

Lecture 1

Introduction to Compilers

Spring 2018/2019

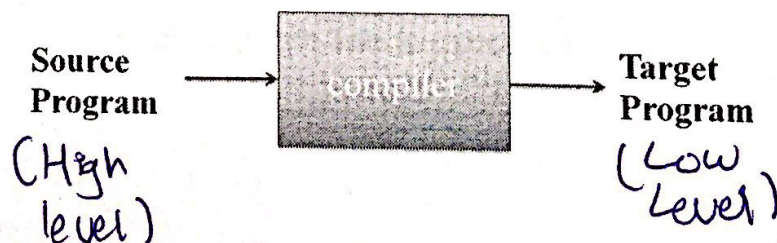
Instructor: Dr. Fahed Jubair
 Computer Engineering Department
 University of Jordan



Definition



A compiler is a program that translates a program written in one language, the source language, into an equivalent program in another language, the target language.



Like Assembly.

© All Rights Reserved

Compilers
 translate from
 High level language
 to Low level
 language

User Expectation



We expect the compiler to:

1. Preserve the meaning of the program being translated correctness.
2. Report "detectable" errors → interactive
3. Perform optimizations during the translation

↳ to execute on any hardware machine

© All Rights Reserved

يعني البرنامج بعد

التحويل
القوي يكون نفسه قبل

the compiler interact with the user to tell them if they made mistakes

Program Lifecycle



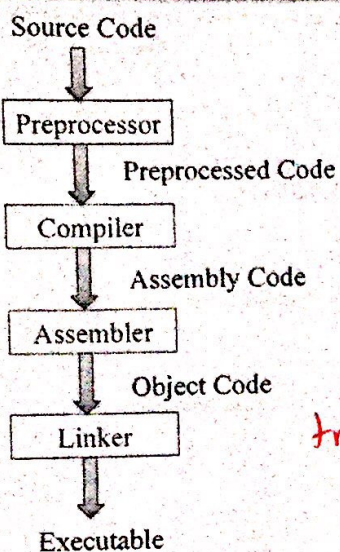
- Compile-time: the source program is translated into object code
- Link-time: the object code is turned into an executable file
- Run-time: executable file is executed

© All Rights Reserved

before execution

execution time

Source-to-Executable



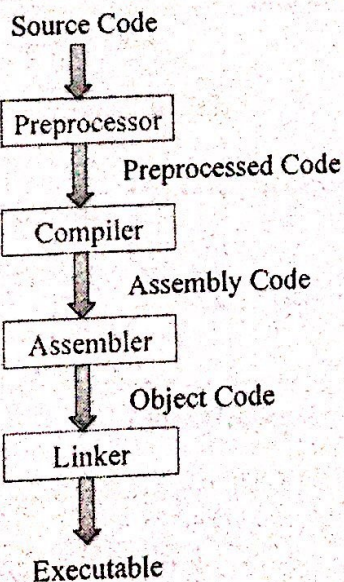
1. Preprocessors take the input source code and prepare it for the compiler

- Handle directives
- Expand macros
- etc

هذه الخطوة بتجهز لكود الـ translation (optional)

