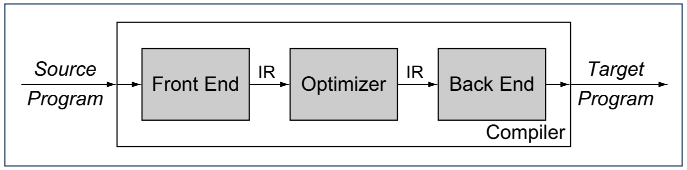
Compiler: Typical Infrastructure (1)



- FRON END:
  - Encodes: knowledge of the **source** language
  - Transforms: from the **source** to some ***IR*** (*intermediate representation*)
  - Principle: ***meaning*** of the source must be *preserved* in the ***IR***.
- BACK END:
  - Encodes knowledge of the **target** language
  - Transforms: from the ***IR*** to the **target**
  - Principle: ***meaning*** of the ***IR*** must be *reflected* in the **target**.

Q. How many ***IRs*** needed for building a number of compilers:
JAVA-TO-C, C#-TO-C, JAVA-TO-PYTHON, C#-TO-PYTHON?
A. Two ***IRs*** suffice: One for ***OO***; one for ***procedural***.
$\Rightarrow$ IR should be as *language-independent* as possible.

Compiler: Typical Infrastructure (2)



OPTIMIZER:

- An ***IR**-to-**IR*** transformer that aims at "improving" the **output** of front end, before passing it as **input** of the back end.
- Think of this transformer as attempting to discover an "***optimal***" solution to some computational problem.
  e.g., runtime performance, static design

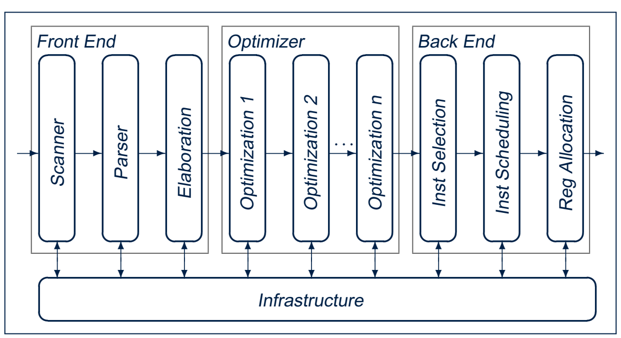
Q. Behaviour of the **target** program depends upon?

1. ***Meaning*** of the **source** preserved in ***IR***?
2. ***IR**-to-**IR*** transformation of the optimizer ***semantics-preserving***?
3. ***Meaning*** of ***IR*** preserved in the generated **target**?

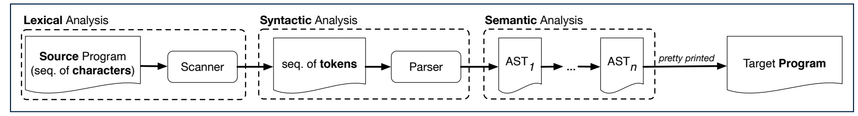
(1) – (3) necessary & sufficient for the *soundness* of a compiler.

Example Compiler 1

- Consider a <u>conventional</u> compiler which turns a ***C-like program*** into executable ***machine instructions***.
- The ***source*** and ***target*** are at different levels of *abstractions*:
  - C-like program is like "high-level" ***specification***.
  - Macine instructions are the low-level, efficient ***implementation***.



Compiler Infrastructure: Scanner vs. Parser vs. Optimizer



- The same input program may be <u>perceived</u> differently:
  1. As a ***character sequence*** [subject to ***lexical*** analysis]
  2. As a ***token sequence*** [subject to ***syntactic*** analysis]
  3. As a *abstract syntax tree (AST)* [subject to *semantic* analysis]
- ***(1)*** & ***(2)*** are routine tasks of lexical/grammar rule specification.
- *(3)* is where the most creativity is used to a compiler:
  A series of ***semantics-preserving AST***-to-***AST*** transformations.

Compiler Infrastructure: Scanner

- The source program is perceived as a sequence of ***characters***.
- A scanner performs *lexical analysis* on the input character sequence and produces a sequence of ***tokens***.
- ANALOGY: Tokens are like individual ***words*** in an essay.
  ⇒ Invalid tokens ≈ Misspelt words
  e.g., a token for a <u>useless</u> delimiter: e.g., space, tab, new line
  e.g., a token for a <u>useful</u> delimiter: e.g., `(`, `)`, `{`, `}`, `,`
  e.g., a token for an identifier (for e.g., a variable, a function)
  e.g., a token for a keyword (e.g,. `int`, `char`, `if`, `for`, `while`)
  e.g., a token for a number (for e.g., `1.23`, `2.46`)
  Q. How to specify such ***pattern of characters***?
  A. ***Regular Expressions*** (***REs***)
  e.g., RE for keyword `while` [`while`]
  e.g., RE for an identifier [`[a-zA-Z][a-zA-Z0-9_]*`]
  e.g., RE for a white space [`[ \t\r]+`]

Compiler Infrastructure: Parser

- A parser's input is a sequence of ***tokens*** (by some scanner).
- A parser performs *syntactic analysis* on the input token sequence and produces an ***abstract syntax tree (AST)***.
- ANALOGY: ASTs are like individual ***sentences*** in an essay.

  ⇒ Tokens not ***parseable*** into a valid AST ≈ Grammatical errors

  Q. An essay with no speling and grammatical errors good enough?
  A. No, it may talk about non-sense (sentences in wrong contexts).

  ⇒ An input program with no lexical/syntactic errors should still be subject to *semantic analysis* (e.g., type checking, code optimization).

Q.: How to specify such ***pattern of tokens***?
A.: ***Context-Free Grammars*** (***CFGs***)
e.g., CFG (with **terminals** and **non-terminals**) for a while-loop:

WhileLoop	::=	WHILE LPAREN *BoolExpr* RPAREN LCBRAC *Impl* RCBRAC
Impl	::=	
	|	*Instruction* SEMICOL *Impl*

Compiler Infrastructure: Optimizer (1)

- Consider an input *AST* which has the pretty printing:

```
b := ... ; c := ... ; a := ...
across i |..| n is i
  loop
    read d
    a := a * 2 * b * c * d
  end
```

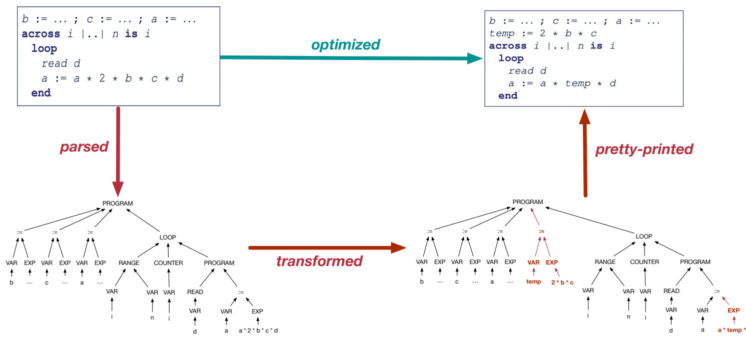
  Q. AST of above program ***optimized*** for performance?
  A. No ∵ values of 2, b, c stay invariant within the loop.

- An ***optimizer*** may *transform* AST like above into:

```
b := ... ; c := ... ; a := ...
temp := 2 * b * c
across i |..| n is i
  loop
    read d
    a := a * temp * d
  end
```


Compiler Infrastructure: Optimizer (2)

Problem: Given a user-written program, ***optimize*** it for best runtime performance.



Example Compiler 2

- Consider a compiler which turns an object-based ***Domain-Specific Language (DSL)*** into a ***SQL database***.
- Why is an ***object-to-relational compiler*** valuable?

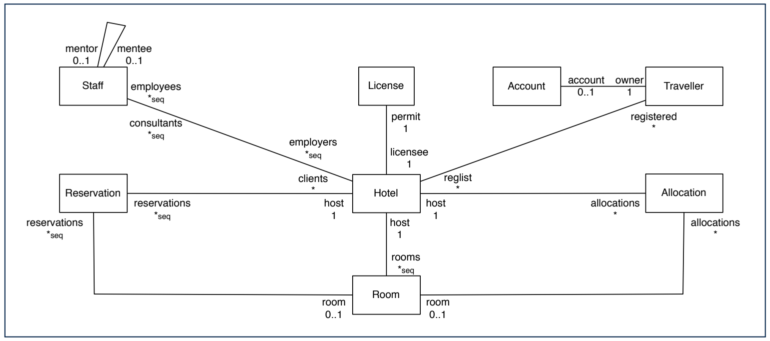
  Hint. Which semantic domain is better for high-level specification?
  Hint. Which semantic domain is better for data management?

	managing big data	specifying data & updates
object-oriented environment	×	✓
relational database	✓	×

- *Challenge*: ***Object-Relational Impedance Mismatch***

Example Compiler 2

- The input/source contains 2 parts:
  - **Data Model**: ***classes*** & ***associations***
    e.g., data model of a Hotel Reservation System:



  - **Behavioural Model**: update methods specified as ***predicates***

Example Compiler 2: Transforming Data

```
class A {
  attributes
    s: string
    bs: set(B . a) [*] }
```

```
class B {
  attributes
    is: set (int)
    a: A . bs }
```

- Each class is turned into a ***class table***:
  - Column `oid` stores the object reference. [Primary Key]
  - Implementation strategy for attributes:

	Single-Valued	Multi-Valued
Primitive-Typed	column in ***class table***	***collection table***
Reference-Typed	***association table***	

- Each ***collection table***:
  - Column `oid` stores the context object.
  - 1 column stores the corresponding primitive value or `oid`.
- Each ***association table***:
  - Column `oid` stores the association reference.
  - 2 columns store `oid`'s of both association ends. [Foreign Key]

Example Compiler 2: Input/Source

- Consider a **valid** input/source program:

```
class Account {
  attributes
    owner: Traveller . account
    balance: int
}
```

```
class Traveller {
  attributes
    name: string
    reglist: set(Hotel . registered)[*]
}
```

```
class Hotel {
  attributes
    name: string
    registered: set(Traveller . reglist)[*]
  methods
    register {
        t? : extent(Traveller)
      & t? /: registered
      ==>
          registered := registered \/ {t?}
      || t?.reglist := t?.reglist \/ {this}
    }
}
```

- How do you specify the ***scanner*** and ***parser*** accordingly?

Example Compiler 2: Output/Target

- Class ***associations*** are transformed to ***database schemas***.

```
CREATE TABLE `Account`(
  `oid` INTEGER AUTO_INCREMENT,`balance` INTEGER,
  PRIMARY KEY (`oid`));
CREATE TABLE `Traveller`(
  `oid` INTEGER AUTO_INCREMENT,`name` CHAR(30),
  PRIMARY KEY (`oid`));
CREATE TABLE `Hotel`(
  `oid` INTEGER AUTO_INCREMENT,`name` CHAR(30),
  PRIMARY KEY (`oid`));
CREATE TABLE `Account_owner_Traveller_account`(
  `oid` INTEGER AUTO_INCREMENT, `owner` INTEGER, `account` INTEGER,
  PRIMARY KEY (`oid`));
CREATE TABLE `Traveller_reglist_Hotel_registered`(
  `oid` INTEGER AUTO_INCREMENT, `reglist` INTEGER, `registered` INTEGER,
  PRIMARY KEY (`oid`));
```

- Method ***predicates*** are compiled into ***stored procedures***.

```
CREATE PROCEDURE `Hotel_register`(IN `this?` INTEGER, IN `t?` INTEGER)
  BEGIN
    ...
  END
```

Example Compiler 2: Transforming Updates

Challenge : Transform ***dot notations*** into ***relational queries***.

e.g., The AST corresponding to the following dot notation
(in the context of class `Account`,
retrieving the owner's list of registrations)

```
this.owner.reglist
```

may be transformed into the following (nested) table lookups:

```
SELECT (VAR 'reglist')
       (TABLE 'Hotel_registered_Traveller_reglist')
       (VAR 'registered' = (SELECT (VAR 'owner')
                            (TABLE 'Account_owner_Traveller_account')
                            (VAR 'owner' = VAR 'this')))
```


Beyond this lecture . . .

- Read Chapter 1 of EAC2 to find out more about Example Compiler 1
- Read this paper to find out more about Example Compiler 2:

  http://dx.doi.org/10.4204/EPTCS.105.8

Index (1)

Index (2)

Scanner: Lexical Analysis

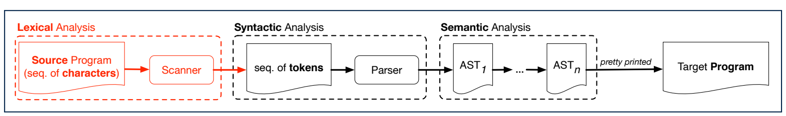
Readings: EAC2 Chapter 2

EECS4302 A:
Compilers and Interpreters
Fall 2022

CHEN-WEI WANG

Scanner in Context

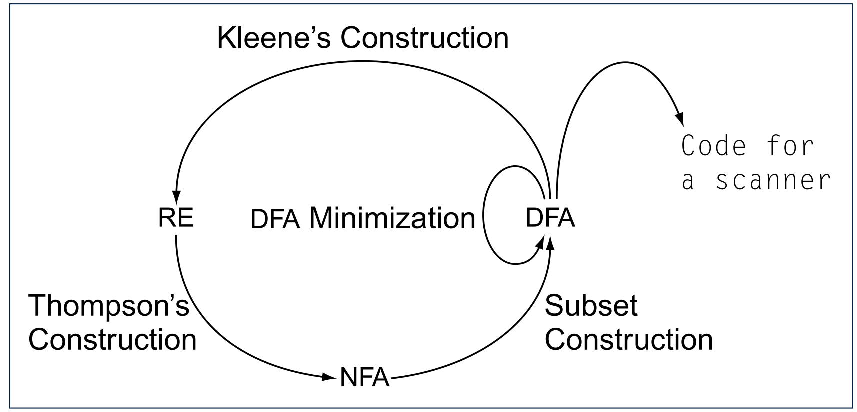
- Recall:



- Treats the input programas as a ***a sequence of characters***
- Applies rules ***recognizing*** character sequences as ***tokens***

  [***lexical*** analysis]
- Upon termination:
  - Reports character sequences not recognizable as tokens
  - Produces a ***a sequence of tokens***
- Only part of compiler touching ***every character*** in input program.
- Tokens *recognizable* by scanner constitute a *regular language*.

Scanner: Formulation & Implementation



Alphabets

An *alphabet* is a ***finite***, ***nonempty*** set of symbols.

- The convention is to write Σ, possibly with a informative subscript, to denote the alphabet in question.
- Use either a ***set enumeration*** or a ***set comprehension*** to define your own alphabet.

  e.g., $\Sigma_{eng} = \{a, b, \ldots, z, A, B, \ldots, Z\}$ [the English alphabet]
  e.g., $\Sigma_{bin} = \{0, 1\}$ [the binary alphabet]
  e.g., $\Sigma_{dec} = \{d \mid 0 \leq d \leq 9\}$ [the decimal alphabet]
  e.g., Σ_{key} [the keyboard alphabet]

Strings (1)

- A **string** or a **word** is ***finite*** sequence of symbols chosen from some ***alphabet***.
  - e.g., Oxford is a string over the English alphabet Σ_{eng}
  - e.g., 01010 is a string over the binary alphabet Σ_{bin}
  - e.g., 01010.01 is **not** a string over Σ_{bin}
  - e.g., 57 is a string over the decimal alphabet Σ_{dec}
- It is **not** correct to say, e.g., $01010 \in \Sigma_{bin}$ [Why?]
- The *length* of a string w, denoted as $|w|$, is the number of characters it contains.
  - e.g., $|Oxford| = 6$
  - ϵ is the ***empty string*** ($|\epsilon| = 0$) that may be from any alphabet.
- Given two strings x and y, their *concatenation*, denoted as xy, is a new string formed by a copy of x followed by a copy of y.
  - e.g., Let $x = 01101$ and $y = 110$, then $xy = 01101110$
  - The empty string ϵ is the *identity for concatenation*:
    $\epsilon w = w = w\epsilon$ for any string w

Strings (2)

- Given an ***alphabet*** Σ, we write Σ^k, where $k \in \mathbb{N}$, to denote the ***set of strings of length k from*** Σ

$$\Sigma^k = \{w \mid \underbrace{w \text{ is a string over } \Sigma}_{\textit{more formal?}} \wedge |w| = k\}$$

  - e.g., $\{0,1\}^2 = \{00, 01, 10, 11\}$
  - Given Σ, Σ^0 is $\{\epsilon\}$
- Given Σ, Σ^+ is the ***set of nonempty strings***.

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots = \{w \mid w \in \Sigma^k \wedge k > 0\} = \bigcup_{k>0} \Sigma^k$$

- Given Σ, Σ^* is the ***set of strings of all possible lengths***.

$$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$$

Review Exercises: Strings

1. What is $|\{a, b, \ldots, z\}^5|$?
2. Enumerate, in a systematic manner, the set $\{a, b, c\}^4$.
3. Explain the difference between Σ and Σ^1.
4. Prove or disprove: $\Sigma_1 \subseteq \Sigma_2 \Rightarrow \Sigma_1^* \subseteq \Sigma_2^*$

Languages

- *A language L over* Σ (where $|\Sigma|$ is finite) is a set of strings s.t.
  $$L \subseteq \Sigma^*$$
- When useful, include an informative subscript to denote the ***language*** L in question.
  - e.g., The language of ***compilable*** Java programs
    $$L_{Java} = \{prog \mid prog \in \Sigma_{key}^* \wedge prog \texttt{ compiles in Eclipse}\}$$
    Note. *prog* compiling means **no** ***lexical***, ***syntactical***, or ***type*** errors.
  - e.g., The language of strings with n 0's followed by n 1's ($n \geq 0$)
    $$\{\epsilon, 01, 0011, 000111, \ldots\} = \{0^n 1^n \mid n \geq 0\}$$
  - e.g., The language of strings with an equal number of 0's and 1's
    $$\{\epsilon, 01, 10, 0011, 0101, 0110, 1100, 1010, 1001, \ldots\}$$
    $$= \{w \mid \texttt{\# of 0's in } w = \texttt{\# of 1's in } w\}$$

Review Exercises: Languages

1. Use ***set comprehensions*** to define the following ***languages***. Be as ***formal*** as possible.
   - A language over $\{0, 1\}$ consisting of strings beginning with some 0's (possibly none) followed by at least as many 1's.
   - A language over $\{a, b, c\}$ consisting of strings beginning with some a's (possibly none), followed by some b's and then some c's, s.t. the # of a's is at least as many as the sum of #'s of b's and c's.
2. Explain the difference between the two languages $\{\epsilon\}$ and $\varnothing$.
3. Justify that Σ^*, $\varnothing$, and $\{\epsilon\}$ are all languages over Σ.
4. Prove or disprove: If L is a language over Σ, and $\Sigma_2 \supseteq \Sigma$, then L is also a language over Σ_2.
   Hint: Prove that $\Sigma \subseteq \Sigma_2 \wedge L \subseteq \Sigma^* \Rightarrow L \subseteq \Sigma_2^*$
5. Prove or disprove: If L is a language over Σ, and $\Sigma_2 \subseteq \Sigma$, then L is also a language over Σ_2.
   Hint: Prove that $\Sigma_2 \subseteq \Sigma \wedge L \subseteq \Sigma^* \Rightarrow L \subseteq \Sigma_2^*$

Problems

- Given a ***language*** L over some ***alphabet*** Σ, a *problem* is the ***decision*** on whether or not a given ***string*** w is a member of L.

$$w \in L$$

Is this equivalent to deciding $w \in \Sigma^*$? [***No***]

$w \in \Sigma^* \Rightarrow w \in L$ is **not** necessarily true.

- e.g., The Java compiler solves the problem of ***deciding*** if a user-supplied ***string of symbols*** is a **member** of L_{Java}.

Regular Expressions (RE): Introduction

- *Regular expressions* (RegExp's) are:
  - A type of language-defining notation
    - This is ***similar*** to the equally-expressive ***DFA***, ***NFA***, and ϵ-***NFA***.
  - *Textual* and look just like a programming language
    - e.g., Set of strings denoted by `01* + 10*`? [specify formally]
      $L = \{0x \mid x \in \{1\}^*\} \cup \{1x \mid x \in \{0\}^*\}$
    - e.g., Set of strings denoted by `(0*10*10*)*10*`?
      $L = \{w \mid w \texttt{ has odd \# of 1's}\}$
    - This is ***dissimilar*** to the diagrammatic ***DFA***, ***NFA***, and ϵ-***NFA***.
    - RegExp's can be considered as a "user-friendly" alternative to ***NFA*** for describing software components. [e.g., text search]
    - Writing a RegExp is like writing an algebraic expression, using the defined operators, e.g., `((4 + 3) * 5) % 6`
- Despite the programming convenience they provide, ***RegExp's***, ***DFA***, ***NFA***, and ϵ-***NFA*** are all *provably equivalent*.
  - They are capable of defining **all** and **only** regular languages.

