Overview of Compilation

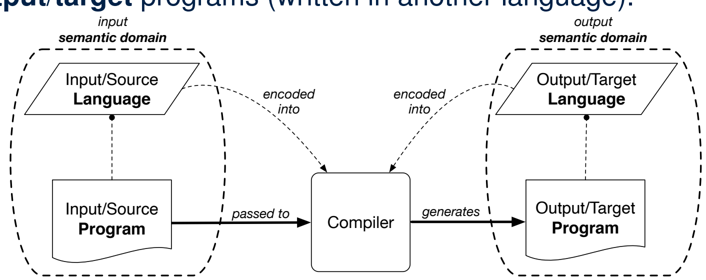
Readings: EAC2 Chapter 1

EECS4302 M:
Compilers and Interpreters
Winter 2020

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).

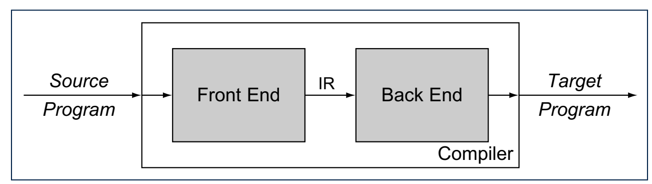


- *Semantic Domain* : context with its own vocabulary and meanings
  e.g., OO, database, predicates
- Source and target may be in **different** ***semantic domains***.
  e.g., Java programs to SQL relational database schemas/queries
  e.g., C procedural programs to MISP assembly instructions

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 them:
  - SYNTAX [***specifiable*** as CFGs]
  - SEMANTICS [***programmable*** as mapping functions]
- Construction of a compiler should conform to good *software engineering principles* .
  - Modularity & Information Hiding [interacting components]
  - Single Choice Principle
  - Design Patterns (e.g., composite, visitor)
  - Regression Testing at different levels: e.g., Unit & Acceptance

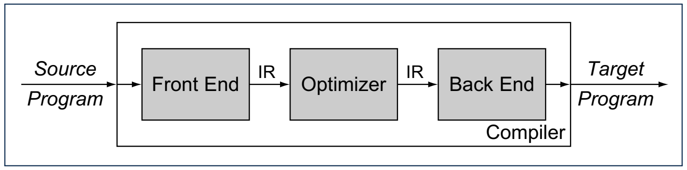
Compiler: Typical Infrastructure (1)



- **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**

Q. How many ***IRs*** needed for building a number of compilers:
JAVA-TO-C, EIFFEL-TO-C, JAVA-TO-PYTHON, EIFFEL-TO-PYTHON?
A. Two IRs suffice: One for OO; one for procedural.
⇒ 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

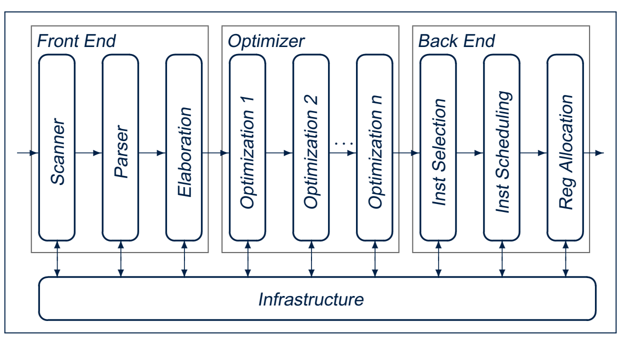
Q. Behaviour of the **target** program predicated upon?

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**?

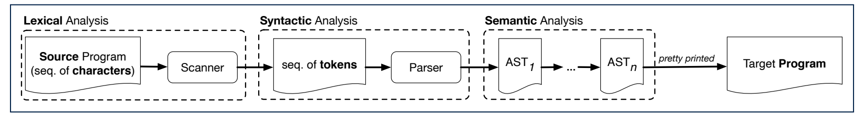
(1) – (3) necessary & sufficient for the *soundness* of a compiler.

Example Compiler One

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



Example Compiler One: Scanner vs. Parser vs. Optimizer



- The same input program may be treated 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 fun is about writing a compiler:

  A series of *semantics-preserving* AST-to-AST transformations.

Example Compiler One: Scanner

- The source program is treated 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 useless delimiter: e.g., space, tab, new line
  e.g., a token for a useful 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 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]+`]

Example Compiler One: 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 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*

Example Compiler One: Optimizer

- 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
```

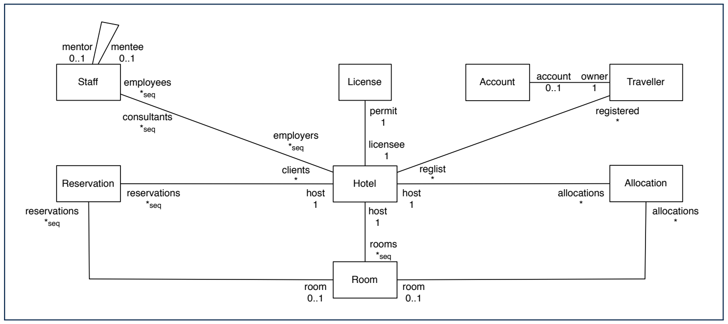
Q. AST of above program ***optimized*** for performance?
A. No ∵ values of `2`, `b`, `c` stay invariant within the loop.

- 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 * d
  end
```


Example Compiler Two

- Consider a compiler which turns a ***Domain-Specific Language (DSL)*** of classes & predicates into a **SQL database**.
- The input/source contains 2 parts:
  - DATA MODEL: classes and associations (client-supplier relations) e.g., data model of a Hotel Reservation System:



  - BEHAVIOURAL MODEL: update methods specified as predicates

Example Compiler Two: Mapping Data

```
class A {
  attributes
    s: string
    as: set(A . b) [*] }
```

```
class B {
  attributes
    is: set (int)
    b: B . as }
```

- 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 Two: 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 Two: Output/Target

- Class associations are compiled into ***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`));
```

- Predicate methods are compiled into ***stored procedures***.

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


Example Compiler Two: Mapping Behaviour

- Challenge: Transform the OO dot notation into table queries.
  e.g., The AST corresponding to the following dot notation
  (in 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`)))
```

- At the database level:
  - Maintaining a large amount of data is ***efficient***
  - Specifying **data** and **updates** is ***tedious*** & ***error-prone***.
  - RESOLUTIONS:
    - Define a DSL supporting the right level of ***abstraction*** for specification
    - Implement a DSL-TO-SQL compiler.

Beyond this lecture ...

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

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

Index (1)

Index (2)

Scanner: Lexical Analysis

Readings: EAC2 Chapter 2

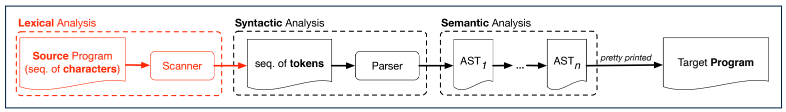
YORK U
UNIVERSITÉ
UNIVERSITY

EECS4302 M:
Compilers and Interpreters
Winter 2020

CHEN-WEI WANG

Scanner in Context

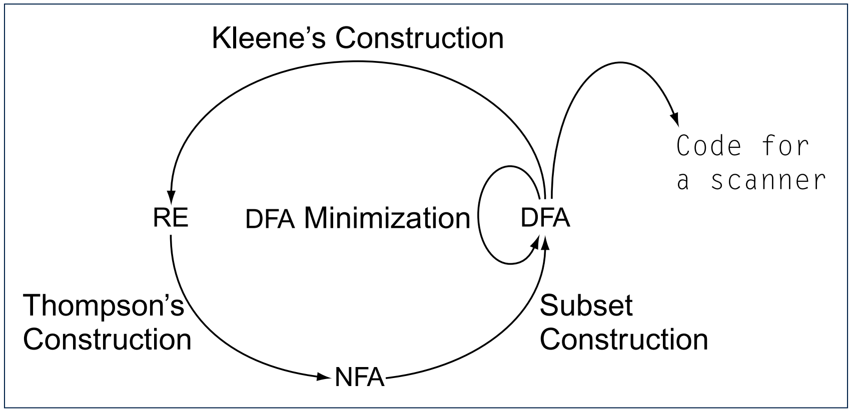
- Recall:



- 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.