© All Rights Reserved

Source-to-Executable

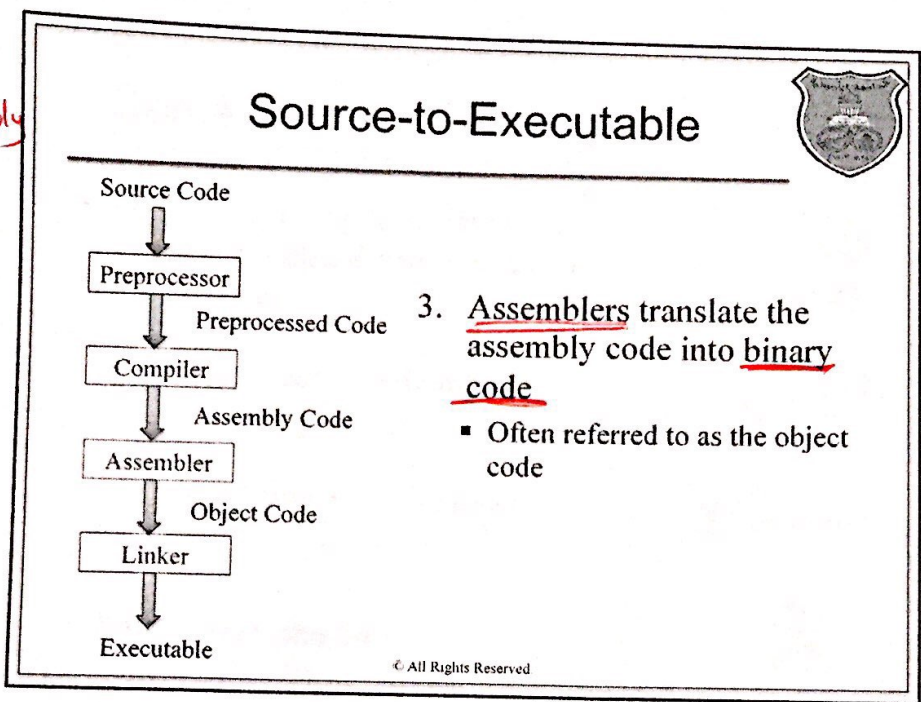


2. Compilers translate the source code into a machine code (Assembly)

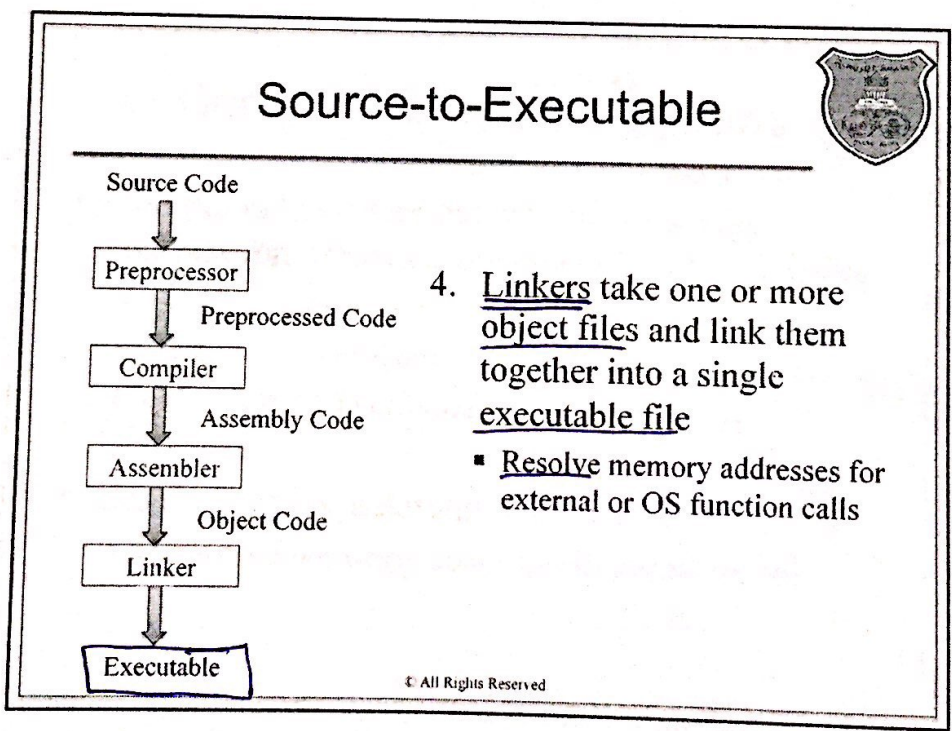
- Often referred to as the assembly code
- Some compilers generate machine-independent codes

© All Rights Reserved

Machine/Assembly Code
↓
Low Level Language Code



Object Code
↓
0's & 1's code
(binary code).



Some Existing Compilers

- GCC (GNU Compiler Collection)
 - Multiple languages: Fortran, C, C++, etc
 - Open source
- ICC (Intel C and C++ Compiler)
- Microsoft Visual C++ Compiler
- Javac (Java Compiler)



© All Rights Reserved

Compiler Command Options

- Compilers have user options that make the compilation process more flexible and interactive
 - Help users debug their code
 - Help users control how to utilize machine resources
 - Help users find performance bottlenecks
- Compilers have a default behavior
 - Users need not worry about specifying all options



© All Rights Reserved

input for the compiler Source code
 Output of the compiler object file.