RE: Language Operations (1)

- Given Σ of input alphabets, the simplest RegExp is? [$s \in \Sigma^1$]
  - e.g., Given $\Sigma = \{a, b, c\}$, expression a denotes the language $\{ a \}$ consisting of a single string a.
- Given two languages $L, M \in \Sigma^*$, there are 3 operators for building a *larger language* out of them:

1. ***Union***

$$L \cup M = \{w \mid w \in L \vee w \in M\}$$

In the textual form, we write $+$ for union.

2. ***Concatenation***

$$LM = \{xy \mid x \in L \wedge y \in M\}$$

In the textual form, we write either `.` or nothing at all for concatenation.

RE: Language Operations (2)

3. ***Kleene Closure*** (or ***Kleene Star***)

$$L^* = \bigcup_{i \geq 0} L^i$$

where

$$\begin{array}{lcl} L^0 & = & \{\epsilon\} \\ L^1 & = & L \\ L^2 & = & \{x_1 x_2 \mid x_1 \in L \land x_2 \in L\} \\ \ldots & & \\ L^i & = & \{ \underbrace{x_1 x_2 \ldots x_i}_{i \text{ concatenations}} \mid x_j \in L \land 1 \leq j \leq i\} \\ \ldots & & \end{array}$$

In the textual form, we write $\star$ for closure.

Question: What is $|L^i|$ $(i \in \mathbb{N})$? [$|L|^i$]
Question: Given that $L = \{0\}^*$, what is L^*? [L]

RE: Construction (1)

We may build *regular expressions* ***recursively***:

- Each (***basic*** or ***recursive***) form of regular expressions denotes a ***language*** (i.e., a set of strings that it accepts).
- ***Base Case***:
  - Constants ϵ and $\varnothing$ are regular expressions.

$$\begin{array}{lcl} L(\ \epsilon\ ) & = & \{\epsilon\} \\ L(\ \varnothing\ ) & = & \varnothing \end{array}$$

  - An input symbol $a \in \Sigma$ is a regular expression.

$$L(\ \mathbf{a}\ ) = \{a\}$$

  If we want a regular expression for the language consisting of only the string $w \in \Sigma^*$, we write w as the regular expression.
  - Variables such as **L**, **M**, *etc.*, might also denote languages.

RE: Construction (2)

- ***Recursive Case***: Given that E and F are regular expressions:
  - The union $E + F$ is a regular expression.

$$L(\ \mathbf{E + F}\ ) = L(E) \cup L(F)$$

  - The concatenation EF is a regular expression.

$$L(\ \mathbf{EF}\ ) = L(E)L(F)$$

  - Kleene closure of E is a regular expression.

$$L(\ E^*\ ) = (L(E))^*$$

  - A parenthesized E is a regular expression.

$$L(\ \mathbf{(E)}\ ) = L(E)$$

RE: Construction (3)

Exercises:

- $\varnothing + L$ [$\varnothing + L = L = \varnothing + L$]
- $\varnothing L$ [$\varnothing L = \varnothing = L\varnothing$]
- $\varnothing^*$

$$\begin{array}{lcl} \varnothing^* & = & \varnothing^0 \cup \varnothing^1 \cup \varnothing^2 \cup \ldots \\ & = & \{\epsilon\} \cup \varnothing \cup \varnothing \cup \ldots \\ & = & \{\epsilon\} \end{array}$$

- $\varnothing^* L$ [$\varnothing^* L = L = L\varnothing^*$]

RE: Construction (4)

Write a regular expression for the following language

$$\{ w \mid w \texttt{ has alternating } 0\texttt{'s and } 1\texttt{'s} \}$$

- Would $(01)^*$ work? [alternating 10's?]
- Would $(01)^* + (10)^*$ work? [starting and ending with 1?]
- $0(10)^* + (01)^* + (10)^* + 1(01)^*$
- It seems that:
  - 1st and 3rd terms have $(10)^*$ as the common factor.
  - 2nd and 4th terms have $(01)^*$ as the common factor.
- Can we simplify the above regular expression?
- $(\epsilon + 0)(10)^* + (\epsilon + 1)(01)^*$

RE: Review Exercises

Write the regular expressions to describe the following languages:

- $\{ w \mid w \texttt{ ends with } 01 \}$
- $\{ w \mid w \texttt{ contains } 01 \texttt{ as a substring} \}$
- $\{ w \mid w \texttt{ contains no more than three consecutive } 1\texttt{'s} \}$
- $\{ w \mid w \texttt{ ends with } 01 \lor w \texttt{ has an odd \# of } 0\texttt{'s} \}$
- $$\left\{ sx.y \;\middle|\; \begin{array}{ll} & s \in \{+, -, \epsilon\} \\ \land & x \in \Sigma_{dec}^* \\ \land & y \in \Sigma_{dec}^* \\ \land & \neg(x = \epsilon \land y = \epsilon) \end{array} \right\}$$
- $$\left\{ xy \;\middle|\; \begin{array}{ll} & x \in \{0,1\}^* \land y \in \{0,1\}^* \\ \land & x \texttt{ has alternating } 0\texttt{'s and } 1\texttt{'s} \\ \land & y \texttt{ has an odd \# } 0\texttt{'s and an odd \# } 1\texttt{'s} \end{array} \right\}$$

RE: Operator Precedence

- In an order of ***decreasing precedence***:
  - Kleene star operator
  - Concatenation operator
  - Union operator
- When necessary, use ***parentheses*** to force the intended order of evaluation.
- e.g.,
  - 10^* vs. $(10)^*$ [10^* is equivalent to $1(0^*)$]
  - $01^* + 1$ vs. $0(1^* + 1)$ [$01^* + 1$ is equivalent to $(0(1^*)) + (1)$]
  - $0 + 1^*$ vs. $(0 + 1)^*$ [$0 + 1^*$ is equivalent to $(0) + (1^*)$]

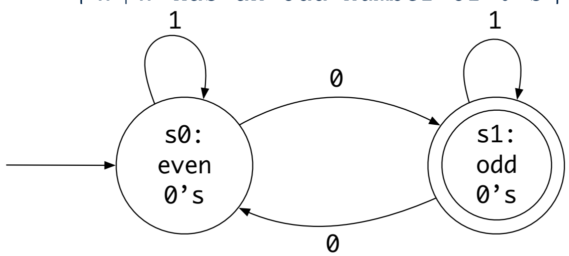
DFA: Deterministic Finite Automata (1.1)

- A *deterministic finite automata (DFA)* is a ***finite state machine*** (***FSM***) that ***accepts*** (or ***recognizes***) a pattern of behaviour.
  - For ***lexical*** analysis, we study patterns of ***strings*** (i.e., how ***alphabet*** symbols are ordered).
  - Unless otherwise specified, we consider strings in $\{0,1\}^*$
  - Each pattern contains the set of satisfying strings.
  - We describe the patterns of strings using set comprehensions:
    - $\{ w \mid w \texttt{ has an odd number of } 0\texttt{'s} \}$
    - $\{ w \mid w \texttt{ has an even number of } 1\texttt{'s} \}$
    - $\left\{ w \;\middle|\; \begin{array}{ll} & w \neq \epsilon \\ \land & w \texttt{ has equal \# of alternating } 0\texttt{'s and } 1\texttt{'s} \end{array} \right\}$
    - $\{ w \mid w \texttt{ contains } 01 \texttt{ as a substring} \}$
    - $\left\{ w \;\middle|\; \begin{array}{ll} & w \texttt{ has an even number of } 0\texttt{'s} \\ \land & w \texttt{ has an odd number of } 1\texttt{'s} \end{array} \right\}$
- Given a pattern description, we design a ***DFA*** that accepts it.
  - The resulting ***DFA*** can be transformed into an executable program.

DFA: Deterministic Finite Automata (1.2)

- The ***transition diagram*** below defines a DFA which ***accepts***/***recognizes*** exactly the language

$$\{ w \mid w \texttt{ has an odd number of 0's} \}$$

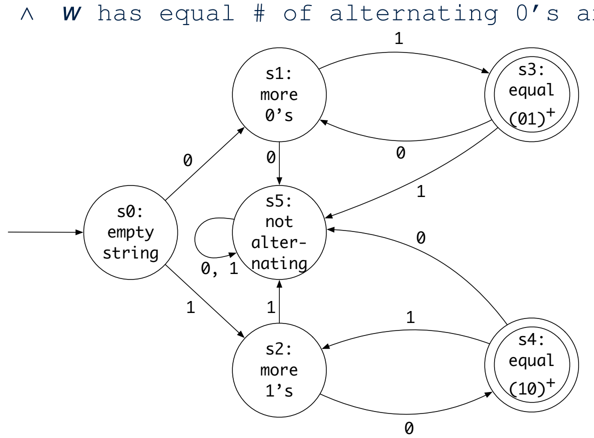


- Each **incoming** or **outgoing** arc (called a ***transition***) corresponds to an input alphabet symbol.
- s_0 with an unlabelled **incoming** transition is the ***start state***.
- s_3 drawn as a double circle is a ***final state***.
- All states have **outgoing** transitions covering $\{0, 1\}$.

DFA: Deterministic Finite Automata (1.3)

The ***transition diagram*** below defines a DFA which ***accepts***/***recognizes*** exactly the language

$$\left\{ w \mid \begin{array}{l} w \neq \epsilon \\ \land \;\; w \texttt{ has equal \# of alternating 0's and 1's} \end{array} \right\}$$



Review Exercises: Drawing DFAs

Draw the transition diagrams for DFAs which accept other example string patterns:

- $\{ w \mid w \texttt{ has an even number of 1's} \}$
- $\{ w \mid w \texttt{ contains 01 as a substring} \}$
- $\left\{ w \mid \begin{array}{l} w \texttt{ has an even number of 0's} \\ \land \;\; w \texttt{ has an odd number of 1's} \end{array} \right\}$

DFA: Deterministic Finite Automata (2.1)

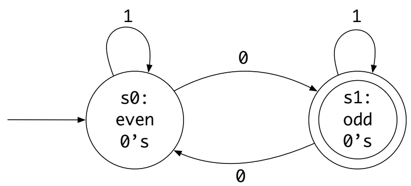
A *deterministic finite automata (DFA)* is a 5-tuple

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

- Q is a finite set of *states*.
- Σ is a finite set of *input symbols* (i.e., the *alphabet*).
- $\delta : (Q \times \Sigma) \to Q$ is a *transition function*

  δ takes as arguments a state and an input symbol and returns a state.
- $q_0 \in Q$ is the *start state*.
- $F \subseteq Q$ is a set of *final* or *accepting states*.

DFA: Deterministic Finite Automata (2.2)



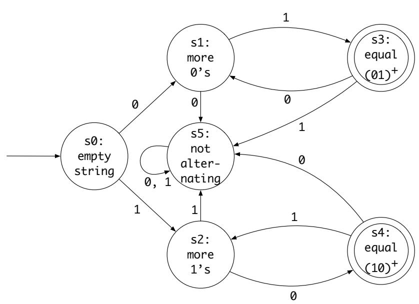
We formalize the above DFA as $M = (Q, \Sigma, \delta, q_0, F)$, where

- $Q = \{s_0, s_1\}$
- $\Sigma = \{0, 1\}$
- $\delta = \{((s_0, 0), s_1), ((s_0, 1), s_0), ((s_1, 0), s_0), ((s_1, 1), s_1)\}$

state \ input	0	1
s_0	s_1	s_0
s_1	s_0	s_1

- $q_0 = s_0$
- $F = \{s_1\}$

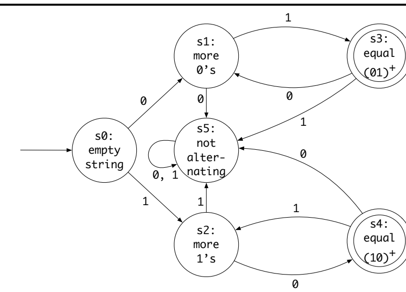
DFA: Deterministic Finite Automata (2.3.1)



We formalize the above DFA as $M = (Q, \Sigma, \delta, q_0, F)$, where

- $Q = \{s_0, s_1, s_2, s_3, s_4, s_5\}$
- $\Sigma = \{0, 1\}$
- $q_0 = s_0$
- $F = \{s_3, s_4\}$

DFA: Deterministic Finite Automata (2.3.2)



- δ =

state \ input	0	1
s_0	s_1	s_2
s_1	s_5	s_3
s_2	s_4	s_5
s_3	s_1	s_5
s_4	s_5	s_2
s_5	s_5	s_5

DFA: Deterministic Finite Automata (2.4)

- Given a DFA $M = (Q, \Sigma, \delta, q_0, F)$:
  - We write *L(M)* to denote the *language of M*: the set of strings that M ***accepts***.
  - A string is ***accepted*** if it results in a sequence of transitions: beginning from the ***start*** state and ending in a ***final*** state.

$$L(M) = \left\{ \begin{array}{l} a_1 a_2 \dots a_n \mid \\ \quad 1 \le i \le n \ \wedge \ a_i \in \Sigma \ \wedge \ \delta(q_{i-1}, a_i) = q_i \ \wedge \ q_n \in F \end{array} \right\}$$

  - M ***rejects*** any string $w \notin L(M)$.
- We may also consider *L(M)* as concatenations of labels from the set of all valid ***paths*** of *M*'s transition diagram; each such path starts with q_0 and ends in a state in F.

DFA: Deterministic Finite Automata (2.5)

- Given a ***DFA*** $M = (Q, \Sigma, \delta, q_0, F)$, we may simplify the definition of $L(M)$ by extending δ (which takes an input symbol) to $\hat{\delta}$ (which takes an input string).

$$\hat{\delta} : (Q \times \Sigma^*) \to Q$$

We may define $\hat{\delta}$ recursively, using δ!

$$\begin{array}{rcl} \hat{\delta}(q, \epsilon) & = & q \\ \hat{\delta}(q, xa) & = & \delta(\ \hat{\delta}(q, x), a\ ) \end{array}$$

where $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$

- A neater definition of $L(M)$: the set of strings $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w)$ is an ***accepting state***.

$$L(M) = \{w \mid w \in \Sigma^* \wedge \hat{\delta}(q_0, w) \in F\}$$

- A language L is said to be a *regular language*, if there is some ***DFA M*** such that $L = L(M)$.

Review Exercises: Formalizing DFAs

Formalize DFAs (as 5-tuples) for the other example string patterns mentioned:

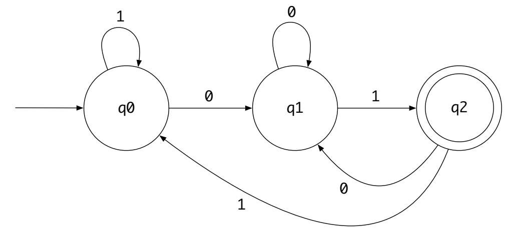
- $\{ w \mid w \texttt{ has an even number of 0's} \}$
- $\{ w \mid w \texttt{ contains 01 as a substring} \}$
- $\left\{ w \mid \begin{array}{l} w \texttt{ has an even number of 0's} \\ \wedge\ w \texttt{ has an odd number of 1's} \end{array} \right\}$

NFA: Nondeterministic Finite Automata (1.1)

Problem: Design a DFA that accepts the following language:

$$L = \{ x01 \mid x \in \{0, 1\}^* \}$$

That is, L is the set of strings of 0s and 1s ending with 01.



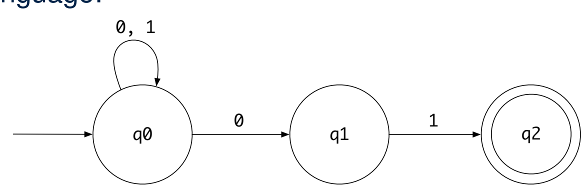
Given an input string w, we may simplify the above DFA by:

- *nondeterministically* treating state q_0 as both:
  - a state *ready* to read the last two input symbols from w
  - a state *not yet ready* to read the last two input symbols from w
- substantially reducing the outgoing transitions from q_1 and q_2

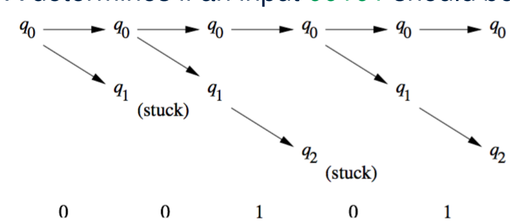
Compare the above DFA with the DFA in slide 39.

NFA: Nondeterministic Finite Automata (1.2)

- A *non-deterministic finite automata (NFA)* that accepts the same language:



- How an NFA determines if an input *00101* should be processed:



NFA: Nondeterministic Finite Automata (2)

- A *nondeterministic finite automata (NFA)*, like a **DFA**, is a ***FSM*** that ***accepts*** (or ***recognizes***) a pattern of behaviour.
- An ***NFA*** being *nondeterministic* means that from a given state, the **same input label** might corresponds to **multiple transitions** that lead to **distinct states**.
  - Each such transition offers an ***alternative path***.
  - Each alternative path is explored in parallel.
  - If **there exists** an alternative path that ***succeeds*** in processing the input string, then we say the ***NFA accepts*** that input string.
  - If **all** alternative paths get stuck at some point and ***fail*** to process the input string, then we say the ***NFA rejects*** that input string.
- ***NFAs*** are often more succinct (i.e., fewer states) and easier to design than **DFAs**.
- However, ***NFAs*** are just as ***expressive*** as are **DFAs**.
  - We can **always** convert an ***NFA*** to a **DFA**.

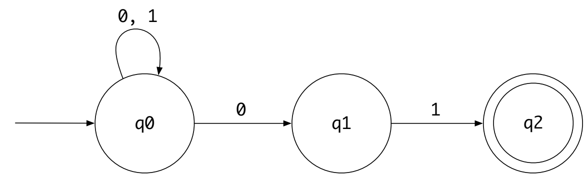
NFA: Nondeterministic Finite Automata (3.1)

- A *nondeterministic finite automata (NFA)* is a 5-tuple

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

  - Q is a finite set of *states*.
  - Σ is a finite set of *input symbols* (i.e., the *alphabet*).
  - $\delta : (Q \times \Sigma) \rightarrow \mathbb{P}(Q)$ is a *transition function*
    - Given a state and an input symbol, δ returns a set of states.
    - Equivalently, we can write: $\delta : (Q \times \Sigma) \nrightarrow Q$ [a partial function]
  - $q_0 \in Q$ is the *start state*.
  - $F \subseteq Q$ is a set of *final* or *accepting states*.
- What is the difference between a ***DFA*** and an ***NFA***?
  - δ of a ***DFA*** returns a single state.
  - δ of an ***NFA*** returns a (possibly empty) set of states.