    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]
- Use either a *set enumeration* or a *set comprehension* to define your own alphabet.

Strings (1)

- A **string** or a **word** is *finite* sequence of symbols chosen from some *alphabet*.

  e.g., Oxford is a string from the English alphabet Σ_{eng}

  e.g., 01010 is a string from the binary alphabet Σ_{bin}

  e.g., 01010.01 is *not* a string from Σ_{bin}

  e.g., 57 is a string from the binary 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

$$\Sigma^k = \{w \mid w \texttt{ is from } \Sigma \wedge |w| = k\}$$

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

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

- Σ^* is the set of strings of *all possible lengths* from alphabet Σ

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

Review Exercises: Strings

1. What is $|\{a, b, \dots, z\}^5|$?
2. Enumerate, in a systematic manner, the set $\{a, b, c\}^4$.
3. Explain the difference between Σ and Σ^1.
   Σ is a set of *symbols*; Σ^1 is a set of *strings* of length 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 *valid* Java programs

$$L_{Java} = \{prog \mid prog \in \Sigma_{key}^* \wedge prog \texttt{ compiles in Eclipse}\}$$

  - e.g., The language of strings with n 0's followed by n 1's ($n \geq 0$)

$$\{\epsilon, 01, 0011, 000111, \dots\} = \{0^n 1^n \mid n \geq 0\}$$

  - e.g., The language of strings with an equal number of 0's and 1's

$$\begin{aligned} & \{\epsilon, 01, 10, 0011, 0101, 0110, 1100, 1010, 1001, \dots\} \\ = & \{w \mid \texttt{\# of 0's in } w = \texttt{\# of 1's in } w\} \end{aligned}$$

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*]

- e.g., The Java compiler solves the problem of *deciding* if the *string of symbols* typed in the Eclipse editor is a *member* of L_{Java} (i.e., set of Java programs with no syntax and type errors).

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., 01* + 10* denotes $L = \{0x \mid x \in \{1\}^*\} \cup \{1x \mid x \in \{0\}^*\}$
    - e.g., (0*10*10*)*10* denotes $L = \{w \mid w \texttt{ has odd \# of 1's}\}$
    - 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 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**

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

In the textual form, we write + for union.

2. **Concatenation**

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

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

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

In the textual form, we write $*$ 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.

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

    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.

$$L(\ E + F\ ) = L(E) \cup L(F)$$

  - The concatenation EF is a regular expression.

$$L(\ EF\ ) = L(E)L(F)$$

  - Kleene closure of E is a regular expression.

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

  - A parenthesized E is a regular expression.

$$L(\ (E)\ ) = L(E)$$

RE: Construction (3)

Exercises:

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

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

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

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

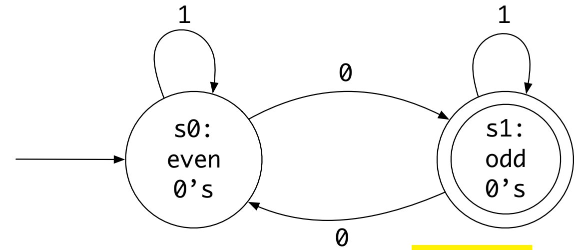
DFA: Deterministic Finite Automata (1.1)

- A *deterministic finite automata (DFA)* is a *finite state machine* (*FSM*) that *accepts* (or recognizes) a pattern of behaviour.
  - For our purpose of this course, 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*:
    - $\{ w \mid w \texttt{ has an odd number of 0's} \}$
    - $\{ w \mid w \texttt{ has an even number of 1's} \}$
    - $\left\{ w \;\middle|\; \begin{array}{l} w \neq \epsilon \\ \land \; w \texttt{ has equal \# of alternating 0's and 1's} \end{array} \right\}$
    - $\{ w \mid w \texttt{ contains } 01 \texttt{ as a substring} \}$
    - $\left\{ w \;\middle|\; \begin{array}{l} w \texttt{ has an even number of 0's} \\ \land \; w \texttt{ has an odd number of 1's} \end{array} \right\}$
- 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* exactly the language

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

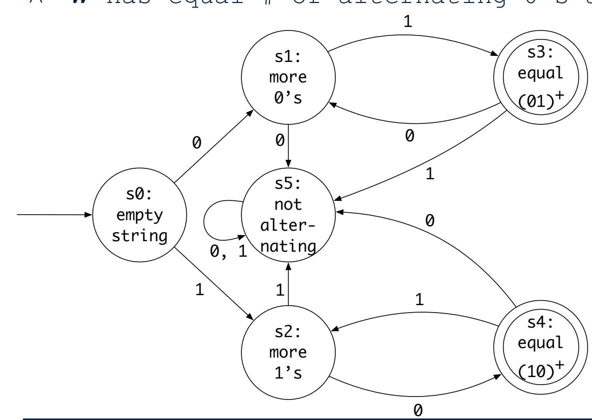


- Each *incoming* or *outgoing* arc (called a *transition*) corresponds to an input alphabet symbol.
- s_0 with an unlabelled *incoming* transition is the *start state*.
- s_3 drawn as a double circle is a *final state*.
- All states have *outgoing* transitions covering $\{0,1\}$.

DFA: Deterministic Finite Automata (1.3)

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

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



Review Exercises: Drawing DFAs

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

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

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)

- 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 (2.3)

- 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!

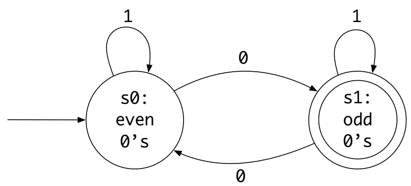
$$\begin{array}{rcl} \hat{\delta}(q, \epsilon) & = & q \\ \hat{\delta}(q, xa) & = & \delta(\ \hat{\delta}(q, x), a\ ) \end{array}$$

  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 $L = L(M)$.

DFA: Deterministic Finite Automata (2.4)



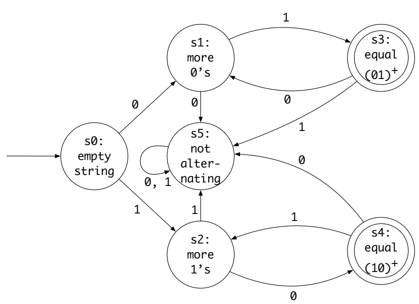
We formalize the above DFA as $M = (Q, \Sigma, \delta, q_0, F)$, where

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

state \ input	0	1
s_0	s_1	s_0
s_1	s_0	s_1

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

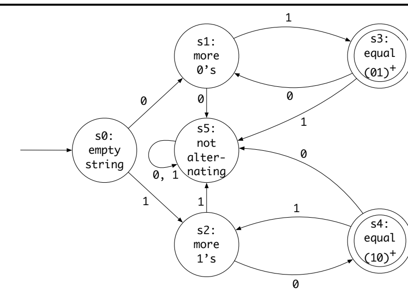
DFA: Deterministic Finite Automata (2.5.1)



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 (2.5.2)



- 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

Review Exercises: Formalizing DFAs

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

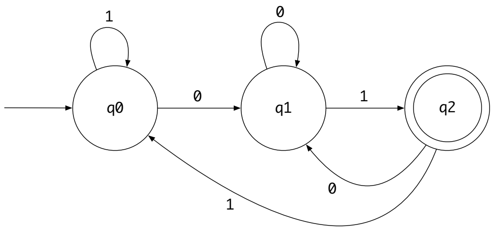
- $\{ w \mid w \texttt{ has an even number of 0's} \}$
- $\{ w \mid w \texttt{ contains } 01 \texttt{ as a substring} \}$
- $\left\{ w \;\middle|\; \begin{array}{l} w \texttt{ has an even number of 0's} \\ \wedge \; w \texttt{ has an odd number of 1's} \end{array} \right\}$

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.