Linker → combines the obj. file into executable file

Demo With GCC on Linux

- Consider two source code files in C:
 - factorial.c
 - Computes the factorial for an integer x
 - main.c
 - Reads an input integer from the user
 - Computes the factorial of this integer by calling routines from factorial.c
 - Prints the result

© All Rights Reserved

for optimization -

Demo With GCC on Linux

- gcc -O3 -c main.c -o main.o
 - Compiles "main.c" and produces the **object file** "main.o"
 - Option "O3" specifies the optimization level
 - Option "c" instructs the compiler to compile and assemble but do not link
 - Option "o" specifies the name of the output file
 - Note that this command performs preprocessing, compiling and assembling in one command
- gcc -O3 -c factorial.c -o factorial.o
 - Compiles "factorial.c" and produces the **object file** "factorial.o"

© All Rights Reserved

لو - ما - جملين
-c
كان روح يطلعني
executable
File
بمباشرة

إذا ما استخدمنا
بجانبه عن نفس اسم
الفايل

gcc -c main.c → Command يستخدمه عندنا اقل Compile
الفايل اللي عندي روح يطلعني
main.o
file

links to
show 2
by default
(a.out).

Demo With GCC on Linux



• `gcc main.o factorial.o -o exec`

- Links and combines both object files together into a single executable file "exec"
- The linking process also includes any runtime routines

• `gcc -O3 main.c factorial.c -o exec`

- Do everything in one go: compile and assemble both "main.c" and "factorial.c" to obtain object files, then link and combine the object files to obtain the executable file "exec"

© All Rights Reserved

This ~~code~~ Command
Combines
the two object
files
(Linker).

Related Terminology



- Runtime library
- Interpreter
- JIT: Just-in-time compilation
- Source-to-source translation
- IDE: Integrated Development Environment

© All Rights Reserved

Runtime Library



- Collection of routines that provides services to programs during their executions
 - Examples: garbage collection, stream input/output, thread management, memory management, etc
- Programmers can directly request specific runtime services using the APIs provided by runtime libraries

© All Rights Reserved

Cons of interpreter (translation overhead)

صار البرنامج انبطأ

Python uses interpreter

Interpreters



- An interpreter is a program that directly executes an input program without the need to do translation first
 - Interpreters often rely on runtime support from the OS



© All Rights Reserved


سؤاله من interpreter?
ما جتهد على اد OS
ان تستعمل على كل اد hardware

Interpreter → translation and execution at the same time (it executes line by line).

Java takes benefits of both compiler and interpreter

Two commands:
- javac
↓
Compiler
- java
↓
Interpreter


An Example: Java



- At compile time, Java compiler translates the source program into a "virtual" machine code called **bytecode**
 - Unlike "native" machine codes produced by traditional compilers, bytecode is machine-independent, i.e., designed for an abstract (i.e., virtual) machine
- At execution time, Java Virtual Machine (JVM) has an interpreter that executes bytecode programs

© All Rights Reserved

An Example: Java



- Advantage of Java: codes run everywhere!
 - Compile on one machine and execute on any other machine
 - Bytecode is designed to favor compactness
 - Making it popular with web application
- Disadvantage of Java: virtual codes are slower than traditional native machine codes

© All Rights Reserved

takes the
byte code
→ does full
translation to
it
→ executable
file
→ run

Just-in-time Compilation (JIT)



- Combine the best of both worlds
 - At compile time, a compiler translates the source program into a virtual machine code
 - At execution time, a virtual machine **quickly** translates the virtual code into a native machine code, then the code is executed
- JIT compilation is both portable and fast
- JIT is used by Microsoft .Net framework and most current implementations of JVM

© All Rights Reserved



Source-to-Source Translators



- Compilers that translate from any programming language to any other programming language
- Examples:
 - f2c: translates from Fortran77 to C
 - Rose: translates multiple languages, including C++, C and Fortran
 - LLVM: translates any language supported by gcc
 - Cetus: translates sequential C programs into parallel C programs