NFA: Nondeterministic Finite Automata (3.2)



Given an input string 00101:

- **Read 0**: $\delta(q_0, 0) = \{\ q_0, q_1\ \}$
- **Read 0**: $\delta(q_0, 0) \cup \delta(q_1, 0) = \{\ q_0, q_1\ \} \cup \varnothing = \{\ q_0, q_1\ \}$
- **Read 1**: $\delta(q_0, 1) \cup \delta(q_1, 1) = \{\ q_0\ \} \cup \{\ q_2\ \} = \{\ q_0, q_2\ \}$
- **Read 0**: $\delta(q_0, 0) \cup \delta(q_2, 0) = \{\ q_0, q_1\ \} \cup \varnothing = \{\ q_0, q_1\ \}$
- **Read 1**: $\delta(q_0, 1) \cup \delta(q_1, 1) = \{\ q_0, q_1\ \} \cup \{\ q_2\ \} = \{\ q_0, q_1, q_2\ \}$

$\because \{\ q_0, q_1, q_2\ \} \cap \{\ q_2\ \} \neq \varnothing \therefore$ `00101 is accepted`

NFA: Nondeterministic Finite Automata (3.3)

- Given a *NFA* $M = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$, we may simplify the definition of $L(M)$ by extending δ (which takes an input symbol) to $\hat{\delta}$ (which takes an input string).

$$\hat{\delta} : (Q \times \Sigma^*) \rightarrow \mathbb{P}(Q)$$

We may define $\hat{\delta}$ recursively, using δ!

$$\begin{array}{lcl} \hat{\delta}(q, \epsilon) & = & \{q\} \\ \hat{\delta}(q, xa) & = & \bigcup\{\delta(q', a) \mid q' \in \hat{\delta}(q, x)\} \end{array}$$

where $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$

- A neater definition of $L(M)$: the set of strings $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w)$ contains **at least one** *accepting state*.

$$L(M) = \{w \mid w \in \Sigma^* \land \hat{\delta}(q_0, w) \cap F \neq \varnothing\}$$

DFA ≡ NFA (1)

- For many languages, constructing an accepting ***NFA*** is easier than a ***DFA***.
- From each state of an ***NFA***:
  - Outgoing transitions need **not** cover the entire Σ.
  - From a given state, the same symbol may ***non-deterministically*** lead to multiple states.
- In practice:
  - An ***NFA*** has just as many states as its equivalent DFA does.
  - An ***NFA*** often has fewer transitions than its equivalent ***DFA*** does.
- In the **worst** case:
  - While an ***NFA*** has n states, its equivalent ***DFA*** has 2^n states.
- Nonetheless, an ***NFA*** is still just as ***expressive*** as a ***DFA***.
  - A ***language*** accepted by some ***NFA*** is accepted by some ***DFA***:

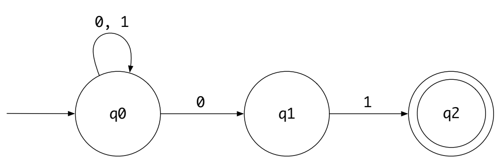
$$\forall N \bullet N \in NFA \Rightarrow (\exists D \bullet D \in DFA \wedge L(D) = L(N))$$

  - And vice versa, trivially?

$$\forall D \bullet D \in DFA \Rightarrow (\exists N \bullet N \in NFA \wedge L(D) = L(N))$$

DFA ≡ NFA (2.2): Lazy Evaluation (1)

Given an ***NFA***:



Subset construction (with ***lazy evaluation***) produces a ***DFA*** with δ as:

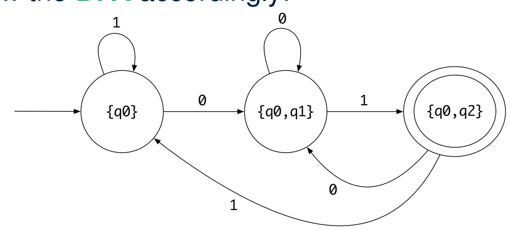
state \ input	0	1
$\{q_0\}$	$\delta(q_0, 0)$ $= \{q_0, q_1\}$	$\delta(q_0, 1)$ $= \{q_0\}$
$\{q_0, q_1\}$	$\delta(q_0, 0) \cup \delta(q_1, 0)$ $= \{q_0, q_1\} \cup \varnothing$ $= \{q_0, q_1\}$	$\delta(q_0, 1) \cup \delta(q_1, 1)$ $= \{q_0\} \cup \{q_2\}$ $= \{q_0, q_2\}$
$\{q_0, q_2\}$	$\delta(q_0, 0) \cup \delta(q_2, 0)$ $= \{q_0, q_1\} \cup \varnothing$ $= \{q_0, q_1\}$	$\delta(q_0, 1) \cup \delta(q_2, 1)$ $= \{q_0\} \cup \varnothing$ $= \{q_0\}$

DFA ≡ NFA (2.2): Lazy Evaluation (2)

Applying ***subset construction*** (with ***lazy evaluation***), we arrive in a ***DFA*** transition table:

state \ input	0	1
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

We then draw the ***DFA*** accordingly:



Compare the above DFA with the DFA in slide 31.

DFA ≡ NFA (2.2): Lazy Evaluation (3)

- Given an ***NFA*** $N = (Q_N, \Sigma_N, \delta_N, q_0, F_N)$:

```
ALGORITHM: ReachableSubsetStates
  INPUT: q0 : Q_N   ;   OUTPUT: Reachable ⊆ ℙ(Q_N)
PROCEDURE:
  Reachable := { {q0} }
  ToDiscover := { {q0} }
  while (ToDiscover ≠ ∅) {
    choose S : ℙ(Q_N) such that S ∈ ToDiscover
    remove S from ToDiscover
    NotYetDiscovered :=
        ( { {δ_N(s,0) | s ∈ S} } ∪ { {δ_N(s,1) | s ∈ S} } ) \ Reachable
    Reachable := Reachable ∪ NotYetDiscovered
    ToDiscover := ToDiscover ∪ NotYetDiscovered
  }
  return Reachable
```

- RT of *ReachableSubsetStates*? [$O(2^{|Q_N|})$]
- Often only a small portion of the $|\mathbb{P}(Q_N)|$ ***subset states*** is ***reachable*** from $\{q_0\}$ ⇒ ***Lazy Evaluation*** efficient in practice!

ϵ-NFA: Examples (1)

Draw the NFA for the following two languages:

1.

$$\left\{ xy \;\middle|\; \begin{array}{ll} & x \in \{0,1\}^* \\ \wedge & y \in \{0,1\}^* \\ \wedge & x \texttt{ has alternating 0's and 1's} \\ \wedge & y \texttt{ has an odd \# 0's and an odd \# 1's} \end{array} \right\}$$

2.

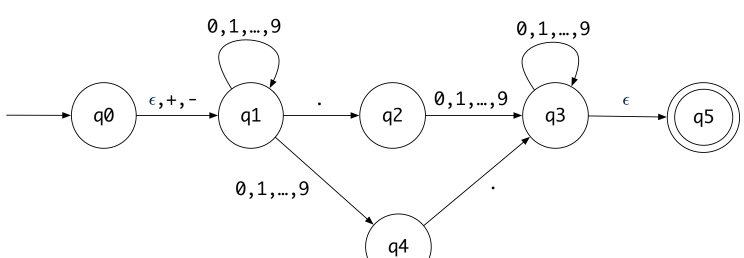
$$\left\{ w : \{0,1\}^* \;\middle|\; \begin{array}{ll} & w \texttt{ has alternating 0's and 1's} \\ \vee & w \texttt{ has an odd \# 0's and an odd \# 1's} \end{array} \right\}$$

3.

$$\left\{ sx.y \;\middle|\; \begin{array}{ll} & s \in \{+,-,\epsilon\} \\ \wedge & x \in \Sigma_{dec}^* \\ \wedge & y \in \Sigma_{dec}^* \\ \wedge & \neg(x = \epsilon \wedge y = \epsilon) \end{array} \right\}$$

ϵ-NFA: Examples (2)

$$\left\{ sx.y \;\middle|\; \begin{array}{ll} & s \in \{+,-,\epsilon\} \\ \wedge & x \in \Sigma_{dec}^* \\ \wedge & y \in \Sigma_{dec}^* \\ \wedge & \neg(x = \epsilon \wedge y = \epsilon) \end{array} \right\}$$



From q_0 to q_1, reading a sign is **optional**: a *plus* or a *minus*, or *nothing at all* (i.e., ϵ).

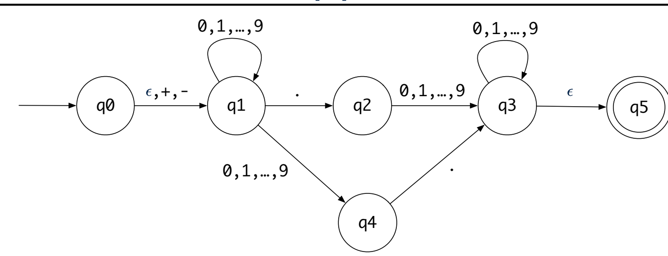
ϵ-NFA: Formalization (1)

An *ϵ-NFA* is a 5-tuple

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

- Q is a finite set of *states*.
- Σ is a finite set of *input symbols* (i.e., the *alphabet*).
- $\delta : (Q \times (\Sigma \cup \{\epsilon\})) \to \mathbb{P}(Q)$ is a *transition function*
  - δ takes as arguments a state and an input symbol, or *an empty string* ϵ, and returns a set of states.
- $q_0 \in Q$ is the *start state*.
- $F \subseteq Q$ is a set of *final* or *accepting states*.

ϵ-NFA: Formalization (2)



Draw a transition table for the above NFA's δ function:

	ϵ	+, -	.	0 .. 9
q_0	$\{q_1\}$	$\{q_1\}$	$\varnothing$	$\varnothing$
q_1	$\varnothing$	$\varnothing$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_3	$\{q_5\}$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_4	$\varnothing$	$\varnothing$	$\{q_3\}$	$\varnothing$
q_5	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

ϵ-NFA: Epsilon-Closures (1)

- Given ϵ-*NFA* N

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

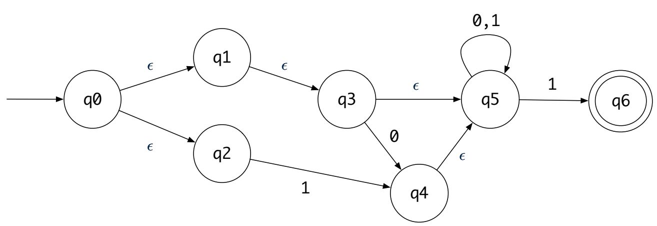
we define the *epsilon closure* (or ϵ-*closure*) as a function

$$\mathtt{ECLOSE} : Q \to \mathbb{P}(Q)$$

- For any state $q \in Q$

$$\mathtt{ECLOSE}(q) = \{q\} \cup \bigcup_{p \in \delta(q,\epsilon)} \mathtt{ECLOSE}(p)$$

ϵ-NFA: Epsilon-Closures (2)



$$
\begin{array}{ll}
& \mathtt{ECLOSE}(q_0) \\
= & \{\ \delta(q_0,\epsilon) = \{q_1, q_2\}\ \} \\
& \{q_0\} \cup \mathtt{ECLOSE}(q_1) \cup \mathtt{ECLOSE}(q_2) \\
= & \{\ \mathtt{ECLOSE}(q_1),\ \ \delta(q_1,\epsilon) = \{q_3\},\ \ \mathtt{ECLOSE}(q_2),\ \ \delta(q_2,\epsilon) = \varnothing\ \} \\
& \{q_0\} \cup (\ \{q_1\} \cup \mathtt{ECLOSE}(q_3)\ ) \cup (\ \{q_2\} \cup \varnothing\ ) \\
= & \{\ \mathtt{ECLOSE}(q_3),\ \ \delta(q_3,\epsilon) = \{q_5\}\ \} \\
& \{q_0\} \cup (\ \{q_1\} \cup (\ \{q_3\} \cup \mathtt{ECLOSE}(q_5)\ )\ ) \cup (\ \{q_2\} \cup \varnothing\ ) \\
= & \{\ \mathtt{ECLOSE}(q_5),\ \ \delta(q_5,\epsilon) = \varnothing\ \} \\
& \{q_0\} \cup (\ \{q_1\} \cup (\ \{q_3\} \cup (\ \{q_5\} \cup \varnothing\ )\ )\ ) \cup (\ \{q_2\} \cup \varnothing\ )
\end{array}
$$

ϵ-NFA: Formalization (3)

- Given a ϵ-*NFA* $M = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$, we may simplify the definition of *L(M)* by extending δ (which takes an input symbol) to $\hat{\delta}$ (which takes an input string).

$$\hat{\delta} : (Q \times \Sigma^*) \to \mathbb{P}(Q)$$

We may define $\hat{\delta}$ recursively, using δ!

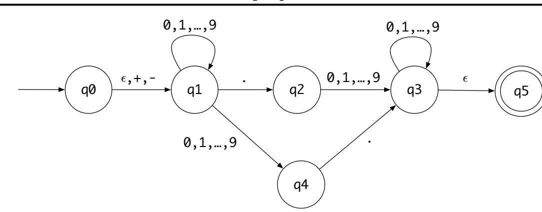
$$
\begin{array}{lcl}
\hat{\delta}(q,\epsilon) & = & \mathtt{ECLOSE}(q) \\
\hat{\delta}(q,xa) & = & \bigcup\{\ \mathtt{ECLOSE}(q'') \mid q'' \in \delta(q',a) \wedge q' \in \hat{\delta}(q,x)\ \}
\end{array}
$$

where $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$

- Then we define $L(M)$ as the set of strings $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w)$ contains **at least one** *accepting state*.

$$L(M) = \{w \mid w \in \Sigma^* \wedge \hat{\delta}(q_0, w) \cap F \neq \varnothing\}$$

ϵ-NFA: Formalization (4)



Given an input string 5.6:

$\hat{\delta}(q_0,\epsilon) = \mathtt{ECLOSE}(q_0) = \{q_0, q_1\}$

- **Read 5**: $\delta(q_0,5) \cup \delta(q_1,5) = \varnothing \cup \{q_1, q_4\} = \{\ q_1, q_4\ \}$

  $\hat{\delta}(q_0,5) = \mathtt{ECLOSE}(q_1) \cup \mathtt{ECLOSE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$

- **Read .**: $\delta(q_1,.) \cup \delta(q_4,.) = \{q_2\} \cup \{q_3\} = \{\ q_2, q_3\ \}$

  $\hat{\delta}(q_0,5.) = \mathtt{ECLOSE}(q_2) \cup \mathtt{ECLOSE}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}$

- **Read 6**: $\delta(q_2,6) \cup \delta(q_3,6) \cup \delta(q_5,6) = \{q_3\} \cup \{q_3\} \cup \varnothing = \{\ q_3\ \}$

  $\hat{\delta}(q_0,5.6) = \mathtt{ECLOSE}(q_3) = \{q_3, q_5\}$ [5.6 is *accepted*]

DFA $\equiv$ ϵ-NFA: Extended Subset Const. (1)

Subset construction (with ***lazy evaluation*** and *epsilon closures*) produces a ***DFA*** transition table.

	$d \in 0..9$	$s \in \{+,-\}$	.
$\{q_0, q_1\}$	$\{q_1, q_4\}$	$\{q_1\}$	$\{q_2\}$
$\{q_1, q_4\}$	$\{q_1, q_4\}$	$\varnothing$	$\{q_2, q_3, q_5\}$
$\{q_1\}$	$\{q_1, q_4\}$	$\varnothing$	$\{q_2\}$
$\{q_2\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$
$\{q_2, q_3, q_5\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$
$\{q_3, q_5\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$

For example, $\delta(\{q_0, q_1\}, d)$ is calculated as follows: $[d \in 0..9]$

$$\begin{aligned}
& \bigcup\{\texttt{ECLOSE}(q) \mid q \in \delta(q_0, d) \cup \delta(q_1, d)\} \\
= \;& \bigcup\{\texttt{ECLOSE}(q) \mid q \in \varnothing \cup \{q_1, q_4\}\} \\
= \;& \bigcup\{\texttt{ECLOSE}(q) \mid q \in \{q_1, q_4\}\} \\
= \;& \texttt{ECLOSE}(q_1) \cup \texttt{ECLOSE}(q_4) \\
= \;& \{q_1\} \cup \{q_4\} \\
= \;& \{q_1, q_4\}
\end{aligned}$$

DFA $\equiv$ ϵ-NFA: Extended Subset Const. (2)

Given an ϵ=*NFA* $N = (Q_N, \Sigma_N, \delta_N, q_0, F_N)$, by applying the ***extended subset construction*** to it, the resulting *DFA* $D = (Q_D, \Sigma_D, \delta_D, q_{D_{start}}, F_D)$ is such that:

$$\begin{aligned}
\Sigma_D &= \Sigma_N \\
q_{D_{start}} &= \texttt{ECLOSE}(q_0) \\
F_D &= \{\ S \mid S \subseteq Q_N \wedge S \cap F_N \neq \varnothing\ \} \\
Q_D &= \{\ S \mid S \subseteq Q_N \wedge (\exists w \bullet w \in \Sigma^* \Rightarrow S = \hat{\delta}_N(q_0, w))\ \} \\
\delta_D(S, a) &= \bigcup\{\ \texttt{ECLOSE}(s') \mid s \in S \wedge s' \in \delta_N(s, a)\ \}
\end{aligned}$$

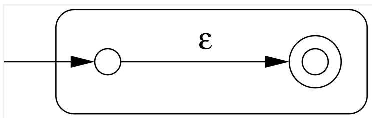
Regular Expression to ϵ-NFA

- Just as we construct each complex *regular expression* recursively, we define its equivalent ϵ-*NFA* *recursively* .
- Given a regular expression R, we construct an ϵ-NFA E, such that $L(R) = L(E)$, with
  - Exactly **one** accept state.
  - No incoming arc to the start state.
  - No outgoing arc from the accept state.

Regular Expression to ϵ-NFA

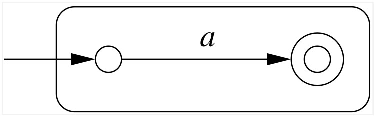
Base Cases:

- ϵ



- $\varnothing$

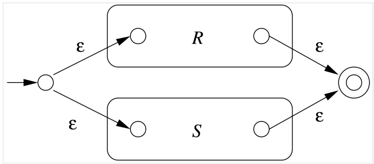
- a $[a \in \Sigma]$



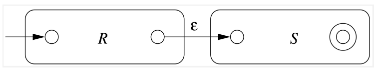
Regular Expression to ϵ-NFA

Recursive Cases: [R and S are RE's]

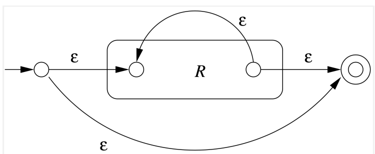
- $R + S$



- RS

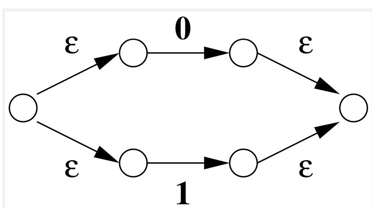


- R^*

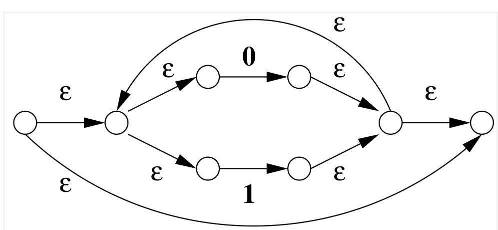


Regular Expression to ϵ-NFA: Examples (1.1)

- $0 + 1$

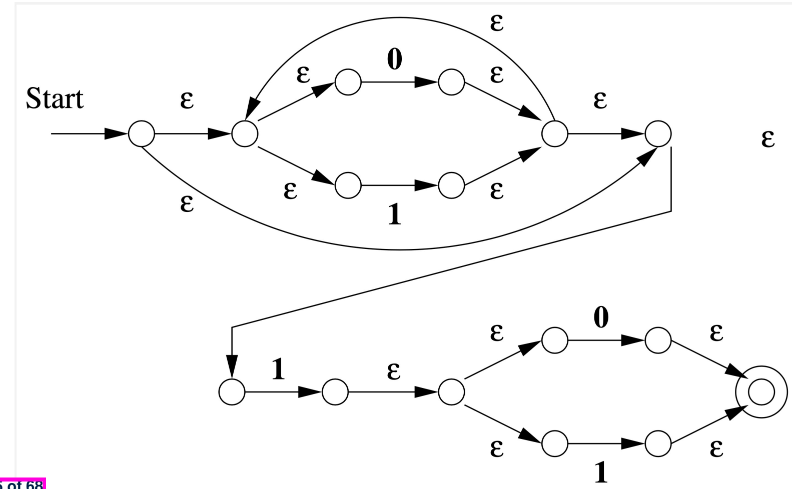


- $(0 + 1)^*$



Regular Expression to ϵ-NFA: Examples (1.2)

- $(0 + 1)^* 1 (0 + 1)$



Minimizing DFA: Motivation

- Recall: $\boxed{\text{Regular Expresion}} \longrightarrow \boxed{\epsilon\text{-NFA}} \longrightarrow \boxed{\text{DFA}}$
- DFA produced by the ***extended subset construction*** (with ***lazy evaluation***) may **not** be ***minimum*** on its size of state.
- When the required size of memory is sensitive
  (e.g., processor's cache memory),
  the fewer number of DFA states, the better.

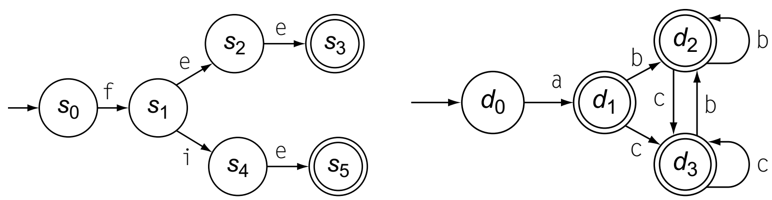
Minimizing DFA: Algorithm

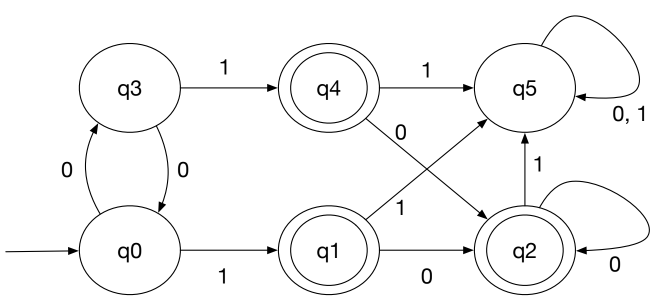
```
ALGORITHM: MinimizeDFAStates
  INPUT: DFA M = (Q, Σ, δ, q0, F)
  OUTPUT: M′ s.t. minimum |Q| and equivalent behaviour as M
PROCEDURE:
  P := ∅ /* refined partition so far */
  T := { F, Q − F } /* last refined partition */
  while (P ≠ T):
    P := T
    T := ∅
    for(p ∈ P):
      find the maximal S ⊂ p s.t. splittable(p, S)
      if S ≠ ∅ then
        T := T ∪ {S, p − S}
      else
        T := T ∪ {p}
      end
```

splittable(p, S) holds iff there is $c \in \Sigma$ s.t.