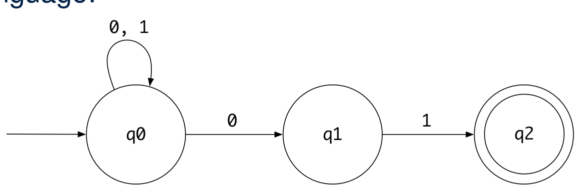
Given an input string w, we may simplify the above DFA by:

- *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

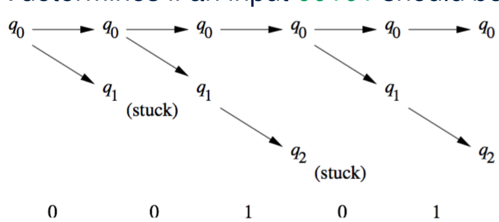
Compare the above DFA with the DFA in slide 39.

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 independently and 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*

    δ takes as arguments a state and an input symbol 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*.
- What is the difference between a *DFA* and an *NFA*?
  - The transition function δ of a *DFA* returns a *single* state.
  - The transition function δ of an *NFA* returns a *set* of states.

NFA: Nondeterministic Finite Automata (3.2)

- 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).

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

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

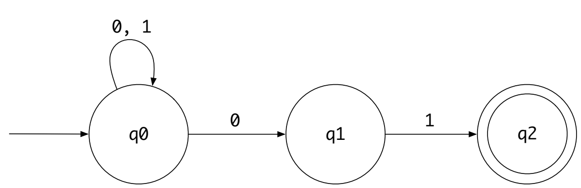
$$\begin{array}{rcl} \hat{\delta}(q, \epsilon) & = & \{q\} \\ \hat{\delta}(q, xa) & = & \bigcup\{\delta(q', a) \mid q' \in \hat{\delta}(q, x)\} \end{array}$$

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*.

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

NFA: Nondeterministic Finite Automata (4)



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`*

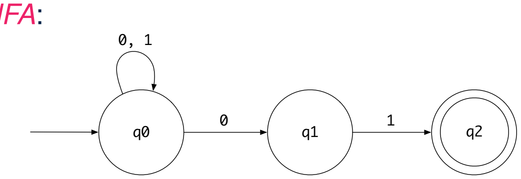
DFA ≡ NFA (1)

- 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.
  - An input 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*.
  - Every language accepted by some *NFA* can also be accepted by some *DFA*.

$$\forall N : NFA \bullet (\exists D : DFA \bullet L(D) = L(N))$$

DFA ≡ NFA (2.2): Lazy Evaluation (1)

Given an *NFA*:



Subset construction (with *lazy evaluation*) produces a *DFA* transition table:

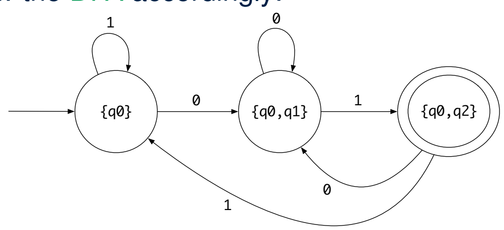
state \ input	0	1
$\{q_0\}$	$\delta(q_0, 0)$ $= \{q_0, q_1\}$	$\delta(q_0, 1)$ $= \{q_0\}$
$\{q_0, q_1\}$	$\delta(q_0, 0) \cup \delta(q_1, 0)$ $= \{q_0, q_1\} \cup \varnothing$ $= \{q_0, q_1\}$	$\delta(q_0, 1) \cup \delta(q_1, 1)$ $= \{q_0\} \cup \{q_2\}$ $= \{q_0, q_2\}$
$\{q_0, q_2\}$	$\delta(q_0,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

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* $N = (Q_N, \Sigma_N, \delta_N, q_0, F_N)$, often only a small portion of the $|\mathbb{P}(Q_N)|$ subset states is *reachable* from $\{q_0\}$.

ALGORITHM: *ReachableSubsetStates*
INPUT: $q_0 : Q_N$; **OUTPUT:** $Reachable \subseteq \mathbb{P}(Q_N)$
PROCEDURE:
$Reachable := \{ \{q_0\} \}$
$ToDiscover := \{ \{q_0\} \}$
while $(ToDiscover \neq \varnothing)$ {
choose $S : \mathbb{P}(Q_N)$ such that $S \in ToDiscover$
remove S from $ToDiscover$
$NotYetDiscovered :=$
$(\{\delta_N(s, 0) \mid s \in S\} \cup \{\delta_N(s, 1) \mid s \in S\}) \setminus Reachable$
$Reachable := Reachable \cup NotYetDiscovered$
$ToDiscover := ToDiscover \cup NotYetDiscovered$
}
return $Reachable$

- RT of *ReachableSubsetStates*? [$O(2^{|Q_N|})$]

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.

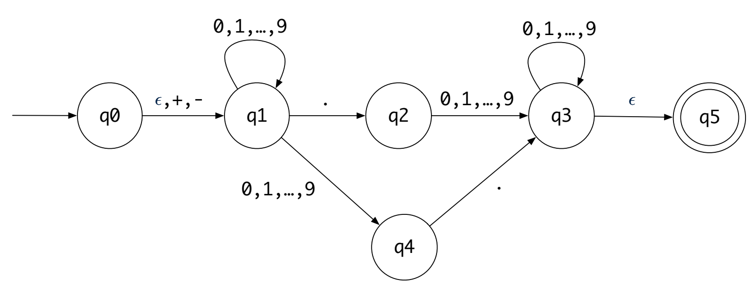
$$\left\{ w : \{0,1\}^* \;\middle|\; \begin{array}{ll} & w \texttt{ has alternating 0's and 1's} \\ \vee & w \texttt{ has an odd \# 0's and an odd \# 1's} \end{array} \right\}$$

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)

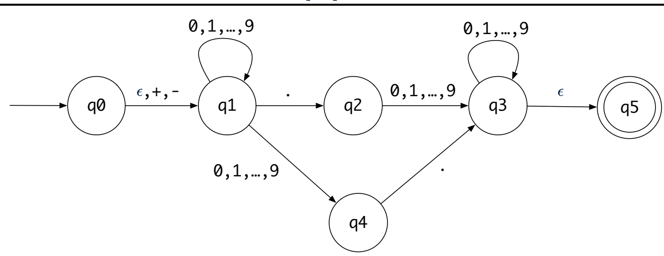
An ϵ-*NFA* is a 5-tuple

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

- Q is a finite set of *states*.
- Σ is a finite set of *input symbols* (i.e., the *alphabet*).
- $\delta : (Q \times (\Sigma \cup \{\epsilon\})) \rightarrow \mathbb{P}(Q)$ 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