ROSE@LLNL



© All Rights Reserved

CETUS

Integrated Development Environment (IDE)



- IDEs are software development tools where programmers can edit their codes, compile them and test their performance
 - Immediate feedback concerning syntax or semantic problems as the code being developed
- IDE examples:
 - Eclipse
 - NetBeans
 - Emacs



© All Rights Reserved

Summary



- A compiler is a computer program that translates a source code written in high-level language into assembly code
- Compilers perform the translation fully offline, no runtime overhead is involved
- During the translation, we expect compilers to preserve correctness, detect errors and perform optimizations
- Existing compilers in the market provide user-options for more flexibility

© All Rights Reserved

Lecture 2

Compiler Structure Spring 2018/2019

Instructor: Dr. Fahed Jubair
Computer Engineering Department
University of Jordan



The Topic of This Lecture



- Describe the compilation process in high-level
- Our goal is to see the big picture first before we dive into the details
- A lot of terminology will be introduced so make sure to keep up

© All Rights Reserved

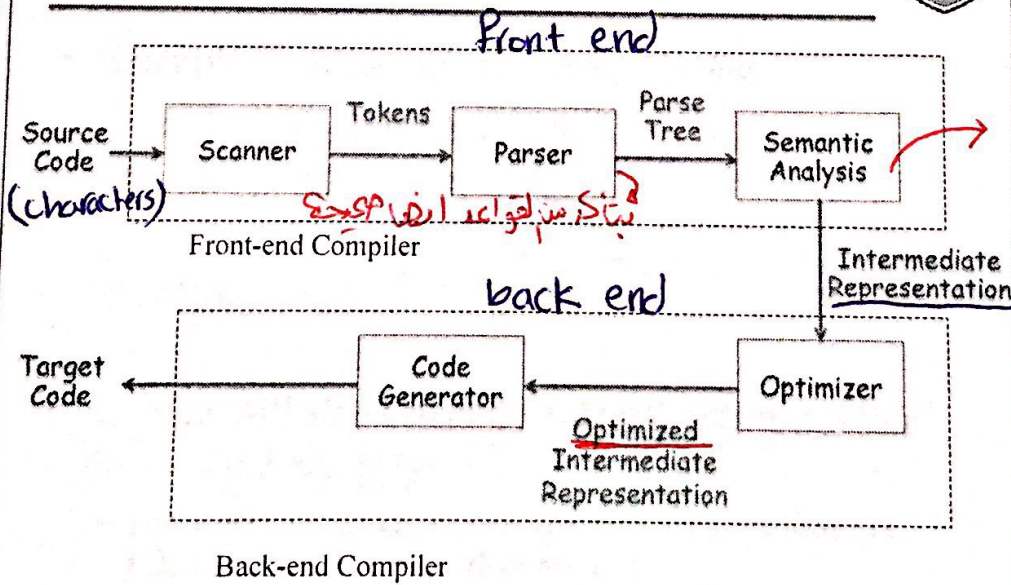
Compiler Structure



- The goal of the compilation process is to translate a source program into a target program
- To do so, the compiler naturally needs to:
 1. Understand the input program and ensure no errors
 2. Map the input program into an equivalent and optimized target program
- The first task is done by the front-end of a compiler
- The second task is done by the back-end of a compiler

© All Rights Reserved

Compiler Structure



no semantic error check

likes matrix lists trees

© All Rights Reserved

High level Picture of Scanner



Phase 1: Scanner

- Also called lexer
- A scanner reads the input program text character by character and transforms these characters into tokens
- A token defines a minimal syntactic unit in programming languages
 - Similar to a word in the English language

© All Rights Reserved

Tokens



- Example: Consider the following C code


```
if ( X >= 0 ) then Y = X ; else Y = - X ;
```

 The tokens are:
 'if', '(', 'X', '>=', '0', ')', 'then', 'Y', '=', '-', 'X',
 ';', 'else', 'Y', '=', '-', ';'
- Scanner will also detect all "illegal" substrings that do not form any token
 - But how can scanners recognize substrings that are tokens and substrings that are not?