1. $S \subset p$ (or equivalently: $p - S \neq \varnothing$)
2. Transitions via c lead all $s \in S$ to states in ***same partition*** $p1$ ($p1 \neq p$).

Minimizing DFA: Examples





Exercises: Minimize the DFA from here; Q1 & Q2, p59, EAC2.

Exercise: Regular Expression to Minimized DFA

Given regular expression `r[0..9]+` which specifies the pattern of a register name, derive the equivalent DFA with the minimum number of states. Show all steps.

Implementing DFA as Scanner

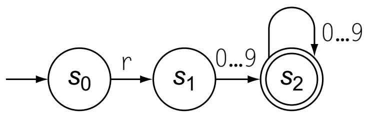
- The source language has a list of ***syntactic categories***:
  - e.g., keyword `while` — `[while]`
  - e.g., identifiers — `[[a-zA-Z][a-zA-Z0-9_]*]`
  - e.g., white spaces — `[[ \t\r]+]`
- A compiler's *scanner* must recognize ***words*** from **all** syntactic categories of the source language.
  - Each syntactic category is specified via a ***regular expression***.

$$\underbrace{r_1}_{\texttt{syn. cat. 1}} + \underbrace{r_1}_{\texttt{syn. cat. 2}} + \ldots + \underbrace{r_n}_{\texttt{syn. cat. } n}$$

  - Overall, a scanner should be implemented based on the ***minimized DFA*** accommodating all syntactic categories.
- Principles of a scanner:
  - Returns one ***word*** at a time
  - Each returned word is the ***longest possible*** that matches a ***pattern***
  - A **priority** may be specified among patterns (e.g., `new` is a keyword, not identifier)

Implementing DFA: Table-Driven Scanner (1)

- Consider the ***syntactic category*** of `register` names.
- Specified as a *regular expression* : `r[0..9]+`
- Afer conversion to ϵ-NFA, then to DFA, then to ***minimized DFA***:



- The following tables encode knowledge about the above DFA:

Classifier (CharCat)

r	0, 1, 2, ..., 9	EOF	**Other**
Register	*Digit*	*Other*	*Other*

Transition (δ)

	Register	***Digit***	***Other***
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Token Type (Type)

s_0	s_1	s_2	s_e
invalid	*invalid*	*register*	*invalid*

Implementing DFA: Table-Driven Scanner (2)

The scanner then is implemented via a 4-stage skeleton:

```
NextWord()
  -- Stage 1:  Initialization
  state := s0 ; word := ϵ
  initialize an empty stack s ; s.push(bad)
  -- Stage 2:  Scanning Loop
  while (state ≠ se)
    NextChar(char) ; word := word + char
    if state ∈ F then reset stack s end
    s.push(state)
    cat := CharCat[char]
    state := δ[state, cat]
  -- Stage 3:  Rollback Loop
  while (state ∉ F ∧ state ≠ bad)
    state := s.pop()
    truncate word
  -- Stage 4:  Interpret and Report
  if state ∈ F then return Type[state]
  else return invalid
  end
```


Index (1)

Index (2)

Index (3)

Index (4)

Index (5)

Index (6)

Parser: Syntactic Analysis

Readings: EAC2 Chapter 3

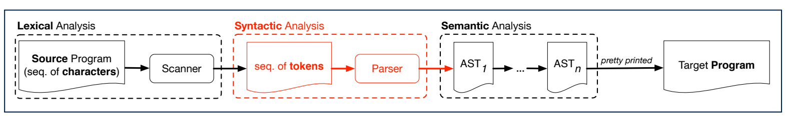
YORK UNIVERSITÉ UNIVERSITY

EECS4302 A:
Compilers and Interpreters
Fall 2022

CHEN-WEI WANG

Parser in Context

- Recall:



- Treats the input programas as a ***a sequence of classified tokens/words***
- Applies rules ***parsing*** token sequences as ***abstract syntax trees (ASTs)*** [***syntactic*** analysis]
- Upon termination:
  - Reports token sequences not derivable as ASTs
  - Produces an ***AST***
- No longer considers ***every character*** in input program.
- *Derivable* token sequences constitute a *context-free language (CFL)*.

Context-Free Languages: Introduction

- We have seen ***regular languages***:
  - Can be described using ***finite automata*** or ***regular expressions***.
  - Satisfy the ***pumping lemma***.
- Language with ***recursive*** structures are provably ***non-regular***.
  e.g., $\{0^n 1^n \mid n \geq 0\}$
- *Context-Free Grammars (CFG's)* are used to describe strings that can be generated in a ***recursive*** fashion.
- *Context-Free Languages (CFL's)* are:
  - Languages that can be described using CFG's.
  - A proper superset of the set of regular languages.

CFG: Example (1.1)

- The following language that is ***non-regular***

$$\{0^n \# 1^n \mid n \geq 0\}$$

can be described using a ***context-free grammar (CFG)***:

$$\begin{array}{rcl} A & \rightarrow & 0A1 \\ A & \rightarrow & B \\ B & \rightarrow & \# \end{array}$$

- A grammar contains a collection of ***substitution*** or ***production*** rules, where:
  - A **terminal** is a word $w \in \Sigma^*$ (e.g., 0, 1, *etc.*).
  - A ***variable*** or ***non-terminal*** is a word $w \notin \Sigma^*$ (e.g., A, B, *etc.*).
  - A ***start variable*** occurs on the LHS of the topmost rule (e.g., A).

CFG: Example (1.2)

- Given a grammar, generate a string by:
  1. Write down the ***start variable***.
  2. Choose a production rule where the ***start variable*** appears on the LHS of the arrow, and ***substitute*** it by the RHS.
  3. There are two cases of the re-written string:
     - 3.1 It contains **no** variables, then you are done.
     - 3.2 It contains **some** variables, then ***substitute*** each variable using the relevant ***production rules***.
  4. Repeat Step 3.
- e.g., We can generate an infinite number of strings from

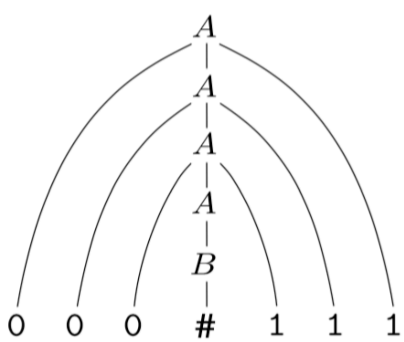
$$\begin{array}{lll} A & \rightarrow & 0A1 \\ A & \rightarrow & B \\ B & \rightarrow & \# \end{array}$$

  - $A \Rightarrow B \Rightarrow \#$
  - $A \Rightarrow 0A1 \Rightarrow 0B1 \Rightarrow 0\#1$
  - $A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 00B11 \Rightarrow 00\#11$
  - ...

CFG: Example (1.2)

Given a CFG, a string's ***derivation*** can be shown as a ***parse tree***.

e.g., The derivation of $000\#111$ has the parse tree



CFG: Example (2)

Design a CFG for the following language:

$$\{w \mid w \in \{0,1\}^* \wedge w \texttt{ is a palidrome}\}$$

e.g., 00, 11, 0110, 1001, *etc.*

$$\begin{array}{lll} P & \rightarrow & \epsilon \\ P & \rightarrow & 0 \\ P & \rightarrow & 1 \\ P & \rightarrow & 0P0 \\ P & \rightarrow & 1P1 \end{array}$$

CFG: Example (3)

Design a CFG for the following language:

$$\{ww^R \mid w \in \{0,1\}^*\}$$

e.g., 00, 11, 0110, *etc.*

$$\begin{array}{lll} P & \rightarrow & \epsilon \\ P & \rightarrow & 0P0 \\ P & \rightarrow & 1P1 \end{array}$$

CFG: Example (4)

Design a CFG for the set of binary strings, where each block of 0's followed by at least as many 1's.

e.g., 000111, 0001111, *etc.*

- We use S to represent one such string, and A to represent each such block in S.

$$
\begin{array}{lcll}
S & \rightarrow & \epsilon & \{\texttt{BC of } S\} \\
S & \rightarrow & AS & \{\texttt{RC of } S\} \\
A & \rightarrow & \epsilon & \{\texttt{BC of } A\} \\
A & \rightarrow & 01 & \{\texttt{BC of } A\} \\
A & \rightarrow & 0A1 & \{\texttt{RC of } A\texttt{: equal 0's and 1's}\} \\
A & \rightarrow & A1 & \{\texttt{RC of } A\texttt{: more 1's}\}
\end{array}
$$

CFG: Example (5.1) Version 1

Design the grammar for the following small expression language, which supports:

- Arithmetic operations: `+`, `-`, `*`, `/`
- Relational operations: `>`, `<`, `>=`, `<=`, `==`, `/=`
- Logical operations: `true`, `false`, `!`, `&&`, `||`, `=>`

  Start with the variable ***Expression***.
- There are two possible versions:
  1. All operations are mixed together.
  2. Relevant operations are grouped together.

     Try both!

CFG: Example (5.2) Version 1

$$
\begin{array}{lcl}
\textit{Expression} & \rightarrow & \textit{IntegerConstant} \\
 & | & -\textit{IntegerConstant} \\
 & | & \textit{BooleanConstant} \\
 & | & \textit{BinaryOp} \\
 & | & \textit{UnaryOp} \\
 & | & (\ \textit{Expression}\ ) \\
\\
\textit{IntegerConstant} & \rightarrow & \textit{Digit} \\
 & | & \textit{Digit IntegerConstant} \\
\\
\textit{Digit} & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\\
\textit{BooleanConstant} & \rightarrow & \texttt{TRUE} \\
 & | & \texttt{FALSE}
\end{array}
$$

CFG: Example (5.3) Version 1

$$
\begin{array}{lcl}
\textit{BinaryOp} & \rightarrow & \textit{Expression} + \textit{Expression} \\
 & | & \textit{Expression} - \textit{Expression} \\
 & | & \textit{Expression} * \textit{Expression} \\
 & | & \textit{Expression} \ / \ \textit{Expression} \\
 & | & \textit{Expression} \ \&\& \ \textit{Expression} \\
 & | & \textit{Expression} \ || \ \textit{Expression} \\
 & | & \textit{Expression} \Rightarrow \textit{Expression} \\
 & | & \textit{Expression} == \textit{Expression} \\
 & | & \textit{Expression} \ /= \ \textit{Expression} \\
 & | & \textit{Expression} > \textit{Expression} \\
 & | & \textit{Expression} < \textit{Expression} \\
\\
\textit{UnaryOp} & \rightarrow & !\ \textit{Expression}
\end{array}
$$

CFG: Example (5.4) Version 1

However, Version 1 of CFG:

- ***Parses*** string that requires further ***semantic analysis*** (e.g., type checking):
  e.g., `3 => 6`
- Is ***ambiguous***, meaning?
  - Some string may have more than one ways to interpreting it.
  - An interpretation is either visualized as a ***parse tree***, or written as a sequence of ***derivations***.

  e.g., Draw the parse tree(s) for `3 * 5 + 4`

CFG: Example (5.5) Version 2

Expression	→	*ArithmeticOp*
	|	*RelationalOp*
	|	*LogicalOp*
	|	`(` *Expression* `)`
IntegerConstant	→	*Digit*
	|	*Digit IntegerConstant*
Digit	→	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
BooleanConstant	→	`TRUE`
	|	`FALSE`

CFG: Example (5.6) Version 2

ArithmeticOp	→	*ArithmeticOp* `+` *ArithmeticOp*
	|	*ArithmeticOp* `-` *ArithmeticOp*
	|	*ArithmeticOp* `*` *ArithmeticOp*
	|	*ArithmeticOp* `/` *ArithmeticOp*
	|	`(`*ArithmeticOp*`)`
	|	*IntegerConstant*
	|	`-`*IntegerConstant*
RelationalOp	→	*ArithmeticOp* `==` *ArithmeticOp*
	|	*ArithmeticOp* `/=` *ArithmeticOp*
	|	*ArithmeticOp* `>` *ArithmeticOp*
	|	*ArithmeticOp* `<` *ArithmeticOp*
LogicalOp	→	*LogicalOp* `&&` *LogicalOp*
	|	*LogicalOp* `||` *LogicalOp*
	|	*LogicalOp* `=>` *LogicalOp*
	|	`!` *LogicalOp*
	|	`(`*LogicalOp*`)`
	|	*RelationalOp*
	|	*BooleanConstant*

CFG: Example (5.7) Version 2

However, Version 2 of CFG:

- Eliminates some cases for further semantic analysis:
  e.g., `(1 + 2) => (5 / 4)` [no parse tree]
- Still ***parses*** strings that might require further ***semantic analysis***:
  e.g., `(1 + 2) / (5 - (2 + 3))`
- Still is ***ambiguous***.
  e.g., Draw the parse tree(s) for `3 * 5 + 4`

CFG: Formal Definition (1)

- A ***context-free grammar (CFG)*** is a 4-tuple (V, Σ, R, S):
  - V is a finite set of **variables**.
  - Σ is a finite set of ***terminals***. $[V \cap \Sigma = \varnothing]$
  - R is a finite set of ***rules*** s.t.

$$R \subseteq \{v \rightarrow s \mid v \in V \wedge s \in (V \cup \Sigma)^*\}$$

  - $S \in V$ is is the **start variable**.
- Given strings $u, v, w \in (V \cup \Sigma)^*$, variable $A \in V$, a rule $A \rightarrow w$:
  - $\boxed{uAv \Rightarrow uwv}$ menas that uAv ***yields*** uwv.
  - $\boxed{u \overset{*}{\Rightarrow} v}$ means that u ***derives*** v, if:
    - $u = v$; or
    - $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$ [a ***yield sequence***]
- Given a CFG $G = (V, \Sigma, R, S)$, the language of G

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$$

CFG: Formal Definition (2): Example

- Design the ***CFG*** for strings of properly-nested parentheses.

  e.g., (), ()(), ((()()))(), *etc.*

  Present your answer in a ***formal*** manner.
- $G = (\{S\}, \{(,)\}, R, S)$, where R is

$$S \rightarrow (\ S\ ) \mid SS \mid \epsilon$$

- Draw ***parse trees*** for the above three strings that G generates.

CFG: Formal Definition (3): Example

- Consider the grammar $G = (V, \Sigma, R, S)$:
  - R is

$$\begin{array}{lcl} Expr & \rightarrow & Expr + Term \\ & \mid & Term \\ Term & \rightarrow & Term * Factor \\ & \mid & Factor \\ Factor & \rightarrow & (Expr) \\ & \mid & \mathtt{a} \end{array}$$

  - $V = \{Expr, Term, Factor\}$
  - $\Sigma = \{\mathtt{a}, +, *, (,)\}$
  - $S = Expr$
- ***Precedence*** of operators $+$, $*$ is embedded in the grammar.
  - "Plus" is specified at a **higher** level (*Expr*) than is "times" (*Term*).
  - Both operands of a multiplication (*Factor*) may be **parenthesized**.

Regular Expressions to CFG's

- Recall the semantics of regular expressions (assuming that we do not consider $\varnothing$):

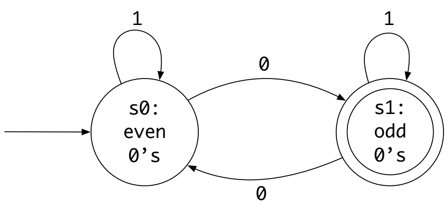
$$\begin{array}{lcl} L(\ \epsilon\ ) & = & \{\epsilon\} \\ L(\ a\ ) & = & \{a\} \\ L(\ E + F\ ) & = & L(E) \cup L(F) \\ L(\ EF\ ) & = & L(E)L(F) \\ L(\ E^*\ ) & = & (L(E))^* \\ L(\ (E)\ ) & = & L(E) \end{array}$$

- e.g., Grammar for $(00 + 1)^* + (11 + 0)^*$

$$\begin{array}{lcl} S & \rightarrow & A \mid B \\ A & \rightarrow & \epsilon \mid AC \\ C & \rightarrow & 00 \mid 1 \\ B & \rightarrow & \epsilon \mid BD \\ D & \rightarrow & 11 \mid 0 \end{array}$$

DFA to CFG's

- Given a DFA $M = (Q, \Sigma, \delta, q_0, F)$:
  - Make a ***variable*** R_i for each ***state*** $q_i \in Q$.
  - Make R_0 the ***start variable***, where q_0 is the ***start state*** of M.
  - Add a rule $R_i \rightarrow aR_j$ to the grammar if $\delta(q_i, a) = q_j$.
  - Add a rule $R_i \rightarrow \epsilon$ if $q_i \in F$.
- e.g., Grammar for



$$
\begin{array}{lcl}
R_0 & \rightarrow & 1R_0 \mid 0R_1 \\
R_1 & \rightarrow & 0R_0 \mid 1R_1 \mid \epsilon
\end{array}
$$

CFG: Leftmost Derivations (1)

$$
\begin{array}{lcl}
Expr & \rightarrow & Expr + Term \mid Term \\
Term & \rightarrow & Term * Factor \mid Factor \\
Factor & \rightarrow & (Expr) \mid a
\end{array}
$$

- Given a string ($\in (V \cup \Sigma)^*$), a ***left-most derivation (LMD)*** keeps substituting the leftmost non-terminal ($\in V$).
- ***Unique LMD*** for the string `a + a * a`:

$$
\begin{array}{lcl}
Expr & \Rightarrow & Expr + Term \\
 & \Rightarrow & Term + Term \\
 & \Rightarrow & Factor + Term \\
 & \Rightarrow & a + Term \\
 & \Rightarrow & a + Term * Factor \\
 & \Rightarrow & a + Factor * Factor \\
 & \Rightarrow & a + a * Factor \\
 & \Rightarrow & a + a * a
\end{array}
$$

- This ***LMD*** suggests that `a * a` is the right operand of `+`.

CFG: Rightmost Derivations (1)

$$
\begin{array}{lcl}
Expr & \rightarrow & Expr + Term \mid Term \\
Term & \rightarrow & Term * Factor \mid Factor \\
Factor & \rightarrow & (Expr) \mid a
\end{array}
$$

- Given a string ($\in (V \cup \Sigma)^*$), a ***right-most derivation (RMD)*** keeps substituting the rightmost non-terminal ($\in V$).
- ***Unique RMD*** for the string `a + a * a`:

$$
\begin{array}{lcl}
Expr & \Rightarrow & Expr + Term \\
 & \Rightarrow & Expr + Term * Factor \\
 & \Rightarrow & Expr + Term * a \\
 & \Rightarrow & Expr + Factor * a \\
 & \Rightarrow & Expr + a * a \\
 & \Rightarrow & Term + a * a \\
 & \Rightarrow & Factor + a * a \\
 & \Rightarrow & a + a * a
\end{array}
$$

- This ***RMD*** suggests that `a * a` is the right operand of `+`.

CFG: Leftmost Derivations (2)

$$
\begin{array}{lcl}
Expr & \rightarrow & Expr + Term \mid Term \\
Term & \rightarrow & Term * Factor \mid Factor \\
Factor & \rightarrow & (Expr) \mid a
\end{array}
$$

- ***Unique LMD*** for the string `(a + a) * a`:

$$
\begin{array}{lcl}
Expr & \Rightarrow & Term \\
 & \Rightarrow & Term * Factor \\
 & \Rightarrow & Factor * Factor \\
 & \Rightarrow & (Expr) * Factor \\
 & \Rightarrow & (Expr + Term) * Factor \\
 & \Rightarrow & (Term + Term) * Factor \\
 & \Rightarrow & (Factor + Term) * Factor \\
 & \Rightarrow & (a + Term) * Factor \\
 & \Rightarrow & (a + Factor) * Factor \\
 & \Rightarrow & (a + a) * Factor \\
 & \Rightarrow & (a + a) * a
\end{array}
$$

- This ***LMD*** suggests that `(a + a)` is the left operand of `*`.

CFG: Rightmost Derivations (2)

$$\begin{array}{lcl} Expr & \rightarrow & Expr\ +\ Term \mid Term \\ Term & \rightarrow & Term\ *\ Factor \mid Factor \\ Factor & \rightarrow & (\ Expr\ ) \mid a \end{array}$$

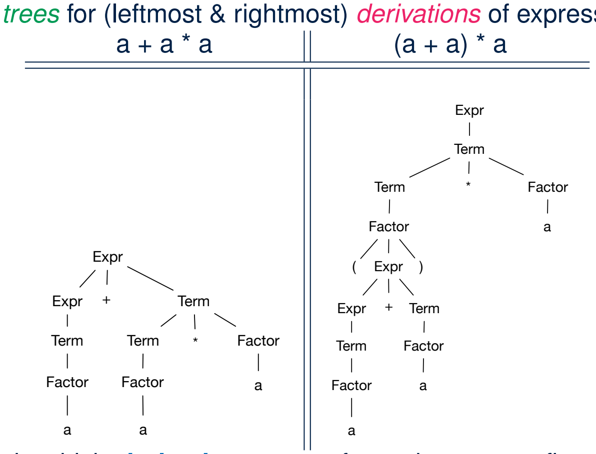
- ***Unique RMD*** for the string `(a + a) * a`:

$$\begin{array}{lcl} Expr & \Rightarrow & Term \\ & \Rightarrow & Term\ *\ Factor \\ & \Rightarrow & Term\ *\ a \\ & \Rightarrow & Factor\ *\ a \\ & \Rightarrow & (\ Expr\ )\ *\ a \\ & \Rightarrow & (\ Expr\ +\ Term\ )\ *\ a \\ & \Rightarrow & (\ Expr\ +\ Factor\ )\ *\ a \\ & \Rightarrow & (\ Expr\ +\ a\ )\ *\ a \\ & \Rightarrow & (\ Term\ +\ a\ )\ *\ a \\ & \Rightarrow & (\ Factor\ +\ a\ )\ *\ a \\ & \Rightarrow & (\ a\ +\ a\ )\ *\ a \end{array}$$

- This ***RMD*** suggests that `(a + a)` is the left operand of $*$.