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

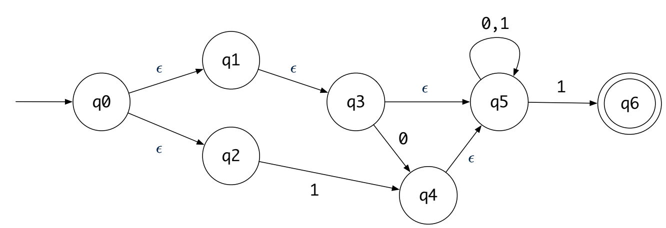
  we define the *epsilon closure* (or ϵ-*closure*) as a function

  $$\texttt{ECLOSE} : Q \rightarrow \mathbb{P}(Q)$$

- For any state $q \in Q$

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

ϵ-NFA: Epsilon-Closures (2)



$$
\begin{aligned}
& \texttt{ECLOSE}(q_0) \\
= \; & \{\delta(q_0, \epsilon) = \{q_1, q_2\}\} \\
& \{q_0\} \cup \texttt{ECLOSE}(q_1) \cup \texttt{ECLOSE}(q_2) \\
= \; & \{\texttt{ECLOSE}(q_1),\ \delta(q_1, \epsilon) = \{q_3\},\ \texttt{ECLOSE}(q_2),\ \delta(q_2, \epsilon) = \varnothing\} \\
& \{q_0\} \cup (\ \{q_1\} \cup \texttt{ECLOSE}(q_3)\ ) \cup (\ \{q_2\} \cup \varnothing\ ) \\
= \; & \{\texttt{ECLOSE}(q_3),\ \delta(q_3, \epsilon) = \{q_5\}\} \\
& \{q_0\} \cup (\ \{q_1\} \cup (\ \{q_3\} \cup \texttt{ECLOSE}(q_5)\ )\ ) \cup (\ \{q_2\} \cup \varnothing\ ) \\
= \; & \{\texttt{ECLOSE}(q_5),\ \delta(q_5, \epsilon) = \varnothing\} \\
& \{q_0\} \cup (\ \{q_1\} \cup (\ \{q_3\} \cup (\ \{q_5\} \cup \varnothing\ )\ )\ ) \cup (\ \{q_2\} \cup \varnothing\ )
\end{aligned}
$$

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!

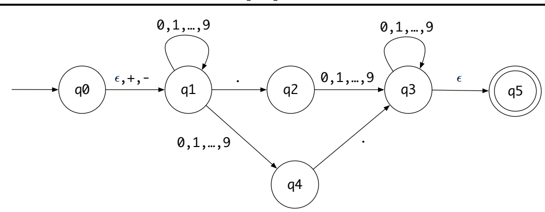
$$\hat{\delta}(q, \epsilon) = \texttt{ECLOSE}(q)$$
$$\hat{\delta}(q, xa) = \bigcup\{\ \texttt{ECLOSE}(q'') \mid q'' \in \delta(q', a) \land q' \in \hat{\delta}(q, x)\ \}$$

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*.

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

ϵ-NFA: Formalization (4)



Given an input string 5.6:

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

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

  $\hat{\delta}(q_0, 5) = \texttt{ECLOSE}(q_1) \cup \texttt{ECLOSE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$
- **Read .**: $\delta(q_1, .) \cup \delta(q_4, .) = \{q_2\} \cup \{q_3\} = \{\ q_2, q_3\ \}$

  $\hat{\delta}(q_0, 5.) = \texttt{ECLOSE}(q_2) \cup \texttt{ECLOSE}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}$
- **Read 6**: $\delta(q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6) = \{q_3\} \cup \{q_3\} \cup \varnothing = \{\ q_3\ \}$

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

DFA $\equiv$ ϵ-NFA: Subset Construction (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: Subset Construction (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:

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

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.

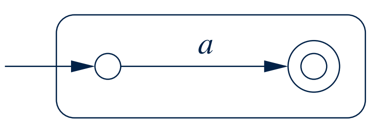
Regular Expression to ϵ-NFA

Base Cases:

- ϵ

- $\varnothing$

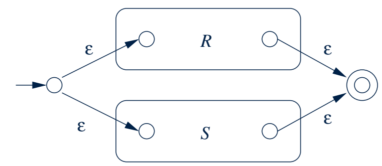
- a [$a \in \Sigma$]



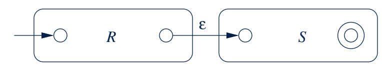
Regular Expression to ϵ-NFA

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

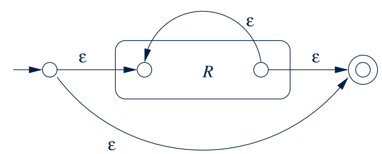
- $R + S$



- RS

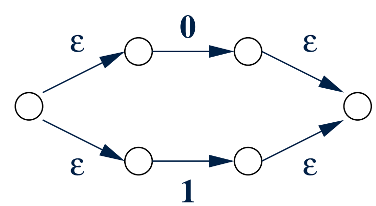


- R^*

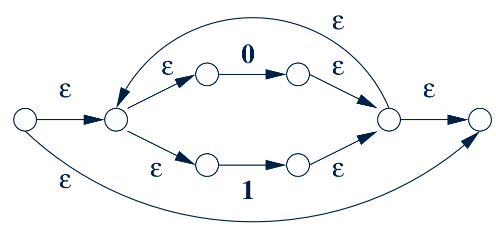


Regular Expression to ϵ-NFA: Examples (1.1)

- $0 + 1$

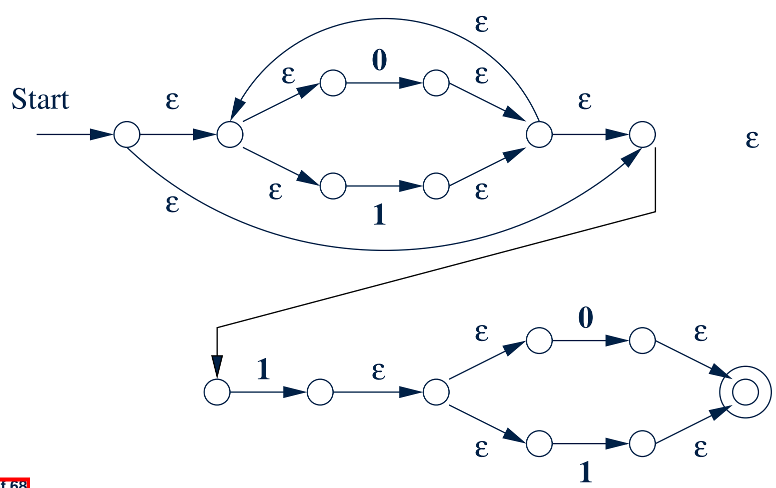


- $(0 + 1)^*$



Regular Expression to ϵ-NFA: Examples (1.2)

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



Minimizing DFA: Motivation

- Recall: Regular Expresion ⟶ ϵ-NFA ⟶ DFA
- DFA produced by the ***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.