© All Rights Reserved

tokens JS
type ال
Value.

ال token
لان يكون اشئ ال
معنى

هون ما أخذ اكان
زر f كان لاظم
ما الهم معنى أخذ
if كلمة (كلمة)
مجزوءة

قواعد
من خلاله بقدر
Scanner يعرف
tokens من
التي هي صحيحة ومن
لا .

Regular Expression

- A regular expression expresses a set of rules for describing valid tokens in a language can be formed
- Using regular expressions, a scanner can recognize all tokens for an input program text, as well as identify erroneous tokens
- We refer to a language that can be fully expressed by regular expressions as a regular language
 - Modern programming languages are regular languages

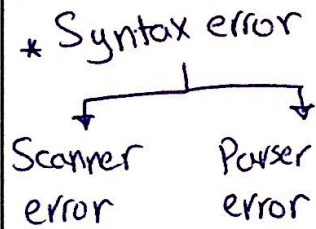
© All Rights Reserved

top level of
the parse
tree is
Program

Phase 2: Parser

- A parser reads the string of tokens returned by a scanner and performs the following two tasks:
 1. Confirm whether this string has a valid structure in the programming language or not
 2. Generate a tree representation, called the parse tree, of the input code structure
- The parser returns a syntax error if the code structure of the input program is invalid for the given programming language

© All Rights Reserved



Scanner →

Scanner ~~error~~ إذا عندني error يعني إذا tokens are valid

Parser →

كل tokens التي يفوتوا عليه لازم يكونوا مع لانهم تسمى قبل بار Scanner
- بناء من Structure.

Syntactic Structure



- A string of tokens is legal if it has a valid syntactic structure in the programming language
- But how can a parser validate the code structure of an input program?
 - For a human language such English or Arabic, it is easy, just check the grammar
 - Compilers do the same! They check the grammar of the programming language

© All Rights Reserved

Grammar



- Set of production or derivation rules that describe how to form strings in a language
- The English language has a grammar: a set of rules that describe how a *sentence*, i.e., a set of words, can be structured
- A programming language has a grammar: a set of rules that describe how a code statement, i.e., a set of tokens, can be structured

© All Rights Reserved

A Sentence in English



- In English, we have the following production rules
 - A "SENTENCE" can have the structure "SUBJECT VERB OBJECT"
 - A "SUBJECT" can have the structure "PRONOUN"
 - A "VERB" can have the structure "AUXILIARY"
 - An "OBJECT" can have the structure "ADJECTIVE"
 - A "PRONOUN" is "he | she | it | ..."
 - An "AUXILIARY" is "is | was | has | ..."
 - An "ADJECTIVE" is "big | small | ..."

- Exercise: use the above rules to validate the structure of the sentence "He is late"

© All Rights Reserved

An Assignment Statement In C



- In C, we have the following production rules
 - A "STATEMENT" can have the structure "ASSIGN_STATEMENT"
 - A "ASSIGN_STATEMENT" can have the structure "IDENTIFIER EQUAL EXPRESSION SCOLON"
 - An "IDENTIFIER" is any sequence of [a-z] characters
 - An "EQUAL" is "="
 - An "EXPRESSION" is "IDENTIFIER OPERATOR IDENTIFIER"
 - An "OPERATOR" is "+ | - | * | ..."
 - An "SCOLON" is ";

- Exercise: use the above rules to validate the structure of the sentence "x = x + y;"

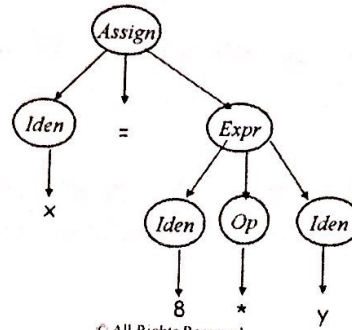
© All Rights Reserved

The top level of the parse tree is the program.

Parse Tree



- The parser produces a parse tree for a valid input program, which is a tree representation of the syntactic structure in the input code
- Parse trees are really intuitive – see the below example:



© All Rights Reserved.