CFG: Parse Trees vs. Derivations (1)

- *Parse trees* for (leftmost & rightmost) *derivations* of expressions:



- Orders in which ***derivations*** are performed are ***not*** reflected on parse trees.

CFG: Parse Trees vs. Derivations (2)

- A string $w \in \Sigma^*$ may have more than one ***derivations***.
  Q: distinct ***derivations*** for $w \in \Sigma^* \Rightarrow$ distinct ***parse trees*** for w?
  A: Not in general $\because$ Derivations with ***distinct orders*** of variable substitutions may still result in the ***same parse tree***.
- For example:

$$\begin{array}{lcl} Expr & \rightarrow & Expr\ +\ Term \mid Term \\ Term & \rightarrow & Term\ *\ Factor \mid Factor \\ Factor & \rightarrow & (\ Expr\ ) \mid a \end{array}$$

For string `a + a * a`, the ***LMD*** and ***RMD*** have ***distinct orders*** of variable substitutions, but their corresponding ***parse trees are the same***.

CFG: Ambiguity: Definition

Given a grammar $G = (V, \Sigma, R, S)$:

- A string $w \in \Sigma^*$ is derived *ambiguously* in G if there exist
  two or more ***distinct parse trees*** or, equally,
  two or more ***distinct LMDs*** or, equally,
  two or more ***distinct RMDs***.

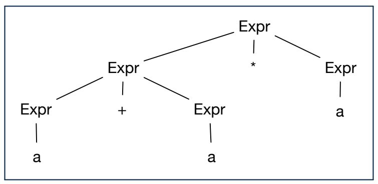
  We require that all such derivations are completed by following a consisten order (**leftmost** or **rightmost**) to avoid ***false positive***.
- G is *ambiguous* if it generates some string ambiguously.

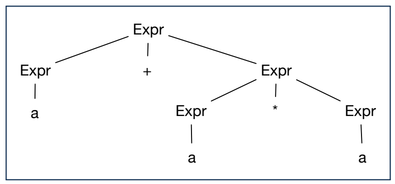
CFG: Ambiguity: Exercise (1)

- Is the following grammar *ambiguous* ?

$$Expr \rightarrow Expr \ \texttt{+} \ Expr \mid Expr \ \texttt{*} \ Expr \mid \texttt{(} \ Expr \ \texttt{)} \mid a$$

- Yes ∵ it generates the string `a + a * a` *ambiguously* :





- ***Distinct ASTs*** (for the ***same input***) imply ***distinct semantic interpretations***: e.g., a pre-order traversal for evaluation
- **Exercise**: Show ***LMDs*** for the two parse trees.

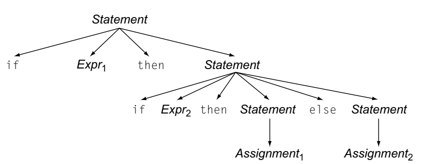
CFG: Ambiguity: Exercise (2.1)

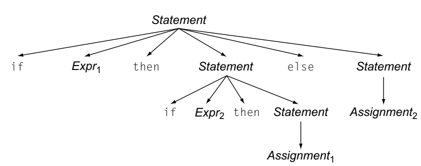
- Is the following grammar *ambiguous* ?

$$\begin{array}{lcl} Statement & \rightarrow & \texttt{if} \ Expr \ \texttt{then} \ Statement \\ & \mid & \texttt{if} \ Expr \ \texttt{then} \ Statement \ \texttt{else} \ Statement \\ & \mid & Assignment \\ & \dots & \end{array}$$

- Yes ∵ it derives the following string *ambiguously* :

$$\texttt{if} \ Expr_1 \ \texttt{then} \ \texttt{if} \ Expr_2 \ \texttt{then} \ Assignment_1 \ \texttt{else} \ Assignment_2$$

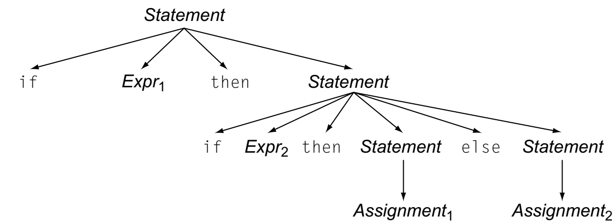




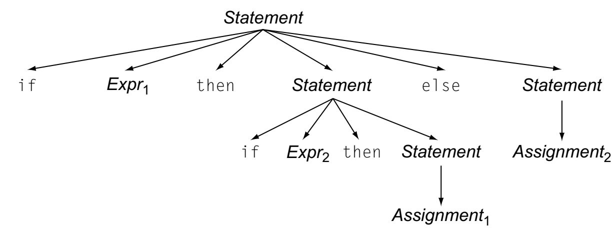
- This is called the ***dangling else*** problem.
- **Exercise:** Show ***LMDs*** for the two parse trees.

CFG: Ambiguity: Exercise (2.2)

(***Meaning 1***) *Assignment*$_2$ may be associated with the inner `if`:



(***Meaning 2***) *Assignment*$_2$ may be associated with the outer `if`:



CFG: Ambiguity: Exercise (2.3)

- We may remove the ***ambiguity*** by specifying that the *dangling* `else` is associated with the **nearest** `if`:

$$\begin{array}{lcl} Statement & \rightarrow & \texttt{if} \ Expr \ \texttt{then} \ Statement \\ & \mid & \texttt{if} \ Expr \ \texttt{then} \ WithElse \ \texttt{else} \ Statement \\ & \mid & Assignment \\ WithElse & \rightarrow & \texttt{if} \ Expr \ \texttt{then} \ WithElse \ \texttt{else} \ WithElse \\ & \mid & Assignment \end{array}$$

- When applying $\boxed{\texttt{if} \ \dots \ \texttt{then} \ WithElse \ \texttt{else} \ Statement}$:
  - The ***true*** branch will be produced via *WithElse*.
  - The ***false*** branch will be produced via *Statement*.

  There is ***no circularity*** between the two non-terminals.

Discovering Derivations

- Given a CFG $G = (V, \Sigma, R, S)$ and an input program $p \in \Sigma^*$:
  - So far we **manually** come up a valid ***derivation*** s.t. $S \stackrel{*}{\Rightarrow} p$.
  - A *parser* is supposed to **automate** this ***derivation*** process.
    - Input: **A sequence of (t, c) pairs**, where each ***token*** t (e.g., `r241`) belongs to a ***syntactic category*** c (e.g., `register`); and a ***CFG*** G.
    - Output: A ***valid derivation*** (as an ***AST***); or A ***parse error***.
- In the process of constructing an ***AST*** for the input program:
  - ***Root*** of AST: The ***start symbol*** S of G
  - ***Internal nodes***: A ***subset of variables*** V of G
  - ***Leaves*** of AST: A ***token/terminal*** sequence

  $\Rightarrow$ Discovering the *grammatical connections* (w.r.t. R of G) between the ***root***, ***internal nodes***, and ***leaves*** is the hard part!
- Approaches to Parsing: $[\, w \in (V \cup \Sigma)^*,\ A \in V,\ \boxed{A \rightarrow w} \in R \,]$
  - **Top-down** parsing

    For a node representing A, extend it with a subtree representing w.
  - **Bottom-up** parsing

    For a substring matching w, build a node representing A accordingly.

TDP: Discovering Leftmost Derivation

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R then
        create β1, β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
```

backtrack $\triangleq$ pop *focus*.siblings; *focus* := *focus*.parent; *focus*.resetChildren

TDP: Exercise (1)

- Given the following CFG **G**:

$$
\begin{array}{lcl}
Expr & \rightarrow & Expr \ \texttt{+} \ Term \\
 & | & Term \\
Term & \rightarrow & Term \ \star \ Factor \\
 & | & Factor \\
Factor & \rightarrow & \texttt{(} Expr \texttt{)} \\
 & | & \texttt{a}
\end{array}
$$

  Trace *TDParse* on how to build an AST for input `a + a * a`.
- Running *TDParse* with **G** results an ***infinite loop***!!!
  - *TDParse* focuses on the ***leftmost*** non-terminal.
  - The grammar **G** contains ***left-recursions***.
- We must first convert left-recursions in **G** to ***right-recursions***.

TDP: Exercise (2)

- Given the following CFG **G**:

$$
\begin{array}{lcl}
Expr & \rightarrow & Term \ \ Expr' \\
Expr' & \rightarrow & \texttt{+} \ Term \ \ Expr' \\
 & | & \epsilon \\
Term & \rightarrow & Factor \ \ Term' \\
Term' & \rightarrow & \star \ Factor \ \ Term' \\
 & | & \epsilon \\
Factor & \rightarrow & \texttt{(} Expr \texttt{)} \\
 & | & \texttt{a}
\end{array}
$$

  Exercise. Trace *TDParse* on building AST for `a + a * a`.

  Exercise. Trace *TDParse* on building AST for `(a + a) * a`.

  Q: How to handle ϵ-productions (e.g., $Expr \rightarrow \epsilon$)?

  A: Execute *focus* := *trace*.`pop()` to advance to next node.
- Running *TDParse* will ***terminate*** $\because$ **G** is ***right-recursive***.
- We will learn about a systematic approach to converting left-recursions in a given grammar to ***right-recursions***.

Left-Recursions (LR): Direct vs. Indirect

Given CFG $G = (V, \Sigma, R, S)$, $\alpha, \beta, \gamma \in (V \cup \Sigma)^*$, G contains:

- A *cycle* if $\exists A \in V \bullet A \stackrel{*}{\Rightarrow} A$
- A ***direct*** LR if $A \to A\alpha \in R$ for non-terminal $A \in V$

e.g.,

$$
\begin{array}{lcl}
Expr & \to & Expr + Term \\
 & | & Term \\
Term & \to & Term * Factor \\
 & | & Factor \\
Factor & \to & (Expr) \\
 & | & \mathtt{a}
\end{array}
$$

e.g.,

$$
\begin{array}{lcl}
Expr & \to & Expr + Term \\
 & | & Expr - Term \\
 & | & Term \\
Term & \to & Term * Factor \\
 & | & Term / Factor \\
 & | & Factor
\end{array}
$$

- An ***indirect*** LR if $A \to B\beta \in R$ for non-terminals $A, B \in V$, $B \stackrel{*}{\Rightarrow} A\gamma$

$$
\begin{array}{lcl}
A & \to & Br \\
B & \to & Cd \\
C & \to & At
\end{array}
$$

$A \to Br$, $B \stackrel{*}{\Rightarrow} Atd$

$$
\begin{array}{lcllll}
A & \to & Ba & | & b & \\
B & \to & Cd & | & e & \\
C & \to & Df & | & g & \\
D & \to & f & | & Aa & | \; Cg
\end{array}
$$

$A \to Ba$, $B \stackrel{*}{\Rightarrow} Aafd$

TDP: (Preventively) Eliminating LRs

```
1  ALGORITHM: RemoveLR
2    INPUT: CFG G = (V, Σ, R, S)
3    ASSUME: G has no ε-productions
4    OUTPUT: G′ s.t. G′ ≡ G, G′ has no
5            indirect & direct left-recursions
6  PROCEDURE:
7    impose an order on V: ⟨(A1, A2, ..., An)⟩
8    for i: 1 .. n:
9      for j: 1 .. i − 1:
10       if ∃ Ai → Aj γ ∈ R ∧ Aj → δ1 | δ2 | ... | δm ∈ R then
11         replace Ai → Aj γ with Ai → δ1 γ | δ2 γ | ... | δm γ
12       end
13     for Ai → Ai α | β ∈ R:
14       replace it with: Ai → β Ai′, Ai′ → α Ai′ | ε
```

- **L9** to **L12**: Remove ***indirect*** left-recursions from A_1 to A_{i-1}.
- **L13** to **L14**: Remove ***direct*** left-recursions from A_1 to A_{i-1}.
- ***Loop Invariant*** **(outer for-loop)**? At the start of i^{th} iteration:
  - No ***direct*** or ***indirect*** left-recursions for $A_1, A_2, \ldots, A_{i-1}$.
  - More precisely: $\forall j : j < i \bullet \neg(\exists k \bullet k \le j \land A_j \to A_k \cdots \in R)$

CFG: Eliminating ϵ-Productions (1)

- Motivations:
  - ***TDParse*** handles each ϵ-production as a special case.
  - ***RemoveLR*** produces CFG which may contain ϵ-productions.
- $\epsilon \notin L \Rightarrow \exists$ CFG $G = (V, \Sigma, R, S)$ s.t. G has no ϵ-productions.

  An *ϵ-production* has the form $A \to \epsilon$.
- A variable A is ***nullable*** if $A \stackrel{*}{\Rightarrow} \epsilon$.
  - Each terminal symbol is ***not nullable***.
  - Variable A is ***nullable*** if either:
    - $A \to \epsilon \in R$; or
    - $A \to B_1 B_2 \ldots B_k \in R$, where each variable B_i $(1 \le i \le k)$ is a ***nullable***.
- Given a production $B \to CAD$, if only variable A is ***nullable***, then there are 2 versions of B: $B \to CAD \mid CD$
- In general, given a production $A \to X_1 X_2 \ldots X_k$ with k symbols, if m of the k symbols are ***nullable***:
  - $m < k$: There are 2^m versions of A.
  - $m = k$: There are $2^m - 1$ versions of A. [excluding $A \to \epsilon$]

CFG: Eliminating ϵ-Productions (2)

- Eliminate ϵ-productions from the following grammar:

$$
\begin{array}{lcl}
S & \to & AB \\
A & \to & aAA \mid \epsilon \\
B & \to & bBB \mid \epsilon
\end{array}
$$

- Which are the ***nullable*** variables? [S, A, B]

$$
\begin{array}{lcll}
S & \to & A \mid B \mid AB & \{S \to \epsilon \texttt{ not included}\} \\
A & \to & aAA \mid aA \mid a & \{A \to aA \texttt{ duplicated}\} \\
B & \to & bBB \mid bB \mid b & \{B \to bB \texttt{ duplicated}\}
\end{array}
$$

Backtrack-Free Parsing (1)

- TDParse automates the ***top-down***, ***leftmost*** derivation process by consistently choosing production rules (e.g., in order of their appearance in CFG).
  - This ***inflexibility*** may lead to ***inefficient*** runtime performance due to the need to *backtrack*.
  - e.g., It may take the ***construction of a giant subtree*** to find out a ***mismatch*** with the input tokens, which end up requiring it to *backtrack* all the way back to the ***root*** (start symbol).
- We may avoid backtracking with a modification to the parser:
  - When deciding which production rule to choose, consider:
    (1) the ***current*** input symbol
    (2) the consequential ***first*** symbol if a rule was applied for focus
    [*lookahead* symbol]
  - Using a ***one symbol lookhead***, w.r.t. a ***right-recursive*** CFG, each alternative for the ***leftmost nonterminal*** leads to a ***unique terminal***, allowing the parser to decide on a choice that prevents *backtracking*.
  - Such CFG is ***backtrack free*** with the ***lookhead*** of one symbol.
  - We also call such backtrack-free CFG a ***predictive grammar***.

The FIRST Set: Definition

- Say we write $T \subset \mathbb{P}(\Sigma^*)$ to denote the set of valid tokens recognizable by the scanner.
- **FIRST** $(\alpha) \triangleq$ set of symbols that can appear as the ***first word*** in some string derived from α.
- More precisely:

$$\text{FIRST}(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \in T \\ \{w \mid w \in \Sigma^* \land \alpha \overset{*}{\Rightarrow} w\beta \land \beta \in (V \cup \Sigma)^*\} & \text{if } \alpha \in V \end{cases}$$

The FIRST Set: Examples

- Consider this ***right***-recursive CFG:

0	*Goal*	$\rightarrow$	*Expr*		6	*Term′*	$\rightarrow$	x *Factor Term′*
1	*Expr*	$\rightarrow$	*Term Expr′*		7		|	÷ *Factor Term′*
2	*Expr′*	$\rightarrow$	+ *Term Expr′*		8		|	ϵ
3		|	- *Term Expr′*		9	*Factor*	$\rightarrow$	(*Expr*)
4		|	ϵ		10		|	num
5	*Term*	$\rightarrow$	*Factor Term′*		11		|	name

- Compute **FIRST** for each terminal (e.g., num, +, ():

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST	num	name	+	-	x	÷	(	)	eof	ϵ

- Compute **FIRST** for each non-terminal (e.g., *Expr*, *Term′*):

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

Computing the FIRST Set

$$\text{FIRST}(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \in T \\ \{w \mid w \in \Sigma^* \land \alpha \overset{*}{\Rightarrow} w\beta \land \beta \in (V \cup \Sigma)^*\} & \text{if } \alpha \in V \end{cases}$$

```
ALGORITHM: GetFirst
  INPUT: CFG G = (V, Σ, R, S)
    T ⊂ Σ* denotes valid terminals
  OUTPUT: FIRST : V ∪ T ∪ {ε, eof} ⟶ ℙ(T ∪ {ε, eof})
PROCEDURE:
  for α ∈ (T ∪ {eof, ε}): FIRST(α) := {α}
  for A ∈ V: FIRST(A) := ∅
  lastFirst := ∅
  while (lastFirst ≠ FIRST):
    lastFirst := FIRST
    for A → β1β2...βk ∈ R s.t. ∀βj : βj ∈ (T ∪ V):
      rhs := FIRST(β1) − {ε}
      for (i := 1; ε ∈ FIRST(βi) ∧ i < k; i++):
        rhs := rhs ∪ (FIRST(βi+1) − {ε})
      if i = k ∧ ε ∈ FIRST(βk) then
        rhs := rhs ∪ {ε}
      end
      FIRST(A) := FIRST(A) ∪ rhs
```

Computing the FIRST Set: Extension

- Recall: **FIRST** takes as input a token or a variable.

$$\boxed{\textbf{FIRST} : V \cup T \cup \{\epsilon, eof\} \longrightarrow \mathbb{P}(T \cup \{\epsilon, eof\})}$$

- The computation of variable ***rhs*** in algoritm `GetFirst` actually suggests an extended, overloaded version:

$$\boxed{\textbf{FIRST} : (V \cup T \cup \{\epsilon, eof\})^* \longrightarrow \mathbb{P}(T \cup \{\epsilon, eof\})}$$

  FIRST may also take as input a string $\beta_1 \beta_2 \dots \beta_n$ (RHS of rules).

- More precisely:

$$\textbf{FIRST}(\beta_1 \beta_2 \dots \beta_n) = \left\{ \textbf{FIRST}(\beta_1) \cup \textbf{FIRST}(\beta_2) \cup \dots \cup \textbf{FIRST}(\beta_{k-1}) \cup \textbf{FIRST}(\beta_k) \;\middle|\; \begin{array}{l} \forall i : 1 \le i < k \bullet \epsilon \in \textbf{FIRST}(\beta_i) \\ \wedge \\ \epsilon \notin \textbf{FIRST}(\beta_k) \end{array} \right\}$$

Note. β_k is the first symbol whose **FIRST** set does not contain ϵ.

Extended FIRST Set: Examples

Consider this ***right***-recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

e.g., **FIRST**(*Term Expr′*) = **FIRST**(*Term*) = { (, name, num }

e.g., **FIRST**(+ *Term Expr′*) = **FIRST**(+) = {+}

e.g., **FIRST**(- *Term Expr′*) = **FIRST**(-) = {-}

e.g., **FIRST**(ϵ) = {ϵ}

Is the FIRST Set Sufficient

- Consider the following three productions:

Expr′	→	+	*Term*	*Term′*	(1)
	|	-	*Term*	*Term′*	(2)
	|	ϵ			(3)

  In TDP, when the parser attempts to expand an *Expr′* node, it ***looks ahead with one symbol*** to decide on the choice of rule: **FIRST**(+) = {+}, **FIRST**(−) = {−}, and **FIRST**(ϵ) = {ϵ}.

  Q. When to choose rule (3) (causing ***focus := trace.pop()***)?

  A?. Choose rule (3) when *focus* ≠ **FIRST**(+) ∧ *focus* ≠ **FIRST**(−)?

  - ***Correct*** but ***inefficient*** in case of illegal input string: syntax error is only reported after possibly a long series of ***backtrack***.
  - Useful if parser knows which words can appear, after an application of the ϵ-production (rule (3)), as leadling symbols.

- **FOLLOW** $(v : V) \triangleq$ set of symbols that can appear to the immediate right of a string derived from v.

$$\textbf{FOLLOW}(v) = \{\boldsymbol{w} \mid w, x, y \in \Sigma^* \wedge v \overset{*}{\Rightarrow} \boldsymbol{x} \wedge S \overset{*}{\Rightarrow} \boldsymbol{xwy}\}$$

The FOLLOW Set: Examples

- Consider this ***right***-recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

- Compute **FOLLOW** for each non-terminal (e.g., *Expr*, *Term′*):

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FOLLOW	eof,)	eof,)	eof, +, -,)	eof, +, -,)	eof, +, -, x, ÷,)

Computing the FOLLOW Set

$$\textsc{Follow}(v) = \{ w \mid w, x, y \in \Sigma^* \wedge v \overset{*}{\Rightarrow} x \wedge S \overset{*}{\Rightarrow} xwy \}$$