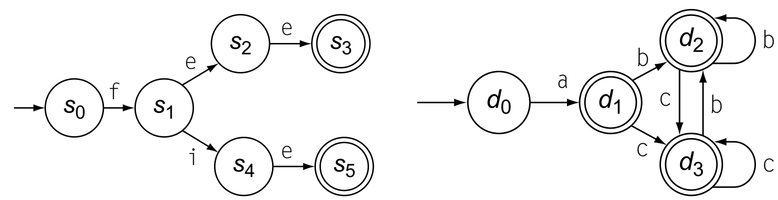
Minimizing DFA: Algorithm

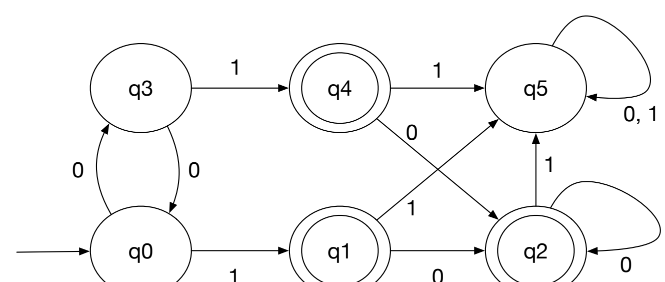
```
ALGORITHM: MinimizeDFAStates
  INPUT: DFA M = (Q, Σ, δ, q0, F)
  OUTPUT: M' s.t. minimum |Q| and equivalent behaviour as M
PROCEDURE:
  P := ∅ /* refined partition so far */
  T := { F, Q - F } /* last refined partition */
  while (P ≠ T):
    P := T
    T := ∅
    for(p ∈ P s.t. |p| > 1):
      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.

- Transition c leads all $s \in S$ to states in the ***same partition*** $p1$.
- Transition c leads some $s \in p - S$ to a ***different partition*** $p2$ ($p2 \neq p1$).

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 <u>all</u> steps.

Implementing DFA as Scanner

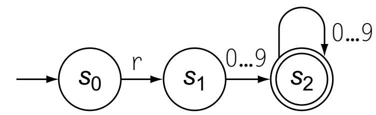
- The source language has a list of ***syntactic categories***:
  - e.g., keyword `while` — `[while]`
  - e.g., identifiers — `[a-zA-Z][a-zA-Z0-9_]*`
  - e.g., white spaces — `[ \t\r]+`
- A compiler's *scanner* must recognize ***words*** from **<u>all</u>** syntactic categories of the source language.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

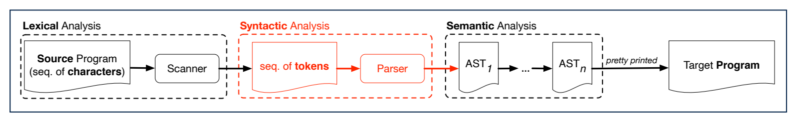
YORK UNIVERSITÉ UNIVERSITY

EECS4302 M:
Compilers and Interpreters
Winter 2020

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*.
- Languages with a *recursive* structure are provably *non-regular*.
  e.g., $\{0^n1^n \mid n \geq 0\}$
- *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 language that we previously proved as *non-regular*

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

can be described using the following *grammar*:

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

- 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

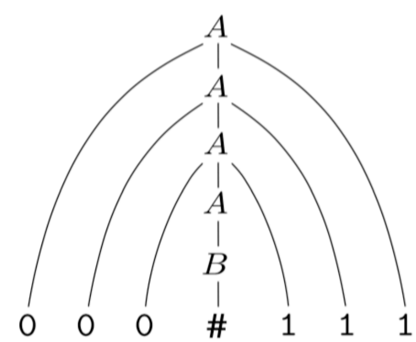
$$\begin{array}{lcl} A & \rightarrow & 0A1 \\ A & \rightarrow & B \\ B & \rightarrow & \# \end{array}$$

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

CFG: Example (1.2)

Given a CFG, the *derivation* of a string 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:

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

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

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

CFG: Example (3)

Design a CFG for the following language:

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

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

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

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. [e.g., (1 + *true*)/*false*]
  2. Relevant operations are grouped together. Try both!

CFG: Example (5.2) Version 1

Expression → *IntegerConstant*
| −*IntegerConstant*
| *BooleanConstant*
| *BinaryOp*
| *UnaryOp*
| (*Expression*)

IntegerConstant → *Digit*
| *Digit IntegerConstant*

Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

BooleanConstant → `TRUE`
| `FALSE`

CFG: Example (5.3) Version 1

BinaryOp → *Expression* + *Expression*
| *Expression* − *Expression*
| *Expression* * *Expression*
| *Expression* / *Expression*
| *Expression* && *Expression*
| *Expression* || *Expression*
| *Expression* => *Expression*
| *Expression* == *Expression*
| *Expression* /= *Expression*
| *Expression* > *Expression*
| *Expression* < *Expression*

UnaryOp → ! *Expression*

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 that a string may have more than one ways to interpret it.
  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

$$
\begin{array}{lcl}
\textit{ArithmeticOp} & \rightarrow & \textit{ArithmeticOp} + \textit{ArithmeticOp} \\
 & | & \textit{ArithmeticOp} - \textit{ArithmeticOp} \\
 & | & \textit{ArithmeticOp} * \textit{ArithmeticOp} \\
 & | & \textit{ArithmeticOp} \; / \; \textit{ArithmeticOp} \\
 & | & (\textit{ArithmeticOp}) \\
 & | & \textit{IntegerConstant} \\
 & | & -\textit{IntegerConstant} \\
\textit{RelationalOp} & \rightarrow & \textit{ArithmeticOp} == \textit{ArithmeticOp} \\
 & | & \textit{ArithmeticOp} \; /= \; \textit{ArithmeticOp} \\
 & | & \textit{ArithmeticOp} > \textit{ArithmeticOp} \\
 & | & \textit{ArithmeticOp} < \textit{ArithmeticOp} \\
\textit{LogicalOp} & \rightarrow & \textit{LogicalOp} \; \&\& \; \textit{LogicalOp} \\
 & | & \textit{LogicalOp} \; || \; \textit{LogicalOp} \\
 & | & \textit{LogicalOp} \; => \; \textit{LogicalOp} \\
 & | & ! \; \textit{LogicalOp} \\
 & | & (\textit{LogicalOp}) \\
 & | & \textit{RelationalOp} \\
 & | & \textit{BooleanConstant}
\end{array}
$$

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*** string that might require further *semantic analysis* :
  e.g., `(1 + 2) / (5 - (2 + 3))`
- Is *ambiguous* , meaning that a string may have more than one ways to interpret it.
  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 $A \in V$, and a rule $A \rightarrow w$:
  - $uAv \Rightarrow uwv$ menas that uAv *yields* uwv.
  - $u \stackrel{*}{\Rightarrow} v$ 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

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

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

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

  - $V = \{Expr, Term, Factor\}$
  - $\Sigma = \{\texttt{a}, +, *, (,)\}$
  - $S = Expr$
- *Precedence* of operators $+$ and $*$ 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 $\varnothing$):

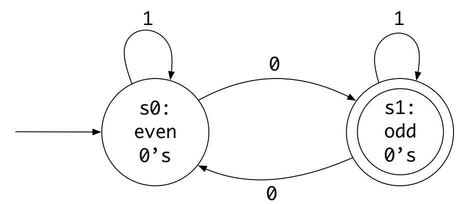
$$\begin{array}{lcl} L(\ \epsilon\ ) & = & \{\epsilon\} \\ L(\ a\ ) & = & \{a\} \\ L(\ E+F\ ) & = & L(E) \cup L(F) \\ L(\ EF\ ) & = & L(E)L(F) \\ L(\ E^*\ ) & = & (L(E))^* \\ L(\ (E)\ ) & = & L(E) \end{array}$$

- 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)

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

- *Unique leftmost derivation* for the string $\texttt{a + a * a}$:

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

- This *leftmost derivation* suggests that $\texttt{a * a}$ is the right operand of $+$.

CFG: Rightmost Derivations (1)

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

- *Unique rightmost derivation* 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 *rightmost derivation* 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 leftmost derivation* 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 *leftmost derivation* suggests that `(a + a)` is the left operand of `*`.

CFG: Rightmost Derivations (2)

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

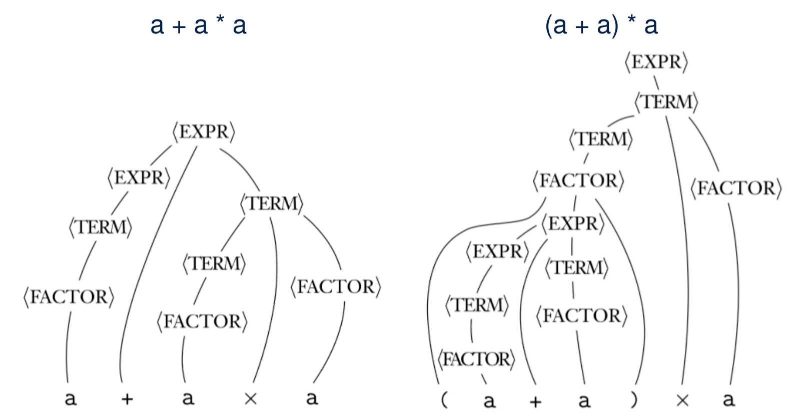
- *Unique rightmost derivation* for the string `(a + a) * a`:

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

This *rightmost derivation* 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 $w \in \Sigma^*$ ⇒ distinct *parse trees* for w?

  A: Not in general ∵ 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 *leftmost* and *rightmost* derivations 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*** *leftmost* derivations or, equally, two or more ***distinct*** *rightmost* derivations.

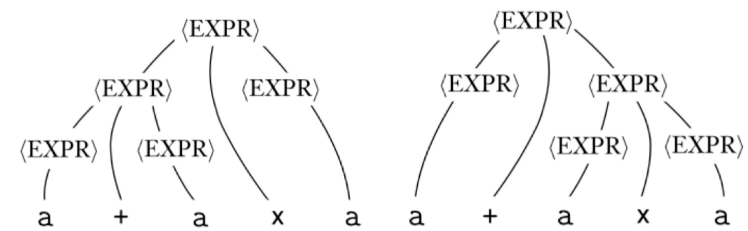
  Here we require that all such derivations have been completed by following a particular order (leftmost or rightmost) to avoid *false alarm*.
- G is *ambiguous* if it generates some string ambiguously.

CFG: Ambiguity: Exercise (1)

- Is the following grammar *ambiguous*?

$$Expr \rightarrow Expr + Expr \mid Expr * Expr \mid (Expr) \mid a$$

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



- ***Distinct ASTs*** (for the ***same input***) mean ***distinct semantic interpretations***: e.g., when a post-order traversal is used to implement evaluation
- **Exercise:** Show *leftmost* derivations for the two parse trees.

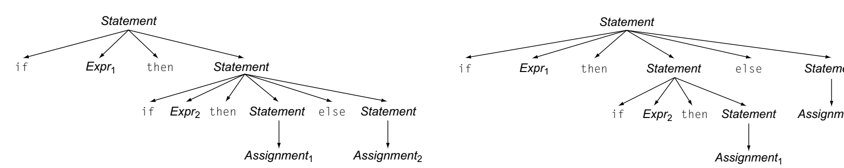
CFG: Ambiguity: Exercise (2.1)

- Is the following grammar *ambiguous*?

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

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

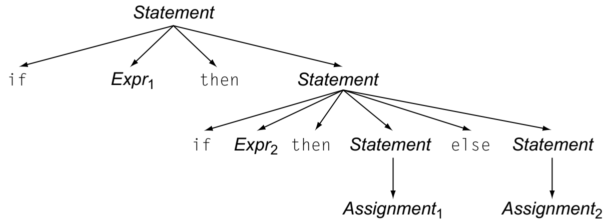
  `if` $Expr_1$ `then if` $Expr_2$ `then` $Assignment_1$ `else` $Assignment_2$



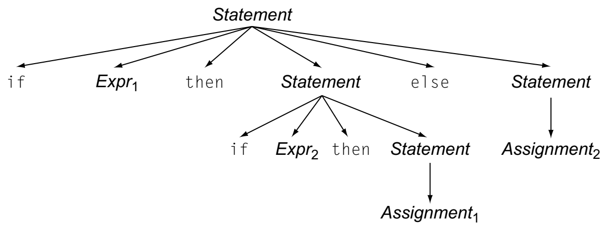
- This is called the *dangling else* problem.
- **Exercise:** Show *leftmost* derivations for the two parse trees.

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`:

```
Statement  →  if Expr then Statement
           |  if Expr then WithElse else Statement
           |  Assignment
WithElse   →  if Expr then WithElse else WithElse
           |  Assignment
```

- When applying `if ... then` *WithElse* `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 $S \overset{*}{\Rightarrow} p$.
  - A parser is supposed to *automate* this derivation process.
    Given an input sequence of (t, c) pairs, where *token* t (e.g., `r241`) belongs to some *syntactic category* c (e.g., `register`):
    Either output a ***valid derivation*** (as an ***AST***), or signal an ***error***.
- In the process of building an *AST* for the input program:
  - **Root** of AST: **start symbol** S of G
  - **Internal nodes**: A subset of variables V of G
  - **Leaves** of AST: **token sequence** input by the scanner

  ⇒ Discovering the *grammatical connections* (according to R) 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 ≜ pop *focus*.siblings; *focus* := *focus*.parent; *focus*.resetChildren

TDP: Exercise (1)

- Given the following CFG **G**:

Expr	$\rightarrow$	*Expr* + *Term*
	$\mid$	*Term*
Term	$\rightarrow$	*Term* $\star$ *Factor*
	$\mid$	*Factor*
Factor	$\rightarrow$	(*Expr*)
	$\mid$	a

  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**:

Expr	$\rightarrow$	*Term* *Expr'*
Expr'	$\rightarrow$	+ *Term* *Expr'*
	$\mid$	ϵ
Term	$\rightarrow$	*Factor* *Term'*
Term'	$\rightarrow$	$\star$ *Factor* *Term'*
	$\mid$	ϵ
Factor	$\rightarrow$	(*Expr*)
	$\mid$	a

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

e.g.,

Expr	$\rightarrow$	*Expr* + *Term*
	$\mid$	*Term*
Term	$\rightarrow$	*Term* $\star$ *Factor*
	$\mid$	*Factor*
Factor	$\rightarrow$	(*Expr*)
	$\mid$	a

e.g.,

Expr	$\rightarrow$	*Expr* + *Term*
	$\mid$	*Expr* − *Term*
	$\mid$	*Term*
Term	$\rightarrow$	*Term* $\star$ *Factor*
	$\mid$	*Term* / *Factor*
	$\mid$	*Factor*

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

A	$\rightarrow$	Br
B	$\rightarrow$	Cd
C	$\rightarrow$	At

$A \rightarrow Br$, $B \overset{*}{\Rightarrow} Atd$

A	$\rightarrow$	Ba	$\mid$	b		
B	$\rightarrow$	Cd	$\mid$	e		
C	$\rightarrow$	Df	$\mid$	g		
D	$\rightarrow$	f	$\mid$	Aa	$\mid$	Cg

$A \rightarrow Ba$, $B \overset{*}{\Rightarrow} Aafd$

TDP: (Preventively) Eliminating LRs