So Far: Scanner and Parser



- Both scanning and parsing analyze the program text for syntax errors, i.e., they check the “structure”
 - The term syntax analysis is used to refer to both steps: scanning and parsing
- However, they do not check the “meaning”
 - Example: “X = Y + 1;” is syntactically correct but meaningless if Y is a string and X is an integer
- Therefore, the third and last step of the front-end compiler is to check the semantics, i.e., the “meaning” of the input program

© All Rights Reserved

Syntax error

Scanner error

Parse error

Phase 3: Semantic Analysis



- The purpose of semantic analysis is to

1. Check for semantic errors
2. Build a data structure for storing declared variables, called the symbol table
3. Convert the parse tree into another data structure, called the *intermediate representation (IR)*

© All Rights Reserved

مثال إذا جمع

x و y

$x \rightarrow \text{string}$

$y \rightarrow \text{int}$

($x = x + y$)

لـ Structure ما

نـ مستطاب

Semantic error لـ

أنه ما بيزيد اجمع
int مع string
دا خزنة بـ int.

ما يكون في خطا
بـ يكون مش
منطقية عند التنفيذ

Examples on Semantic Errors



1. Multiple declaration of the same variable within the same *scope*
2. A variable is not declared before it is used
3. A function call with the incorrect number or type of arguments
4. An algebraic expression performing operations with invalid types

© All Rights Reserved

كل د Symbols
 اللي بايكون عندي
 بجهتهم بار Symbol
 Table
 وبعدها مدغم معلوماتهم

Symbol Table



- A data structure maintained by the compiler for storing information about declared identifiers in the input program
- Examples of stored information in symbol tables:
 - Each variable's type and scope
 - Each function's return type and number and type of its arguments
 - Each class's name and relationships
 - etc

© All Rights Reserved

Symbol
 ↳ class name
 ↳ variable name
 ↳ function name

Intermediate Representation (IR)




- A popularly used term for describing the internal representation of the input program by the compiler
- The IR preserves the meaning of the input program
- Some compilers may use the parse tree as an IR
- Other IR formats are also available
 - For flexibility
 - The selection of the IR format significantly affects the design and implementation of the back-end compiler

© All Rights Reserved

* Front end → checks for semantic & syntax error
 ← بعد اذن 3 خطرات بلون برناجي اذ كيد هج وما فيه errors.

IR Example: Three-Address Code



- A popular IR in compilers is the three-address code
 - Each instruction can take at most three operands
 - Easy to map into machine (i.e., assembly) code
- Three-address code Example


```


      Label L1
      Add a, b ⇒ T
      SW T ⇒ c
      Add i, 1 ⇒ i
      BEQ i, 10, L1
      
```

© All Rights Reserved

~~Handwritten scribbles~~

Back end

Phase 4: Optimizer



- The purpose of this phase is to improve the translated program in some discernable way
 - Minimize execution time → زي مندا استبدل اهنوب و shift
 - Minimize memory footprint → او انخير ترتيب instructions
 - Minimize power consumption
 - etc
- Golden rule: optimization must preserve correctness, i.e., the meaning of the input program
- An optimization technique must guarantee that any changes this technique performs on the IR preserve correctness

© All Rights Reserved

Optimization Example: Dead Code Elimination



- Dead code is a code segment whose execution result does not affect the result of the program
- Example: in the below code example, the statements "b = a;" and "d = c + ..." are dead because they do not affect the final result

```
void main () {
    int a, b, c, d;

    a = 1;
    b = a;
    c = a + a * a + 100;
    d = c + c * c + 100;
    return c;
}
```

Write an optimized code for function *main*

Assuming that you want to develop a compiler analysis that eliminates dead code, in your opinion, how can this analysis guarantee correctness?