```
ALGORITHM: GetFollow
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: FOLLOW : V ⟶ ℙ(T ∪ {eof})
PROCEDURE:
  for A ∈ V: FOLLOW(A) := ∅
  FOLLOW(S) := {eof}
  lastFollow := ∅
  while(lastFollow ≠ FOLLOW):
    lastFollow := FOLLOW
    for A → β1β2...βk ∈ R:
      trailer := FOLLOW(A)
      for i: k .. 1:
        if βi ∈ V then
          FOLLOW(βi) := FOLLOW(βi) ∪ trailer
          if ϵ ∈ FIRST(βi)
            then trailer := trailer ∪ (FIRST(βi) − ϵ)
            else trailer := FIRST(βi)
        else
          trailer := FIRST(βi)
```


Backtrack-Free Grammar

- A ***backtrack-free grammar*** (for a **top-down parser**), when expanding the ***focus internal node***, is always able to choose a unique rule with the ***one-symbol lookahead*** (or report a ***syntax error*** when no rule applies).
- To formulate this, we first define:

$$\textsc{Start}(A \to \beta) = \begin{cases} \textsc{First}(\beta) & \text{if } \epsilon \notin \textsc{First}(\beta) \\ \textsc{First}(\beta) \cup \textsc{Follow}(A) & \text{otherwise} \end{cases}$$

  $\textsc{First}(\beta)$ is the extended version where β may be $\beta_1 \beta_2 \dots \beta_n$
- A ***backtrack-free grammar*** has each of its productions $A \to \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_n$ satisfying:

$$\forall i, j : 1 \leq i, j \leq n \wedge i \neq j \bullet \textsc{Start}(\gamma_i) \cap \textsc{Start}(\gamma_j) = \varnothing$$

TDP: Lookahead with One Symbol

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1β2...βn ∈ R ∧ word ∈ START(β) then
        create β1, β2 ... βn as children of focus
        trace.push(βnβn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
```

backtrack ≜ pop *focus*.siblings; *focus* := *focus*.parent; *focus*.resetChildren

Backtrack-Free Grammar: Exercise

Is the following CFG ***backtrack free***?

11	*Factor*	→	`name`
12		|	`name` `[` *ArgList* `]`
13		|	`name` `(` *ArgList* `)`
15	*ArgList*	→	*Expr MoreArgs*
16	*MoreArgs*	→	`,` *Expr MoreArgs*
17		|	ϵ

- $\epsilon \notin \textsc{First}(\textit{Factor}) \Rightarrow \textsc{Start}(\textit{Factor}) = \textsc{First}(\textit{Factor})$
- $\textsc{First}(\textit{Factor} \to \texttt{name}) = \{\texttt{name}\}$
- $\textsc{First}(\textit{Factor} \to \texttt{name [}\textit{ArgList}\texttt{]}) = \{\texttt{name}\}$
- $\textsc{First}(\textit{Factor} \to \texttt{name (}\textit{ArgList}\texttt{)}) = \{\texttt{name}\}$

∴ The above grammar is ***not*** backtrack free.
⇒ To expand an AST node of *Factor*, with a ***lookahead*** of `name`, the parser has no basis to choose among rules 11, 12, and 13.

Backtrack-Free Grammar: Left-Factoring

- A CFG is not backtrack free if there exists a ***common prefix*** (name) among the RHS of ***multiple*** production rules.
- To make such a CFG ***backtrack-free***, we may transform it using *left factoring*: a process of extracting and isolating ***common prefixes*** in a set of production rules.
  - Identify a common prefix α:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_j$$

[each of $\gamma_1, \gamma_2, \dots, \gamma_j$ does not begin with α]

  - Rewrite that production rule as:

$$\begin{array}{lcl} A & \rightarrow & \alpha B \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_j \\ B & \rightarrow & \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{array}$$

  - New rule $B \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ may also contain ***common prefixes***.
  - Rewriting continues until no common prefixes are identified.

Left-Factoring: Exercise

- Use *left-factoring* to remove all ***common prefixes*** from the following grammar.

11	*Factor*	$\rightarrow$	name
12		|	name [*ArgList*]
13		|	name (*ArgList*)
15	*ArgList*	$\rightarrow$	*Expr MoreArgs*
16	*MoreArgs*	$\rightarrow$	, *Expr MoreArgs*
17		|	ϵ

- Identify common prefix name and rewrite rules 11, 12, and 13:

$$\begin{array}{lcl} Factor & \rightarrow & \texttt{name} \quad Arguments \\ Arguments & \rightarrow & [\quad ArgList \quad] \\ & \mid & (\quad ArgList \quad) \\ & \mid & \epsilon \end{array}$$

Any more ***common prefixes***? [No]

TDP: Terminating and Backtrack-Free

- Given an arbitrary CFG as input to a ***top-down parser***:
  - **Q.** How do we avoid a ***non-terminating*** parsing process?
    A. Convert left-recursions to right-recursion.
  - **Q.** How do we minimize the need of ***backtracking***?
    A. left-factoring & one-symbol lookahead using START
- ***Not*** every context-free language has a corresponding ***backtrack-free*** context-free grammar.

  Given a CFL l, the following is ***undecidable***:

$$\boxed{\exists cfg \mid L(cfg) = l \wedge isBacktrackFree(cfg)}$$

- Given a CFG $g = (V, \Sigma, R, S)$, whether or not g is ***backtrack-free*** is ***decidable***:

  For each $A \rightarrow \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_n \in R$:

$$\boxed{\forall i, j : 1 \le i, j \le n \wedge i \ne j \bullet \textsc{Start}(\gamma_i) \cap \textsc{Start}(\gamma_j) = \varnothing}$$

Backtrack-Free Parsing (2.1)

- A *recursive-descent* parser is:
  - A top-down parser
  - Structured as a set of *mutually recursive* procedures

    Each procedure corresponds to a ***non-terminal*** in the grammar.

    See an example.
- Given a ***backtrack-free*** grammar, a tool (a.k.a. *parser generator*) can automatically generate:
  - FIRST, FOLLOW, and START sets
  - An efficient ***recursive-descent*** parser

    This generated parser is called an *LL(1) parser*, which:
    - Processes input from Left to right
    - Constructs a Leftmost derivation
    - Uses a lookahead of 1 symbol
- *LL(1) grammars* are those working in an *LL(1)* scheme.

  LL(1) grammars are ***backtrack-free*** by definition.

Backtrack-Free Parsing (2.2)

- Consider this CFG with **START** sets of the RHSs:

	Production			FIRST$^+$
2	*Expr′*	$\rightarrow$	+ *Term Expr′*	{ + }
3		|	- *Term Expr′*	{ - }
4		|	ϵ	{ ϵ, eof,) }

- The corresponding ***recursive-descent*** parser is structured as:

```
ExprPrim()
  if word = + ∨ word = - then /* Rules 2, 3 */
    word := NextWord()
    if(Term())
      then return ExprPrim()
      else return false
  elseif word = ) ∨ word = eof then /* Rule 4 */
    return true
  else
    report a syntax error
    return false
  end

Term()
  ...
```

See: parser generator

LL(1) Parser: Exercise

Consider the following grammar:

$$L \rightarrow R\ \mathtt{a} \mid Q\ \mathtt{ba} \qquad R \rightarrow \mathtt{aba} \mid \mathtt{caba} \mid R\ \mathtt{bc} \qquad Q \rightarrow \mathtt{bbc} \mid \mathtt{bc}$$

Q. Is it suitable for a ***top-down predictive*** parser?

- If so, show that it satisfies the *LL(1)* condition.
- If not, identify the problem(s) and correct it (them). Also show that the revised grammar satisfies the *LL(1)* condition.

BUP: Discovering Rightmost Derivation

- In TDP, we build the start variable as the ***root node***, and then work towards the ***leaves***. [**leftmost** derivation]
- In Bottom-Up Parsing (BUP):
  - Words (terminals) are still returned from **left** to **right** by the scanner.
  - As terminals, or a mix of terminals and variables, are identified as *reducible* to some variable *A* (i.e., matching the RHS of some production rule for *A*), then a layer is added.
  - Eventually:
    - ***accept***: The ***start variable*** is reduced and **all** words have been consumed.
    - ***reject***: The next word is not `eof`, but no further *reduction* can be identified.

Q. Why can BUP find the ***rightmost*** derivation (RMD), if any?

A. BUP discovers steps in a ***RMD*** in its *reverse* order.

BUP: Discovering Rightmost Derivation (1)

- ***table***-driven *LR(1)* parser: an implementation for BUP, which
  - Processes input from **L**eft to right
  - Constructs a **R**ightmost derivation
  - Uses a lookahead of **1** symbol
- A language has the *LR(1)* property if it:
  - Can be parsed in a single **L**eft to right scan,
  - To build a *reversed* **R**ightmost derivation,
  - Using a lookahead of **1** symbol to determine parsing actions.
- Critical step in a ***bottom-up parser*** is to find the ***next*** *handle*.

BUP: Discovering Rightmost Derivation (2)

```
ALGORITHM: BUParse
  INPUT: CFG G = (V, Σ, R, S), Action & Goto Tables
  OUTPUT: Report Parse Success or Syntax Error
PROCEDURE:
  initialize an empty stack trace
  trace.push(0) /* start state */
  word := NextWord()
  while(true)
    state := trace.top()
    act := Action[state, word]
    if act = ``accept'' then
      succeed()
    elseif act = ``reduce based on A → β'' then
      trace.pop() 2 × |β| times /* word + state */
      state := trace.top()
      trace.push(A)
      next := Goto[state, A]
      trace.push(next)
    elseif act = ``shift to s_i'' then
      trace.push(word)
      trace.push(i)
      word := NextWord()
    else
      fail()
```


BUP: Example Tracing (1)

- Consider the following grammar for parentheses:

1	*Goal*	→	*List*
2	*List*	→	*List Pair*
3		|	*Pair*
4	*Pair*	→	(*Pair*)
5		|	( )

- Assume: tables ***Action*** and ***Goto*** constructed accordingly:

State	Action Table: eof	Action Table: (	Action Table:)	Goto Table: *List*	Goto Table: *Pair*
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

In ***Action*** table:

- s_i: shift to state i
- r_j: reduce to the LHS of production #j

BUP: Example Tracing (2.1)

Consider the steps of performing BUP on input ( ):

Iteration	State	word	Stack	Handle	Action
initial	—	(	\$ 0	— *none* —	—
1	0	(	\$ 0	— *none* —	*shift 3*
2	3	)	\$ 0 (3	— *none* —	*shift 7*
3	7	eof	\$ 0 (3) 7	( )	*reduce 5*
4	2	eof	\$ 0 *Pair* 2	*Pair*	*reduce 3*
5	1	eof	\$ 0 *List* 1	*List*	*accept*

BUP: Example Tracing (2.2)

Consider the steps of performing BUP on input (( )) ( ):

Iteration	State	word	Stack	Handle	Action
initial	—	(	\$ 0	— *none* —	—
1	0	(	\$ 0	— *none* —	*shift 3*
2	3	(	\$ 0 (3	— *none* —	*shift 6*
3	6	)	\$ 0 (3 (6	— *none* —	*shift 10*
4	10	)	\$ 0 (3 (6) 10	( )	*reduce 5*
5	5	)	\$ 0 (3 *Pair* 5	— *none* —	*shift 8*
6	8	(	\$ 0 (3 *Pair* 5) 8	(*Pair*)	*reduce 4*
7	2	(	\$ 0 *Pair* 2	*Pair*	*reduce 3*
8	1	(	\$ 0 *List* 1	— *none* —	*shift 3*
9	3	)	\$ 0 *List* 1 (3	— *none* —	*shift 7*
10	7	eof	\$ 0 *List* 1 (3) 7	( )	*reduce 5*
11	4	eof	\$ 0 *List* 1 *Pair* 4	*List Pair*	*reduce 2*
12	1	eof	\$ 0 *List* 1	*List*	*accept*

BUP: Example Tracing (2.3)

Consider the steps of performing BUP on input $\boxed{\texttt{( ))}}$:

Iteration	State	word	Stack	Handle	Action
initial	—	(	$ 0	— *none* —	—
1	0	(	$ 0	— *none* —	*shift 3*
2	3	)	$ 0 (3	— *none* —	*shift 7*
3	7	)	$ 0 (3) 7	— *none* —	*error*

LR(1) Items: Definition

- In **LR(1)** parsing, ***Action*** and ***Goto*** tabeles encode legitimate ways (w.r.t. a CFG) for finding ***handles*** (for ***reductions***).
- In a ***table***-driven **LR(1)** parser, the table-construction algorithm represents each potential ***handle*** (for a ***reduction***) with an **LR(1)** item e.g.,

$$[A \rightarrow \beta \bullet \gamma,\ \texttt{a}]$$

where:
- A ***production rule*** $\boxed{A \rightarrow \beta\gamma}$ is currently being applied.
- A ***terminal symbol*** $\boxed{\texttt{a}}$ servers as a ***lookahead symbol***.
- A ***placeholder*** $\boxed{\bullet}$ indicates the parser's ***stack top***.
  - ✓ The parser's ***stack*** contains β ("left context").
  - ✓ γ is yet to be matched.
  - Upon matching $\beta\gamma$, if $\texttt{a}$ matches the current word, then we "replace" $\beta\gamma$ (and their associated states) with A (and its associated state).

LR(1) Items: Scenarios

An *LR(1) item* can denote:

1. **Possibility** $[A \rightarrow \bullet\beta\gamma,\ \texttt{a}]$
   - In the current parsing context, an A would be valid.
   - $\bullet$ represents the position of the parser's ***stack top***
   - Recognizing a β next would be one step towards discovering an A.
2. **Partial Completion** $[A \rightarrow \beta \bullet \gamma,\ \texttt{a}]$
   - The parser has progressed from $[A \rightarrow \bullet\beta\gamma,\ \texttt{a}]$ by recognizing β.
   - Recognizing a γ next would be one step towards discovering an A.
3. **Completion** $[A \rightarrow \beta\gamma\bullet,\ \texttt{a}]$
   - Parser has progressed from $[A \rightarrow \bullet\beta\gamma,\ \texttt{a}]$ by recognizing $\beta\gamma$.
   - $\beta\gamma$ found in a context where an A followed by $\texttt{a}$ would be valid.
   - If the current input word matches $\texttt{a}$, then:
     - Current ***complet item*** is a *handle*.
     - Parser can ***reduce*** $\beta\gamma$ to A
     - Accordingly, in the ***stack***, $\beta\gamma$ (and their associated states) are replaced with A (and its associated state).

LR(1) Items: Example (1.1)

Consider the following grammar for parentheses:

1	*Goal*	$\rightarrow$	*List*
2	*List*	$\rightarrow$	*List Pair*
3		|	*Pair*
4	*Pair*	$\rightarrow$	(*Pair*)
5		|	( )

Initial State: $[Goal \rightarrow \bullet List,\ \texttt{eof}]$

Desired Final State: $[Goal \rightarrow List\bullet,\ \texttt{eof}]$

Intermediate States: Subset Construction

Q. Derive all *LR(1) items* for the above grammar.
- $\text{FOLLOW}(List) = \{\texttt{eof},\ (\}$ $\quad \text{FOLLOW}(Pair) = \{\texttt{eof},\ (,\ )\}$
- For each production $A \rightarrow \beta$, given $\text{FOLLOW}(A)$, *LR(1) items* are:

$$\{\ [A \rightarrow \bullet\beta\gamma,\ \texttt{a}] \mid a \in \text{FOLLOW}(A)\ \}$$
$$\cup$$
$$\{\ [A \rightarrow \beta \bullet \gamma,\ \texttt{a}] \mid a \in \text{FOLLOW}(A)\ \}$$
$$\cup$$
$$\{\ [A \rightarrow \beta\gamma\bullet,\ \texttt{a}] \mid a \in \text{FOLLOW}(A)\ \}$$

LR(1) Items: Example (1.2)

Q. Given production $A \to \beta$ (e.g., $Pair \to$ ($Pair$)), how many *LR(1) items* can be generated?

- The current parsing progress (on matching the RHS) can be:
  1. $\bullet$($Pair$)
  2. ($\bullet Pair$)
  3. ($Pair\bullet$)
  4. ($Pair$)$\bullet$
- Lookahead symbol following $Pair$? FOLLOW($Pair$) = {`eof`, (,)}
- All possible *LR(1) items* related to $Pair \to$ ($Pair$)?

✓ [$\bullet$($Pair$), eof]	[$\bullet$($Pair$), (]	[$\bullet$($Pair$),)]
✓ [($\bullet Pair$), eof]	[($\bullet Pair$), (]	[($\bullet Pair$),)]
✓ [($Pair\bullet$), eof]	[($Pair\bullet$), (]	[($Pair\bullet$),)]
✓ [($Pair$)$\bullet$, eof]	[($Pair$)$\bullet$, (]	[($Pair$)$\bullet$,)]

A. How many in general (in terms of A and β)?

$$\underbrace{|\beta| + 1}_{\texttt{possible positions of } \bullet} \times \underbrace{|\text{FOLLOW}(A)|}_{\texttt{possible lookahead symbols}}$$

LR(1) Items: Example (1.3)

A. There are 33 *LR(1) items* in the parentheses grammar.

[$Goal \to \bullet List$,eof]

[$Goal \to List \bullet$,eof]

[$List \to \bullet List\ Pair$,eof]	[$List \to \bullet List\ Pair$,(]
[$List \to List \bullet Pair$,eof]	[$List \to List \bullet Pair$,(]
[$List \to List\ Pair \bullet$,eof]	[$List \to List\ Pair \bullet$,(]
[$List \to \bullet Pair$,eof]	[$List \to \bullet Pair$,(]
[$List \to Pair \bullet$,eof]	[$List \to Pair \bullet$,(]

[$Pair \to \bullet$ ($Pair$),eof]	[$Pair \to \bullet$ ($Pair$),)]	[$Pair \to \bullet$ ($Pair$),(]
[$Pair \to$ ($\bullet Pair$),eof]	[$Pair \to$ ($\bullet Pair$),)]	[$Pair \to$ ($\bullet Pair$),(]
[$Pair \to$ ($Pair \bullet$),eof]	[$Pair \to$ ($Pair \bullet$),)]	[$Pair \to$ ($Pair \bullet$),(]
[$Pair \to$ ($Pair$) $\bullet$,eof]	[$Pair \to$ ($Pair$) $\bullet$,)]	[$Pair \to$ ($Pair$) $\bullet$,(]
[$Pair \to \bullet$ ( ),eof]	[$Pair \to \bullet$ ( ),(]	[$Pair \to \bullet$ ( ),)]
[$Pair \to$ ($\bullet$),eof]	[$Pair \to$ ($\bullet$),(]	[$Pair \to$ ($\bullet$),)]
[$Pair \to$ ( ) $\bullet$,eof]	[$Pair \to$ ( ) $\bullet$,(]	[$Pair \to$ ( ) $\bullet$,)]

LR(1) Items: Example (2)

Consider the following grammar for expressions:

0	$Goal$	$\to$	$Expr$	6	$Term'$	$\to$	x $Factor\ Term'$
1	$Expr$	$\to$	$Term\ Expr'$	7		|	÷ $Factor\ Term'$
2	$Expr'$	$\to$	+ $Term\ Expr'$	8		|	ϵ
3		|	- $Term\ Expr'$	9	$Factor$	$\to$	($Expr$)
4		|	ϵ	10		|	num
5	$Term$	$\to$	$Factor\ Term'$	11		|	name

Q. Derive all *LR(1) items* for the above grammar.
Hints. First compute FOLLOW for each non-terminal:

	Expr	*Expr'*	*Term*	*Term'*	*Factor*
FOLLOW	eof,)	eof,)	eof, +, -,)	eof, +, -,)	eof, +, -, x, ÷,)

Tips. Ignore ϵ ***production*** such as $Expr' \to \epsilon$ since the FOLLOW sets already take them into consideration.