```
1   ALGORITHM: RemoveLR
2     INPUT: CFG G = (V, Σ, R, S)
3     ASSUME: G acyclic ∧ with 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: ⟨⟨A_1, A_2, ..., A_n⟩⟩
8     for i: 1 .. n:
9       for j: 1 .. i-1:
10        if ∃ A_i → A_j γ ∈ R ∧ A_j → δ_1 | δ_2 | ... | δ_m ∈ R then
11          replace A_i → A_j γ with A_i → δ_1 γ | δ_2 γ | ... | δ_m γ
12        end
13      for A_i → A_i α | β ∈ R:
14        replace it with: A_i → β A', A' → α A' | ε
```

L9 to **L11**: Remove ***indirect*** left-recursions from A_1 to A_{i-1}.
L12 to **L13**: 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: $\forall k : k < i \bullet \neg(\exists l \bullet l \leq k \wedge A_k \rightarrow A_l \cdots \in R)$

CFG: Eliminating ϵ-Productions (1)

- Motivations:
  - *TDParse* requires CFG with no ϵ-productions.
  - *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 $A \rightarrow \epsilon$.
- A variable A is *nullable* if $A \stackrel{*}{\Rightarrow} \epsilon$.
  - Each terminal symbol is *not nullable*.
  - Variable A is *nullable* if either:
    - $A \rightarrow \epsilon \in R$; or
    - $A \rightarrow B_1 B_2 \dots B_k \in R$, where each variable B_i $(1 \leq i \leq k)$ is a *nullable*.
- Given a production $B \rightarrow CAD$, if only variable A is nullable, then there are 2 versions of B: $B \rightarrow CAD \mid CD$
- In general, given a production $A \rightarrow X_1 X_2 \dots X_k$ 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 $A \rightarrow \epsilon$]

CFG: Eliminating ϵ-Productions (2)

- Eliminate ϵ-productions from the following grammar:

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

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

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

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 a *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:

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

The FIRST Set: Examples

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

0	*Goal*	→	*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 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* $G = (V, \Sigma, R, S)$
  $T \subset \Sigma^*$ *denotes valid terminals*
  OUTPUT: $\text{FIRST} : V \cup T \cup \{\epsilon, eof\} \longrightarrow \mathbb{P}(T \cup \{\epsilon, eof\})$
PROCEDURE:
  for $\alpha \in (T \cup \{eof, \epsilon\})$: $\text{FIRST}(\alpha) := \{\alpha\}$
  for $A \in V$: $\text{FIRST}(A) := \varnothing$
  $lastFirst := \varnothing$
  while ($lastFirst \neq \text{FIRST}$):
    $lastFirst := \text{FIRST}$
    for $A \to \beta_1\beta_2 \dots \beta_k \in R$ s.t. $\forall \beta_j : \beta_j \in (T \cup V)$:
      rhs $:= \text{FIRST}(\beta_1) - \{\epsilon\}$
      for ($i := 1$; $\epsilon \in \text{FIRST}(\beta_i) \land i < k$; i++):
        rhs $:=$ ***rhs*** $\cup (\text{FIRST}(\beta_{i+1}) - \{\epsilon\})$
      if $i = k \land \epsilon \in \text{FIRST}(\beta_k)$ **then**
        rhs $:=$ ***rhs*** $\cup \{\epsilon\}$
      end
      $\text{FIRST}(A) := \text{FIRST}(A) \cup$ ***rhs***

Computing the FIRST Set: Extension

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

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

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

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

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

- More precisely:

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

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:

$Expr'$	$\rightarrow$	+	$Term$	$Term'$	(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**$(\epsilon) = \{\epsilon\}$.

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

A?. Choose rule (3) when $focus \neq$ **FIRST**$(+) \wedge focus \neq$ **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 α.

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

The FOLLOW 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 **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

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

```
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:

$$\text{FIRST}^+(A \rightarrow \beta) = \begin{cases} \text{FIRST}(\beta) & \text{if } \epsilon \notin \text{FIRST}(\beta) \\ \text{FIRST}(\beta) \cup \text{FOLLOW}(A) & \text{otherwise} \end{cases}$$

FIRST(β) is the extended version where β may be $\beta_1\beta_2 \dots \beta_n$

- Now, a *backtrack-free grammar* has each of its productions $A \rightarrow \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_n$ satisfying:

$$\forall i, j : 1 \leq i, j \leq n \wedge i \neq j \bullet \text{FIRST}^+(\gamma_i) \cap \text{FIRST}^+(\gamma_j) = \varnothing$$

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 % use FOLLOW set as well?
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R ∧ word ∈ FIRST+(β) 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 ≙ 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

- $\epsilon \notin$ **FIRST**(*Factor*) ⇒ **FIRST**$^+$(*Factor*) = **FIRST**(*Factor*)
- **FIRST**(*Factor* → name) = {name}
- **FIRST**(*Factor* → name [*ArgList*]) = {name}
- **FIRST**(*Factor* → name (*ArgList*)) = {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*	→	name
12		|	name [*ArgList*]
13		|	name (*ArgList*)
15	*ArgList*	→	*Expr MoreArgs*
16	*MoreArgs*	→	, *Expr MoreArgs*
17		|	ϵ

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

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

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 FIRST$^+$
- ***Not*** every context-free ***language*** has a corresponding ***backtrack***-free context-free ***grammar***.

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

  $$\exists cfg \mid L(cfg) = l \land isBacktrackFree(cfg)$$
- Given a CFG $g = (V, \Sigma, R, S)$, whether or not g is ***backtrack-free*** is *decidable*:

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

  $$\forall i, j : 1 \leq i, j \leq n \land i \neq j \bullet \text{FIRST}^+(\gamma_i) \cap \text{FIRST}^+(\gamma_j) = \varnothing$$

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 FIRST$^+$ 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 FIRST$^+$ 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	a	R	$\rightarrow$	aba		Q	$\rightarrow$	bbc
	|	Q	ba		|	caba			|	bc
					|	R 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(S) /* start state */
  word := NextWord()
  while(true)
    state := trace.top()
    act := Action[state, word]
    if act = ``accept'' then
      succeed()
    elseif act = ``reduce 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 s_i'' then
      trace.push(word)
      trace.push(s_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			Goto Table	
	eof	(	)	*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		

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 `())`:

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 grammar) 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.,

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

where:

- A production rule $A \rightarrow \beta\gamma$ is currently being applied.
- A placeholder, •, indicates the position of the parser's ***stack top***.
  - ✓ The parser's stack contains β ("left context").
  - ✓ γ is yet to be matched.

    Remark. Upon matching $\beta\gamma$, if $\mathtt{a}$ matches the current word, then we "replace" $\beta\gamma$ (and their corresponding states) with A (and its corresponding state).
- A terminal symbol $\mathtt{a}$ servers as a ***lookahead symbol***.

LR(1) Items: Scenarios

An *LR(1) item* can be:

1. **POSSIBILITY** $[A \rightarrow \bullet\beta\gamma,\ \texttt{a}]$
   - In the current parsing context, an *A* would be valid.
   - • represents the position of the parser's ***stack top***
   - Recognizing a β next would be one step towards discovering an *A*.
2. **PARTIALLY 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 a would be valid.
   - If the current input word matches a, then:
     - Current ***complet item*** is a *handle*.
     - Parser can ***reduce*** $\beta\gamma$ to *A* (and replace $\beta\gamma$ with *A* in its stack).

LR(1) Items: Example (1.1)

Consider the following grammar for parentheses:

1	*Goal*	→	*List*
2	*List*	→	*List Pair*
3		|	*Pair*
4	*Pair*	→	(*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.

- FOLLOW(*List*) = {eof, (} FOLLOW(*Pair*) = {eof, (,)}
- For each production $A \rightarrow \beta$, given 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 $A \rightarrow \beta$ (e.g., *Pair* → (*Pair*)), how many *LR(1) items* can be generated?

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

✓ [• (*Pair*), eof]	[• (*Pair*), (]	[• (*Pair*),)]
✓ [(•*Pair*), eof]	[(•*Pair*), (]	[(•*Pair*),)]
✓ [(*Pair*•), eof]	[(*Pair*•), (]	[(*Pair*•),)]
✓ [(*Pair*)•, eof]	[(*Pair*)•, (]	[(*Pair*)•,)]

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* → • *List*,eof]		
[*Goal* → *List* •,eof]		
[*List* → • *List Pair*,eof]	[*List* → • *List Pair*,(]	
[*List* → *List* • *Pair*,eof]	[*List* → *List* • *Pair*,(]	
[*List* → *List Pair* •,eof]	[*List* → *List Pair* •,(]	
[*List* → • *Pair*,eof]	[*List* → • *Pair*,(]	
[*List* → *Pair* •,eof]	[*List* → *Pair* •,(]	
[*Pair* → • (*Pair*),eof]	[*Pair* → • (*Pair*),)]	[*Pair* → • (*Pair*),(]
[*Pair* → (• *Pair*),eof]	[*Pair* → (• *Pair*),)]	[*Pair* → (• *Pair*),(]
[*Pair* → (*Pair* •),eof]	[*Pair* → (*Pair* •),)]	[*Pair* → (*Pair* •),(]
[*Pair* → (*Pair*) •,eof]	[*Pair* → (*Pair*) •,)]	[*Pair* → (*Pair*) •,(]
[*Pair* → • ( ),eof]	[*Pair* → • ( ),(]	[*Pair* → • ( ),)]
[*Pair* → (•),eof]	[*Pair* → (•),(]	[*Pair* → (•),)]
[*Pair* → ( ) •,eof]	[*Pair* → ( ) •,(]	[*Pair* → ( ) •,)]

LR(1) Items: Example (2)

Consider the following grammar for expressions:

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

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 $Expr' \rightarrow \epsilon$ since the **FOLLOW** sets already take them into consideration.