© All Rights Reserved

لو شطبت كودول
الكلتيم ما رح ياتو
عوي نتائج

Optimizations are Challenging



- Optimizing programs **correctly** and **efficiently** require compiler analyses to prove whether certain properties hold or not in the compiled programs
- This is challenging for many reasons, some of which are:
 - Compilers do the translation offline, where the knowledge about runtime behavior is incomplete
 - Some optimizations can be machine-sensitive, i.e., different machines may have different performances for the same program
 - Some optimizations can also be application-sensitive, different programs may have different performance for the same machine

© All Rights Reserved

يعني مثلا
لو بالبرنامج عندي
(C9n)
بمرحلة الكومبايلر
ما بعرف شو لقيتاه اللي
ممكن يدخلها ليوزر
فلازم الoptimization
يكون له value لكل كلاس
المترقعة

But Doing Optimizations is What Gets You Paid



- Compilers that are “**smart**” will always be in demand
- For example, consider the following issue:
 - New architectures with new performance considerations are coming out everyday
 - Therefore, old codes may become slow and obsolete
 - Hiring programmers to rewrite old codes is expensive and time consuming
 - Cheaper solution: perform compiler optimizations on old codes to make them run faster!

© All Rights Reserved

Phase 5: Code Generator



- The code generator maps the IR code into the target machine code
 - E.g., MIPS or x86 assembly code
- Main tasks of code generation
 - ① Instruction selection: which machine instructions to use
 - ② Instruction scheduling: which order of machine instructions to use
 - ③ Register allocations: map variables to physical registers
 - Input programs (as well as front-end compilers) assume unlimited memory, for simplicity
 - Back-end compilers deal with reality: machines have finite resources

© All Rights Reserved

Lecture 3

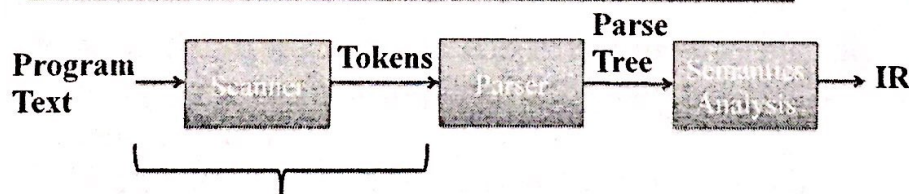
Scanners

Spring 2018/2019

Instructor: Dr. Fahed Jubair
 Computer Engineering Department
 University of Jordan



Scanner Role



- Scanner (or lexer) role is to convert the input stream of characters into a string of tokens
- Tokens define the minimal syntactic unit in a program
- A token has a type and a value:
 - E.g., '5' is an integer with value 5
 - E.g., 'x' is an identifier with value x
 - E.g., '=' is an operator with value =

© All Rights Reserved

Basic Questions For Scanners



1. How tokens are defined?
 → Regular Expressions
2. How tokens are recognized for a stream of character?
3. How scanners are coded?

© All Rights Reserved

Regular Expressions



- A regular expression t describes the rules of which all string patterns for a *language* L with an alphabet Σ can be formed
 - This language is said to be a regular language and is denoted by $L(t)$
- An alphabet Σ defines a finite set of characters that all strings in a language may contain
 - E.g., integers have $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

© All Rights Reserved

Basic Regular Languages



- If a single character $x \in \Sigma$, then x is a regular expression denoting the regular language $\{“x”\}$
- The empty string $\epsilon = “ ”$ is a regular expression and $\{\epsilon\}$ is a regular language with one member: ϵ
- The empty set ϕ is the regular language that contains no string members

© All Rights Reserved

Basic Operations For Building Regular Expressions



- If s and t are regular expressions, then
 - The concatenation (st) is also a regular expression
 - The alternation $(s|t)$ is also a regular expression
 - The kleene closure s^* is also a regular expression, where $s^* = \epsilon | s | ss | sss | ssss | \dots$
 - i.e., s^* denotes zero or more occurrences of s
 - The positive closure s^+ is also a regular expression, where $s^+ = s | ss | sss | ssss | \dots$
 - i.e., s^+ denotes one or more occurrences of s
 - Note that $s^+ = s s^*$

© All Rights Reserved

بجانب مع بعض
OR
يعني ممكن اخذها,
Zero or more
S*
S+ → ممكن اخذها
one or more

Tokens and Regular Expressions



We will now describe the regular expressions of the following tokens:

- ▣ Literals
- ▣ Identifiers
- ▣ Comments
- ▣ Reserved words
- ▣ Operators
- ▣ Punctuations