Canonical Collection ($\mathcal{CC}$) vs. LR(1) items

1	$Goal$	$\to$	$List$
2	$List$	$\to$	$List\ Pair$
3		|	$Pair$
4	$Pair$	$\to$	($Pair$)
5		|	( )

Recall:

LR(1) Items: 33 items

Initial State: [$Goal \to \bullet List$, eof]

Desired Final State: [$Goal \to List\bullet$, eof]

- The *canonical collection* [Example of $\mathcal{CC}$]
  $$\mathcal{CC} = \{cc_0, cc_1, cc_2, \dots, cc_n\}$$
  denotes the set of **valid subset states** of a **LR(1) parser**.
  - Each $cc_i \in \mathcal{CC}$ $(0 \le i \le n)$ is a set of ***LR(1) items***.
  - $\mathcal{CC} \subseteq \mathbb{P}($***LR(1) items***$)$ $|\mathcal{CC}|$? [$|\mathcal{CC}| \le 2^{|\textit{LR(1) items}|}$]
- To model a **LR(1) parser**, we use techniques analogous to how an ϵ-NFA is converted into a DFA (subset construction and ϵ-closure).
- **Analogies.**
  - ✓ ***LR(1) items*** ≈ states of source *NFA*
  - ✓ $\mathcal{CC}$ ≈ subset states of target *DFA*

Constructing $\mathcal{CC}$: The *closure* Procedure (1)

```
1   ALGORITHM: closure
2     INPUT: CFG G = (V, Σ, R, S), a set s of LR(1) items
3     OUTPUT: a set of LR(1) items
4   PROCEDURE:
5     lastS := ∅
6     while(lastS ≠ s):
7       lastS := s
8       for [A → ··· • C δ, a] ∈ s:
9         for C → γ ∈ R:
10          for b ∈ FIRST(δa):
11            s := s ∪ { [C → •γ, b] }
12    return s
```

- **Line 8**: $[A \to \cdots \bullet C\ \delta,\ a] \in s$ indicates that the parser's next task is to match $C\ \delta$ with a lookahead symbol a.
- **Line 9**: Given: matching γ can reduce to C
- **Line 10**: Given: $b \in \textsc{First}(\delta a)$ is a valid lookahead symbol after reducing γ to C
- **Line 11**: Add a new item $[C \to \bullet\gamma,\ b]$ into s.
- **Line 6**: Termination is guaranteed.

  ∵ Each iteration adds ≥ 1 item to s (otherwise $lastS \neq s$ is *false*).

Constructing $\mathcal{CC}$: The *closure* Procedure (2.1)

1	$Goal \to List$
2	$List \to List\ Pair$
3	$\mid\ Pair$
4	$Pair \to \underline{(}\ Pair\ \underline{)}$
5	$\mid\ \underline{(}\ \underline{)}$

Initial State: $[Goal \to \bullet List,\ \texttt{eof}]$

Calculate cc_0 = ***closure***$(\{\ [Goal \to \bullet List,\ \texttt{eof}]\ \})$.

Constructing $\mathcal{CC}$: The *goto* Procedure (1)

```
1   ALGORITHM: goto
2     INPUT: a set s of LR(1) items, a symbol x
3     OUTPUT: a set of LR(1) items
4   PROCEDURE:
5     moved := ∅
6     for item ∈ s:
7       if item = [α → β • xδ, a] then
8         moved := moved ∪ { [α → βx • δ, a] }
9       end
10    return closure(moved)
```

Line 7: Given: item $[\alpha \to \beta \bullet x\delta,\ a]$ (where x is the next to match)
Line 8: Add $[\alpha \to \beta x \bullet \delta,\ a]$ (indicating x is matched) to *moved*
Line 10: Calculate and return ***closure***$(moved)$ as the "***next subset state***" from s with a "transition" x.

Constructing $\mathcal{CC}$: The *goto* Procedure (2)

1	$Goal \to List$
2	$List \to List\ Pair$
3	$\mid\ Pair$
4	$Pair \to \underline{(}\ Pair\ \underline{)}$
5	$\mid\ \underline{(}\ \underline{)}$

$$cc_0 = \left\{ \begin{array}{ccc} [Goal \to \bullet List, \texttt{eof}] & [List \to \bullet List\ Pair, \texttt{eof}] & [List \to \bullet List\ Pair, \underline{(}] \\ [List \to \bullet Pair, \texttt{eof}] & [List \to \bullet Pair, \underline{(}] & [Pair \to \bullet \underline{(}\ Pair\ \underline{)}, \texttt{eof}] \\ [Pair \to \bullet \underline{(}\ Pair\ \underline{)}, \underline{(}] & [Pair \to \bullet \underline{(}\ \underline{)}, \texttt{eof}] & [Pair \to \bullet \underline{(}\ \underline{)}, \underline{(}] \end{array} \right\}$$

Calculate $goto(cc_0,\ \underline{(})$. ["next state" from cc_0 taking $\underline{(}$]

Constructing $\mathcal{CC}$: The Algorithm (1)

```
ALGORITHM: BuildCC
  INPUT: a grammar G = (V, Σ, R, S),  goal production S → S'
  OUTPUT:
    (1) a set CC = {cc_0, cc_1, ..., cc_n} where cc_i ⊆ G's LR(1) items
    (2) a transition function
PROCEDURE:
  cc_0 := closure({[S → •S', eof]})
  CC := {cc_0}
  processed := {cc_0}
  lastCC := ∅
  while (lastCC ≠ CC):
    lastCC := CC
    for cc_i s.t. cc_i ∈ CC ∧ cc_i ∉ processed:
      processed := processed ∪ {cc_i}
      for x s.t. [··· → ··· • x ...] ∈ cc_i
        temp := goto(cc_i, x)
        if temp ∉ CC then
          CC := CC ∪ {temp}
        end
        δ := δ ∪ (cc_i, x, temp)
```


Constructing $\mathcal{CC}$: The Algorithm (2.1)

1	*Goal*	→	*List*
2	*List*	→	*List Pair*
3		|	*Pair*
4	*Pair*	→	(*Pair*)
5		|	( )

- Calculate $\mathcal{CC} = \{cc_0, cc_1, \ldots, cc_{11}\}$
- Calculate the transition function $\delta : \mathcal{CC} \times (\Sigma \cup V) \to \mathcal{CC}$

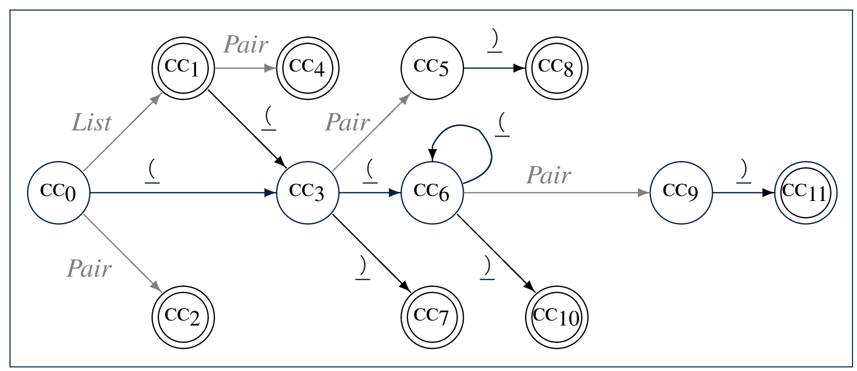
Constructing $\mathcal{CC}$: The Algorithm (2.2)

Resulting transition table:

Iteration	Item	*Goal*	*List*	*Pair*	(	)	eof
0	cc_0	∅	cc_1	cc_2	cc_3	∅	∅
1	cc_1	∅	∅	cc_4	cc_3	∅	∅
	cc_2	∅	∅	∅	∅	∅	∅
	cc_3	∅	∅	cc_5	cc_6	cc_7	∅
2	cc_4	∅	∅	∅	∅	∅	∅
	cc_5	∅	∅	∅	∅	cc_8	∅
	cc_6	∅	∅	cc_9	cc_6	cc_{10}	∅
	cc_7	∅	∅	∅	∅	∅	∅
3	cc_8	∅	∅	∅	∅	∅	∅
	cc_9	∅	∅	∅	∅	cc_{11}	∅
	cc_{10}	∅	∅	∅	∅	∅	∅
4	cc_{11}	∅	∅	∅	∅	∅	∅

Constructing $\mathcal{CC}$: The Algorithm (2.3)

Resulting DFA for the parser:



Constructing $\mathcal{CC}$: The Algorithm (2.4.1)

Resulting canonical collection $\mathcal{CC}$: [Def. of $\mathcal{CC}$]

$$cc_0 = \left\{ \begin{array}{ccc} [Goal \rightarrow \bullet\, List, \texttt{eof}] & [List \rightarrow \bullet\, List\ Pair, \texttt{eof}] & [List \rightarrow \bullet\, List\ Pair, \underline{(}] \\ [List \rightarrow \bullet\, Pair, \texttt{eof}] & [List \rightarrow \bullet\, Pair, \underline{(}] & [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \texttt{eof}] \\ [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \underline{(}] & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \texttt{eof}] & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{(}] \end{array} \right\}$$

$$cc_1 = \left\{ \begin{array}{ccc} [Goal \rightarrow List\, \bullet, \texttt{eof}] & [List \rightarrow List\, \bullet\, Pair, \texttt{eof}] & [List \rightarrow List\, \bullet\, Pair, \underline{(}] \\ [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \texttt{eof}] & [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \underline{(}] & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \texttt{eof}] \\ & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{(}] & \end{array} \right\}$$

$$cc_2 = \left\{ [List \rightarrow Pair\, \bullet, \texttt{eof}] \quad [List \rightarrow Pair\, \bullet, \underline{(}] \right\}$$

$$cc_3 = \left\{ \begin{array}{ccc} [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \underline{)}] & [Pair \rightarrow \underline{(}\, \bullet\, Pair\ \underline{)}, \texttt{eof}] & [Pair \rightarrow \underline{(}\, \bullet\, Pair\ \underline{)}, \underline{(}] \\ [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{)}] & [Pair \rightarrow \underline{(}\, \bullet\, \underline{)}, \texttt{eof}] & [Pair \rightarrow \underline{(}\, \bullet\, \underline{)}, \underline{(}] \end{array} \right\}$$

$$cc_4 = \left\{ [List \rightarrow List\ Pair\, \bullet, \texttt{eof}] \quad [List \rightarrow List\ Pair\, \bullet, \underline{(}] \right\}$$

$$cc_5 = \left\{ [Pair \rightarrow \underline{(}\ Pair\, \bullet\, \underline{)}, \texttt{eof}] \quad [Pair \rightarrow \underline{(}\ Pair\, \bullet\, \underline{)}, \underline{(}] \right\}$$

$$cc_6 = \left\{ \begin{array}{cc} [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \underline{)}] & [Pair \rightarrow \underline{(}\, \bullet\, Pair\ \underline{)}, \underline{)}] \\ [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{)}] & [Pair \rightarrow \underline{(}\, \bullet\, \underline{)}, \underline{)}] \end{array} \right\}$$

$$cc_7 = \left\{ [Pair \rightarrow \underline{(}\ \underline{)}\, \bullet, \texttt{eof}] \quad [Pair \rightarrow \underline{(}\ \underline{)}\, \bullet, \underline{(}] \right\}$$

$$cc_8 = \left\{ [Pair \rightarrow \underline{(}\ Pair\ \underline{)}\, \bullet, \texttt{eof}] \quad [Pair \rightarrow \underline{(}\ Pair\ \underline{)}\, \bullet, \underline{(}] \right\}$$

$$cc_9 = \left\{ [Pair \rightarrow \underline{(}\ Pair\, \bullet\, \underline{)}, \underline{)}] \right\}$$

$$cc_{10} = \left\{ [Pair \rightarrow \underline{(}\ \underline{)}\, \bullet, \underline{)}] \right\}$$

$$cc_{11} = \left\{ [Pair \rightarrow \underline{(}\ Pair\ \underline{)}\, \bullet, \underline{)}] \right\}$$

Constructing *Action* and *Goto* Tables (1)

```
1  ALGORITHM: BuildActionGotoTables
2    INPUT:
3      (1) a grammar G = (V, Σ, R, S)
4      (2) goal production S → S'
5      (3) a canonical collection CC = {cc_0, cc_1, ..., cc_n}
6      (4) a transition function δ : CC × Σ → CC
7    OUTPUT: Action Table & Goto Table
8  PROCEDURE:
9    for cc_i ∈ CC:
10     for item ∈ cc_i:
11       if item = [A → β • xγ, a] ∧ δ(cc_i, x) = cc_j then
12         Action[i, x] := shift j
13       elseif item = [A → β•, a] then
14         Action[i, a] := reduce A → β
15       elseif item = [S → S'•, eof] then
16         Action[i, eof] := accept
17       end
18     for v ∈ V:
19       if δ(cc_i, v) = cc_j then
20         Goto[i, v] = j
21       end
```

- **L12, 13**: Next valid step in discovering A is to match terminal symbol $\texttt{x}$.
- **L14, 15**: Having recognized β, if current word matches lookahead $\texttt{a}$, reduce β to A.
- **L16, 17**: Accept if input exhausted and what's recognized reducible to start var. S.
- **L20, 21**: Record consequence of a reduction to non-terminal v from state i

Constructing *Action* and *Goto* Tables (2)

Resulting **Action** and **Goto** tables:

	Action Table			*Goto* Table	
State	eof	$\underline{(}$	$\underline{)}$	***List***	***Pair***
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

BUP: Discovering Ambiguity (1)

1	*Goal*	$\rightarrow$	*Stmt*
2	*Stmt*	$\rightarrow$	`if expr then` *Stmt*
3		|	`if expr then` *Stmt* `else` *Stmt*
4		|	`assign`

- Calculate $\mathcal{CC} = \{cc_0, cc_1, \ldots, \}$
- Calculate the transition function $\delta : \mathcal{CC} \times \Sigma \rightarrow \mathcal{CC}$

BUP: Discovering Ambiguity (2.1)

Resulting transition table:

	Item	*Goal*	*Stmt*	`if`	`expr`	`then`	`else`	`assign`	`eof`
0	cc_0	$\emptyset$	cc_1	cc_2	$\emptyset$	$\emptyset$	$\emptyset$	cc_3	$\emptyset$
1	cc_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
	cc_2	$\emptyset$	$\emptyset$	$\emptyset$	cc_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
	cc_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	cc_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	cc_5	$\emptyset$	$\emptyset$	$\emptyset$
3	cc_5	$\emptyset$	cc_6	cc_7	$\emptyset$	$\emptyset$	$\emptyset$	cc_8	$\emptyset$
4	cc_6	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	cc_9	$\emptyset$	$\emptyset$
	cc_7	$\emptyset$	$\emptyset$	$\emptyset$	cc_{10}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
	cc_8	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
5	cc_9	$\emptyset$	cc_{11}	cc_2	$\emptyset$	$\emptyset$	$\emptyset$	cc_3	$\emptyset$
	cc_{10}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	cc_{12}	$\emptyset$	$\emptyset$	$\emptyset$
6	cc_{11}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
	cc_{12}	$\emptyset$	cc_{13}	cc_7	$\emptyset$	$\emptyset$	$\emptyset$	cc_8	$\emptyset$
7	cc_{13}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	cc_{14}	$\emptyset$	$\emptyset$
8	cc_{14}	$\emptyset$	cc_{15}	cc_7	$\emptyset$	$\emptyset$	$\emptyset$	cc_8	$\emptyset$
9	cc_{15}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

BUP: Discovering Ambiguity (2.2.1)

Resulting canonical collection $\mathcal{CC}$:

$$cc_0 = \left\{ \begin{array}{ll} [\mathit{Goal} \rightarrow \bullet\, \mathit{Stmt}, \texttt{eof}] & [\mathit{Stmt} \rightarrow \bullet\, \texttt{if expr then}\ \mathit{Stmt}, \texttt{eof}] \\ [\mathit{Stmt} \rightarrow \bullet\, \texttt{assign}, \texttt{eof}] & [\mathit{Stmt} \rightarrow \bullet\, \texttt{if expr then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}, \texttt{eof}] \end{array} \right\}$$

$$cc_1 = \left\{ [\mathit{Goal} \rightarrow \mathit{Stmt}\, \bullet, \texttt{eof}] \right\}$$

$$cc_2 = \left\{ \begin{array}{l} [\mathit{Stmt} \rightarrow \texttt{if}\ \bullet\ \texttt{expr then}\ \mathit{Stmt}, \texttt{eof}], \\ [\mathit{Stmt} \rightarrow \texttt{if}\ \bullet\ \texttt{expr then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}, \texttt{eof}] \end{array} \right\}$$

$$cc_3 = \left\{ [\mathit{Stmt} \rightarrow \texttt{assign}\, \bullet, \texttt{eof}] \right\}$$

$$cc_4 = \left\{ \begin{array}{l} [\mathit{Stmt} \rightarrow \texttt{if expr}\ \bullet\ \texttt{then}\ \mathit{Stmt}, \texttt{eof}], \\ [\mathit{Stmt} \rightarrow \texttt{if expr}\ \bullet\ \texttt{then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}, \texttt{eof}] \end{array} \right\}$$

$$cc_5 = \left\{ \begin{array}{l} [\mathit{Stmt} \rightarrow \texttt{if expr then}\ \bullet\ \mathit{Stmt}, \texttt{eof}], \\ [\mathit{Stmt} \rightarrow \texttt{if expr then}\ \bullet\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}, \texttt{eof}], \\ [\mathit{Stmt} \rightarrow \bullet\ \texttt{if expr then}\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}], \\ [\mathit{Stmt} \rightarrow \bullet\ \texttt{assign}, \{\texttt{eof}, \texttt{else}\}], \\ [\mathit{Stmt} \rightarrow \bullet\ \texttt{if expr then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

$$cc_6 = \left\{ \begin{array}{l} [\mathit{Stmt} \rightarrow \texttt{if expr then}\ \mathit{Stmt}\ \bullet, \texttt{eof}], \\ [\mathit{Stmt} \rightarrow \texttt{if expr then}\ \mathit{Stmt}\ \bullet\ \texttt{else}\ \mathit{Stmt}, \texttt{eof}] \end{array} \right\}$$

$$cc_7 = \left\{ \begin{array}{l} [\mathit{Stmt} \rightarrow \texttt{if}\ \bullet\ \texttt{expr then}\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}], \\ [\mathit{Stmt} \rightarrow \texttt{if}\ \bullet\ \texttt{expr then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

BUP: Discovering Ambiguity (2.2.2)

Resulting canonical collection $\mathcal{CC}$:

$$cc_8 = \{[\mathit{Stmt} \rightarrow \texttt{assign}\, \bullet, \{\texttt{eof}, \texttt{else}\}]\}$$

$$cc_9 = \left\{ \begin{array}{l} [\mathit{Stmt} \rightarrow \texttt{if expr then}\ \mathit{Stmt}\ \texttt{else}\ \bullet\ \mathit{Stmt}, \texttt{eof}], \\ [\mathit{Stmt} \rightarrow \bullet\ \texttt{if expr then}\ \mathit{Stmt}, \texttt{eof}], \\ [\mathit{Stmt} \rightarrow \bullet\ \texttt{if expr then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}, \texttt{eof}], \\ [\mathit{Stmt} \rightarrow \bullet\ \texttt{assign}, \texttt{eof}] \end{array} \right\}$$

$$cc_{10} = \left\{ \begin{array}{l} [\mathit{Stmt} \rightarrow \texttt{if expr}\ \bullet\ \texttt{then}\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}], \\ [\mathit{Stmt} \rightarrow \texttt{if expr}\ \bullet\ \texttt{then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

$$cc_{11} = \{[\mathit{Stmt} \rightarrow \texttt{if expr then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}\ \bullet, \texttt{eof}]\}$$

$$cc_{12} = \left\{ \begin{array}{l} [\mathit{Stmt} \rightarrow \texttt{if expr then}\ \bullet\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}], \\ [\mathit{Stmt} \rightarrow \texttt{if expr then}\ \bullet\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}], \\ [\mathit{Stmt} \rightarrow \bullet\ \texttt{if expr then}\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}], \\ [\mathit{Stmt} \rightarrow \bullet\ \texttt{if expr then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}], \\ [\mathit{Stmt} \rightarrow \bullet\ \texttt{assign}, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

$$cc_{13} = \left\{ \begin{array}{l} [\mathit{Stmt} \rightarrow \texttt{if expr then}\ \mathit{Stmt}\ \bullet\ , \{\texttt{eof}, \texttt{else}\}], \\ [\mathit{Stmt} \rightarrow \texttt{if expr then}\ \mathit{Stmt}\ \bullet\ \texttt{else}\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

$$cc_{14} = \left\{ \begin{array}{l} [\mathit{Stmt} \rightarrow \texttt{if expr then}\ \mathit{Stmt}\ \texttt{else}\ \bullet\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}], \\ [\mathit{Stmt} \rightarrow \bullet\ \texttt{if expr then}\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}], \\ [\mathit{Stmt} \rightarrow \bullet\ \texttt{if expr then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}], \\ [\mathit{Stmt} \rightarrow \bullet\ \texttt{assign}, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

BUP: Discovering Ambiguity (3)

- Consider cc_{13}

$$cc_{13} = \left\{ \begin{array}{l} [\mathit{Stmt} \rightarrow \texttt{if expr then}\ \mathit{Stmt}\ \bullet\ , \{\texttt{eof}, \texttt{else}\}], \\ [\mathit{Stmt} \rightarrow \texttt{if expr then}\ \mathit{Stmt}\ \bullet\ \texttt{else}\ \mathit{Stmt}, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

  Q. What does it mean if the current word to consume is `else`?
  A. We can either ***shift*** (then expecting to match another *Stmt*) or ***reduce*** to a *Stmt*.
  Action[13, `else`] cannot hold ***shift*** and ***reduce*** simultaneously.
  $\Rightarrow$ This is known as the *shift-reduce conflict*.
- Consider another scenario:

$$cc_i = \left\{ \begin{array}{l} [A \rightarrow \gamma\delta\bullet,\ \texttt{a}], \\ [B \rightarrow \gamma\delta\bullet,\ \texttt{a}] \end{array} \right\}$$

  Q. What does it mean if the current word to consume is `a`?
  A. We can either ***reduce*** to *A* or ***reduce*** to *B*.
  Action[i, `a`] cannot hold *A* and *B* simultaneously.
  $\Rightarrow$ This is known as the *reduce-reduce conflict*.