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

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

Recall:

LR(1) Items: 33 items

Initial State: [*Goal* → •*List*, eof]

Desired Final State: [*Goal* → *List*•, eof]

- The *canonical collection*

  $$\mathcal{CC} = \{cc_0, cc_1, cc_2, \ldots, cc_n\}$$

  denotes the set of ***valid states*** of a ***LR(1)*** parser.
  - Each $cc_i \in \mathcal{CC}$ ($0 \leq i \leq n$) is a set of **LR(1) items**.
  - $\mathcal{CC} \subseteq \mathbb{P}$(**LR(1) items**) $|\mathcal{CC}|$?   [$|\mathcal{CC}| \leq 2^{|\text{LR(1) items}|}$]
- To model a ***LR(1)*** parser, we use techniques similar to how we construct a DFA from an NFA (subset construction and ϵ-closure).
- **Analogies.**
  - ✓ **LR(1) items** ≈ states of source *NFA*
  - ✓ $\mathcal{CC}$ ≈ 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* → ⋯ • *C* δ, *a*] ∈ *s* indicates that the parser's next task is to match *C* δ with a lookahead symbol *a*.
- **Line 9**: Given: matching γ can reduce to *C*
- **Line 10**: Given: b ∈ **FIRST**(δ*a*) is a valid lookahead symbol after reducing γ to *C*
- **Line 11**: Add a new item [*C* → •γ, b] into *s*.
- **Line 6**: Termination is guaranteed.
  ∵ Each iteration adds ≥ 1 item to *s* (otherwise *lastS* ≠ *s* is *false*).

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

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

Initial State: [*Goal* → •*List*, eof]

Calculate cc_0 = *closure*([*Goal* → •*List*, 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 $[\alpha \to \beta \bullet x\delta,\ \mathtt{a}]$ (where x is the next to match)
Line 8: Add $[\alpha \to \beta x \bullet \delta,\ \mathtt{a}]$ (indicating $\mathtt{x}$ is matched) to *moved*
Line 10: Calculate and return *closure*(*moved*) as the "next state" from *s* with a "transition" $\mathtt{x}$.

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

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

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

Calculate *goto*(cc_0, $($). ["next state" from cc_0 taking $($]

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 = {cc0, cc1, ..., ccn} where cci ⊆ G's LR(1) items
    (2) a transition function
PROCEDURE:
  cc0 := closure({[S' → •S, eof]})
  CC := {cc0}
  processed := {cc0}
  lastCC := ∅
  while (lastCC ≠ CC):
    lastCC := CC
    for cci s.t. cci ∈ CC ∧ cci ∉ processed:
      processed := processed ∪ {cci}
      for x s.t. [··· → ··· • x ...] ∈ cci
        temp := goto(cci, x)
        if temp ∉ CC then
          CC := CC ∪ {temp}
        end
        δ := δ ∪ (cci, x, temp)
```


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

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

- Calculate $\mathcal{CC} = \{cc_0, cc_1, \dots, cc_{11}\}$
- Calculate the transition function $\delta : \mathcal{CC} \times \Sigma \to \mathcal{CC}$

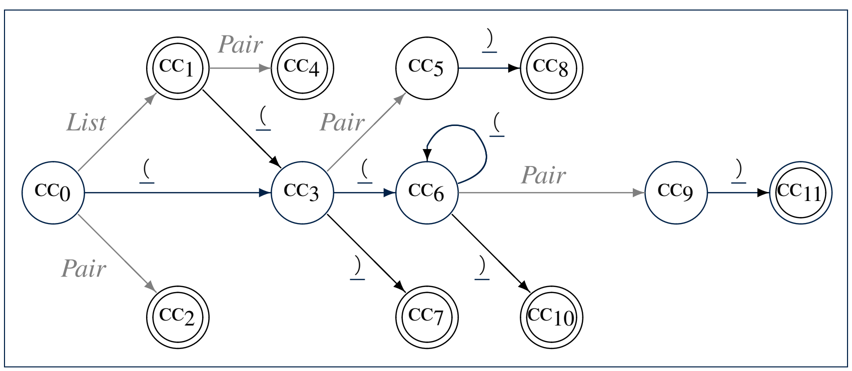
Constructing $\mathcal{CC}$: The Algorithm (2.2)

Resulting transition table:

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

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

Resulting DFA for the parser:



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

Resulting canonical collection $\mathcal{CC}$:

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

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

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

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

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

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

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

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

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

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

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

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

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 = {cc0, cc1, ..., ccn}
6       (4) a transition function δ : CC × Σ → CC
7     OUTPUT: Action Table & Goto Table
8   PROCEDURE:
9     for cci ∈ CC:
10      for item ∈ cci:
11        if item = [A → β • xγ, a]\pause∧δ(cci, x) = ccj 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(cci, v) = ccj then
20          Goto[i, v] = j
21        end
```

- **L12, 13**: Next valid step in discovering A is to match terminal symbol x.
- **L14, 15**: Having recognized β, if current word matches lookahead 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`	`(`	`)`	***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	∅	cc_1	cc_2	∅	∅	∅	cc_3	∅
1	cc_1	∅	∅	∅	∅	∅	∅	∅	∅
	cc_2	∅	∅	∅	cc_4	∅	∅	∅	∅
	cc_3	∅	∅	∅	∅	∅	∅	∅	∅
2	cc_4	∅	∅	∅	∅	cc_5	∅	∅	∅
3	cc_5	∅	cc_6	cc_7	∅	∅	∅	cc_8	∅
4	cc_6	∅	∅	∅	∅	∅	cc_9	∅	∅
	cc_7	∅	∅	∅	cc_{10}	∅	∅	∅	∅
	cc_8	∅	∅	∅	∅	∅	∅	∅	∅
5	cc_9	∅	cc_{11}	cc_2	∅	∅	∅	cc_3	∅
	cc_{10}	∅	∅	∅	∅	cc_{12}	∅	∅	∅
6	cc_{11}	∅	∅	∅	∅	∅	∅	∅	∅
	cc_{12}	∅	cc_{13}	cc_7	∅	∅	∅	cc_8	∅
7	cc_{13}	∅	∅	∅	∅	∅	cc_{14}	∅	∅
8	cc_{14}	∅	cc_{15}	cc_7	∅	∅	∅	cc_8	∅
9	cc_{15}	∅	∅	∅	∅	∅	∅	∅	∅

BUP: Discovering Ambiguity (2.2.1)

Resulting canonical collection $\mathcal{CC}$:

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

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

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

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

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

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

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

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

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*.
  A single *Action* table entry cannot hold these two alternatives.
  This is known as the *shift-reduce conflict*.
- Consider another scenario, say:
  $[A \rightarrow \gamma\delta\bullet,\ \texttt{a}]$
  $[B \rightarrow \gamma\delta\bullet,\ \texttt{a}]$
  Q. What does it mean if the current word to consume is `a`?
  A. We can either ***reduce*** to *A* or ***reduce*** to *B*.
  A single *Action* table entry cannot hold these two alternatives.
  This is known as the *reduce-reduce conflict*.

Index (1)

Index (2)

Index (3)

Index (4)

Index (5)

Index (6)

Index (7)

Index (8)