Index (1)

Index (2)

Index (3)

Index (4)

Index (5)

Index (6)

Index (7)

Index (8)

Composite & Visitor Design Patterns

EECS4302 A:
Compilers and Interpreters
Fall 2022

CHEN-WEI WANG

Learning Objectives

1. Motivating Problem: ***Recursive*** Systems
2. Three Design Attempts
3. Inheritance: ***Abstract Class*** vs. ***Interface***
4. Fourth Design Attempt: ***Composite Design Pattern***
5. Implementing and Testing the Composite Design Pattern

Motivating Problem (1)

- Many manufactured systems, such as computer systems or stereo systems, are composed of ***individual components*** and ***sub-systems*** that contain components.

  e.g., A computer system is composed of:
  - Base equipment (*hard drives*, *cd-rom drives*)

    e.g., Each *drive* has **properties**: e.g., power consumption and cost.
  - Composite equipment such as *cabinets*, *busses*, and *chassis*

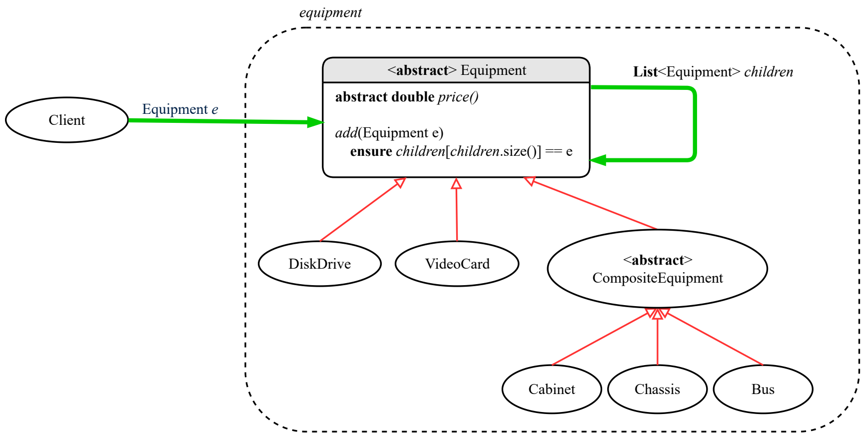
    e.g., Each *cabinet* contains various types of *chassis*, each of which containing components (*hard-drive*, *power-supply*) and *busses* that contain *cards*.
- Design a system that will allow us to easily *build* systems and *compute* their aggregate cost and power consumption.

Motivating Problem (2)

Design of ***hierarchies*** represented in ***tree structures***

Challenge: There are ***base*** and ***recursive*** modelling artifacts.

Design Attempt 1: Architecture



Java List API

Design Attempt 1: Flaw?

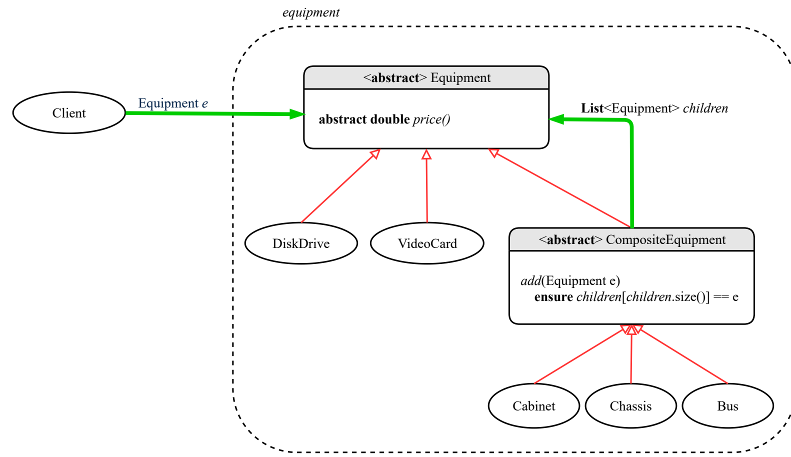
Q: Any flaw of this first design?

A: Two "composite" features defined at the `Equipment` level:

- `List<Equipment> children`
- `add(Equipment child)`

⇒ Inherited to each ***base*** equipment (e.g., `DiskDrive`), for which such features are **not** applicable.

Design Attempt 2: Architecture



Design Attempt 2: Flaw?

Q: Any flaw of this second design?

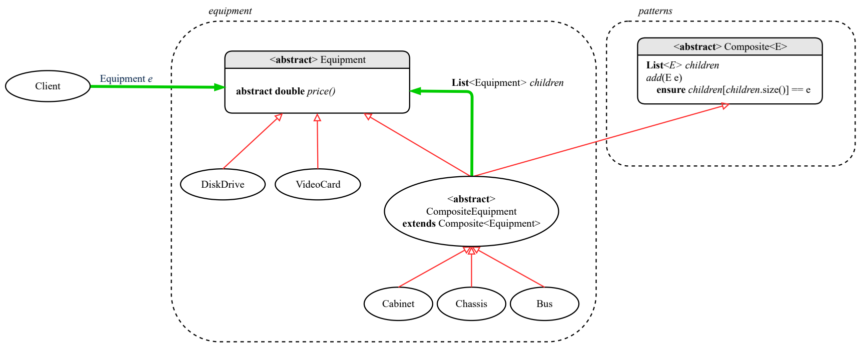
A: Two "composite" features defined at the `Composite` level:

- `List<Equipment> children`
- `add(Equipment child)`

⇒ Multiple ***types*** of the composite (e.g., equipment, furniture) cause duplicates of the `Composite` class.

⇒ Use a ***generic (type) parameter*** to ***abstract*** away the ***concrete*** type of any potential composite.

Design Attempt 3: Architecture



Design Attempt 3: Flaw?

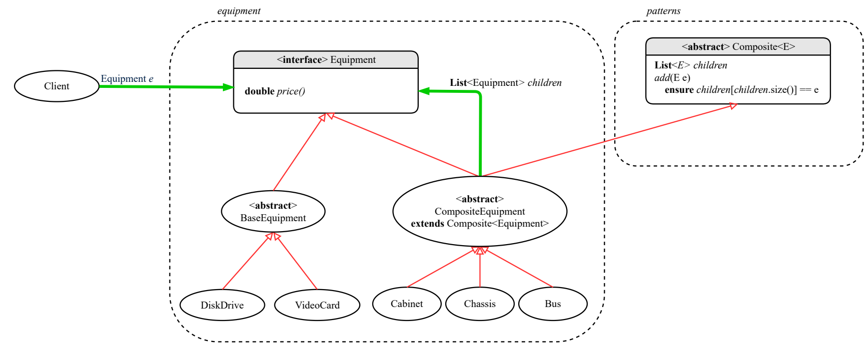
Q: Any flaw of this third design?

A: It does **not** compile:

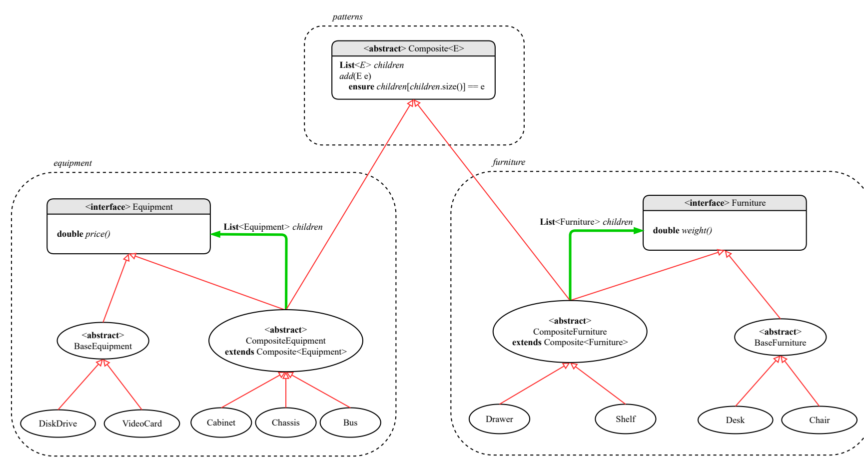
Java does not support ***multiple inheritance***!

- See: https://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html
- A class may inherit from at most one class (**abstract** or not).
  Rationale. ***MI*** results in name clashes
  [a.k.a. the ***Diamond Problem***].
- However, a class may implement multiple ***interfaces***.
  [workaround for implementation]

The Composite Pattern: Architecture



The Composite Pattern: Instantiations



Implementing the Composite Pattern (1)

```java
public interface Equipment {
  public String name();
  public double price(); /* uniform access */
}
```

```java
public abstract class BaseEquipment implements Equipment {
  private String name;
  private double price;
  public BaseEquipment(String name, double price) {
    this.name = name; this.price = price;
  }
  public String name() { return this.name; }
  public double price() { return this.price; }
}
```

```java
public class VideoCard extends BaseEquipment {
  public VideoCard(String name, double price) {
    super(name, price);
  }
}
```


Implementing the Composite Pattern (2.1)

```java
import java.util.List;

public abstract class Composite<E> {
  protected List<E> children;

  public void add(E child) {
    children.add(child); /* polymorphism */
  }
}
```


Implementing the Composite Pattern (2.2)

```java
import java.util.ArrayList;

public abstract class CompositeEquipment
  extends Composite<Equipment>
  implements Equipment
{
  private String name;
  public CompositeEquipment(String name) {
    this.name = name;
    this.children = new ArrayList<>();
  }
  public String name() { return this.name; }
  public double price() {
    double result = 0.0;
    for(Equipment child : this.children) {
      result = result + child.price(); /* dynamic binding */
    }
    return result;
  }
}
```


Implementing the Composite Pattern (2.2)

```java
public class Chassis extends CompositeEquipment {
  public Chassis(String name) {
    super(name);
  }
}
```

Testing the Composite Pattern

```java
@Test
public void test_equipment() {
  Equipment card, drive;
  Bus bus;
  Cabinet cabinet;
  Chassis chassis;

  card = new VideoCard("16Mbs Token Ring", 200);
  drive = new DiskDrive("500 GB harddrive", 500);
  bus = new Bus("MCA Bus");
  chassis = new Chassis("PC Chassis");
  cabinet = new Cabinet("PC Cabinet");
  bus.add(card);
  chassis.add(bus);
  chassis.add(drive);
  cabinet.add(chassis);

  assertEquals(700.00, cabinet.price(), 0.1);
}
```


Summay: The Composite Pattern

- **Design**: Categorize into ***base*** artifacts or ***recursive*** artifacts.
- **Programming**:
  Build the ***tree structure*** representing some ***hierarchy***.
- **Runtime**:
  Allow clients to treat ***base*** objects (leafs) and ***recursive*** compositions (nodes) *uniformly* (e.g., `price()`).

  ⇒ *Polymorphism* : ***leafs*** and ***nodes*** are "substitutable".

  ⇒ *Dynamic Binding* : Different versions of the same operation is applied on ***base objects*** and ***composite objects***.

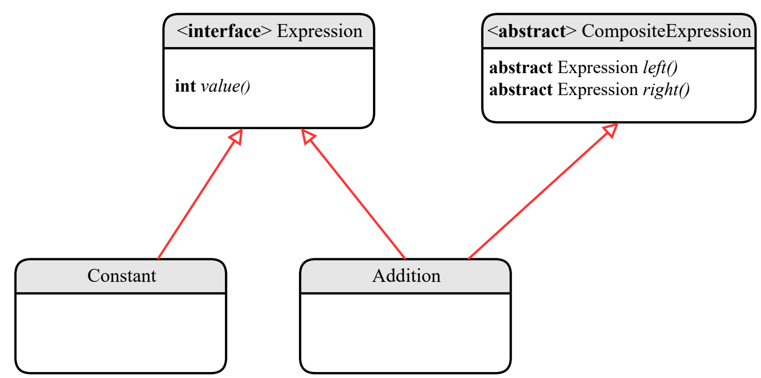
  e.g., Given ***Equipment*** e :
  - `e.price()` may return the unit price, e.g., of a ***`DiskDrive`***.
  - `e.price()` may sum prices, e.g., of a ***`Chassis`***' containing equipment.

Learning Objectives

1. Motivating Problem: ***Processing*** Recursive Systems
2. First Design Attempt: Cohesion & Single-Choice Principle?
3. Design Principles:
   - *Cohesion*
   - *Single Choice* Principle
   - *Open-Closed* Principle
4. Second Design Attempt: ***Visitor Design Pattern***
5. Implementing and Testing the Visitor Design Pattern

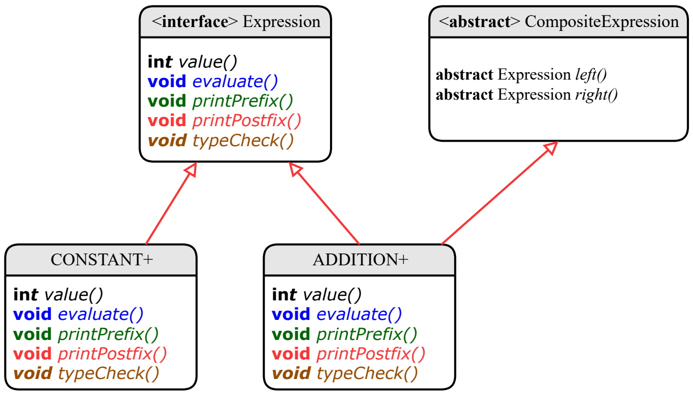
Motivating Problem (1)

Based on the *composite pattern* you learned, design classes to model ***structures*** of arithmetic expressions (e.g., *`341`*, *`2`*, *`341 + 2`*).



Motivating Problem (2)

Extend the *composite pattern* to support ***operations*** such as `evaluate`, pretty printing (`print_prefix`, `print_postfix`), and `type_check`.



Design Principles: Information Hiding & Single Choice

- ***Cohesion***:
  - A class/module groups ***relevant*** features (data & operations).
- ***Single Choice Principle*** (SCP):
  - When a ***change*** is needed, there should be ***a single place*** (or ***a minimal number of places***) where you need to make that change.
  - Violation of SCP means that your design contains ***redundancies***.

Problems of Extended Composite Pattern

- Distributing **unrelated** ***operations*** across nodes of the ***abstract syntax tree*** violates the ***single-choice principle***:
  To add/delete/modify an operation
  ⇒ Change of all descendants of `Expression`
- Each node class lacks in ***cohesion***:
  A **class** should group ***relevant*** concepts in a **single** place.
  ⇒ Confusing to mix codes for evaluation, pretty printing, type checking.
  ⇒ Avoid "polluting" the classes with these **unrelated** operations.

Open/Closed Principle

- Software entities (classes, features, etc.) should be ***open*** for ***extension***, but ***closed*** for ***modification***.
  ⇒ As a system evolves, we:
  - May add/modify the ***open*** (unstable) part of system.
  - May **not** add/modify the ***closed*** (stable) part of system.
- e.g., In designing the application of an expression language:
  - ALTERNATIVE 1:
    Syntactic constructs of the language may be ***open***, whereas operations on the language may be ***closed***.
  - ALTERNATIVE 2:
    Syntactic constructs of the language may be ***closed***, whereas operations on the language may be ***open***.

Visitor Pattern

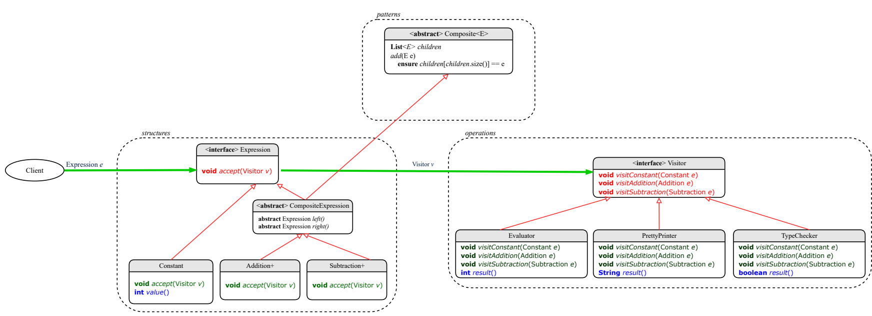
- ***Separation of concerns***:
  - Set of language (syntactic) constructs
  - Set of operations

  ⇒ Classes from these two sets are *decoupled* and organized into two separate packages.
- ***Open-Closed Principle*** (OCP): [**ALTERNATIVE 2**]
  - ***Closed***, staple part of system: set of language constructs
  - ***Open***, unstable part of system: set of operations

  ⇒ ***OCP*** helps us determine if the ***Visitor Pattern*** is applicable.

  ⇒ If it is determined that language constructs are ***open*** and operations are ***closed***, then do **not** use the Visitor Pattern.

Visitor Pattern: Architecture



Visitor Pattern Implementation: Structures

Package ***structures***

- Declare `void accept(Visitor v)` in abstract class `Expression`.
- Implement `accept` in each of `Expression`'s descendant classes.

```
public class Constant implements Expression {
  ...
  public void accept(Visitor v) {
    v.visitConstant(this);
  }
}
```

```
public class Addition extends CompositeExpression {
  ...
  public void accept(Visitor v) {
    v.visitAddition(this);
  }
}
```


Visitor Pattern Implementation: Operations

Package ***operations***

- For each descendant class `C` of `Expression`, declare a method header `void visitC (e: C)` in the ***interface*** `Visitor`.

```
public interface Visitor {
  public void visitConstant(Constant e);
  public void visitAddition(Addition e);
  public void visitSubtraction(Subtraction e);
}
```

- Each descendant of `VISITOR` denotes a kind of operation.

```
public class Evaluator implements Visitor {
  private int result;
  ...
  public void visitConstant(Constant e) {
    this.result = e.value();
  }
  public void visitAddition(Addition e) {
    Evaluator evalL = new Evaluator();
    Evaluator evalR = new Evaluator();
    e.getLeft().accept(evalL);
    e.getRight().accept(evalR);
    this.result = evalL.result() + evalR.result();
  }
}
```

Testing the Visitor Pattern

```
1  @Test
2  public void test_expression_evaluation() {
3    CompositeExpression add;
4    Expression c1, c2;
5    Visitor v;
6    c1 = new Constant(1); c2 = new Constant(2);
7    add = new Addition(c1, c2);
8    v = new Evaluator();
9    add.accept(v);
10   assertEquals(3, ((Evaluator) v).result());
11 }
```

Double Dispatch in **Line 9**:

1. ***DT*** of `add` is `Addition` ⇒ Call `accept` in `ADDITION`.
   `v.visitAddition(add)`
2. ***DT*** of `v` is `Evaluator` ⇒ Call `visitAddition` in `Evaluator`.
   visiting result of `add.left()` + visiting result of `add.right()`

To Use or Not to Use the Visitor Pattern

- In the ***visitor pattern***, what kind of ***extensions*** is easy?
  Adding a new kind of ***operation*** element is easy.
  To introduce a new operation for generating C code, we only need to introduce a new descendant class `CCodeGenerator` of `Visitor`, then implement how to handle each language element in that class.
  ⇒ *Single Choice Principle* is satisfied.
- In the ***visitor pattern***, what kind of ***extensions*** is hard?
  Adding a new kind of ***structure*** element is hard.
  After adding a descendant class `Multiplcation` of `Expression`, every concrete visitor (i.e., descendant of `Visitor`) must be amended with a new `visitMultiplication` operation.
  ⇒ *Single Choice Principle* is violated.
- The applicability of the visitor pattern depends on to what extent the ***structure*** will change.
  ⇒ Use visitor if ***operations*** (applied to structure) change often.
  ⇒ Do not use visitor if the ***structure*** changes often.

Index (1)

Index (2)

Index (3)