} Often found in programming languages

© All Rights Reserved

Strings / الأرقام Literals



- let D be a single digit integer, I be an integer number, and E be a real number
- We can describe the regular expressions D , I and E as follows:

$$D = (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) = [0 - 9]$$

$$I = D^+ = [0 - 9]^+ \text{ integer numbers}$$

$$E = D^+ . D^+ = [0 - 9]^+ . [0 - 9]^+ \text{ Float numbers}$$

- Exercise: write a regular expression that describes integers with no leading 0s, i.e., numbers such as 01 and 001 are not allowed, but numbers such as 0, 10, and 201 are allowed

© All Rights Reserved

عنه زكته
 $[0-9]^+ . [0-9]^+$
 • 3 ✓

Solution For Exercises -

~~...~~ $[1-9][0-9]^*$ | 0

هيك اول digit
 اصورها تكون مني zero
 اناهي مني
 بي digit واحد

عشان انا به
 بس رقم zero

Exercise ① $\varnothing [A-Z] ([a-z] | [A-Z] | [0-9])^*$

Exercise ② $\varnothing ([a-z] | [A-Z] | [0-9])^* \varnothing \varnothing ([a-z] | [A-Z] | [0-9])^*$

Exercise ③ $\varnothing ([a-z] | [A-Z] | [0-9])^* \varnothing$

1/20/19

Identifiers

- Let IDEN be any string that has any of the following characters: a-z, A-Z, 0-9.

$IDEN = ([a-z] | [A-Z] | [0-9])^+$

- Exercise: rewrite the regular expression of IDEN so that the first character in the identifier string can only be a capital letter
- Exercise: rewrite the regular expression of IDEN so that it must have the substring '00'
- Exercise: rewrite the regular expression of IDEN so that it must end with the character '0'

© All Rights Reserved

Comments

- Different programming languages have different format for comments
- As an example, Assume that a comment must start and end with ##
 - The character “#” may appear inside the comment
- For a given alphabet Σ , if $x \in \Sigma$, then we define $Not(x)$ to be the set of all characters in Σ except x
- The following regular expression describes comments:

$COMMENT = ## ((# | \epsilon) Not(\#))^* ##$

© All Rights Reserved

\rightarrow Format for comments.

Valid comment

جود body ما بزب
 احب ## انا
 Comment) نفاذ

Operators (+ | - | * | ÷ | —)
keywords (if | for | while)

1/20/19

Reserved Words, Operators and Punctuations



- Generally, reserved words, punctuations and operators have unique strings

→ Their regular expressions are straightforward

• Examples

- IF = (if)
- GT = (>)
- GEQ = (>=)
- LPARN = (())
- RPARN = (')

Handwritten notes in Arabic: "هذه أمثلة على تعبيرات Reguler Expressions" and "نضع ' ' لتمييزها عن باقي الألفاظ".

We add ' ' to distinguish an input character from meta-characters

© All Rights Reserved

Exercises



- For each of the following regular expressions, write four possible string patterns:

- ① ▪ $r(0|(1|2)[0-9]|3[0-2])$
- ② ▪ $(00|99)^*(2|3)^+$
- ③ ▪ $(a|(bc)^*d)^+$
- ④ ▪ $(a\text{Not}(a))^*aaa$

© All Rights Reserved

Exercise ① $(0|[1-9][0-9]^*) \cdot ([0-9]^*[1-9]|0)$

Exercise ② $(1|0)^* \circ (1|0)^* \circ (1|0)^*$

Exercise ③ $(0|1|1)^+ (0|1)$

1/20/19

Exercises



- ①. Write a regular expression that defines a C-like, fixed-decimal literal with no superfluous leading or trailing zeros. That is, 0.0, 123.01, and 123005.0 are legal, but 00.0, 001.000, and 002345.1000 are illegal.
- ②. Write a regular expression that describes all strings of 0's and 1's with at least two 0's. For example, 0100, 01110, and 00 are legal but 1, 10, and 1011 are not.
- ③. Write a regular expression that describes all strings of 0's and 1's such that two consecutive 0's are not allowed. For example, 0, 1, 0110, and 1010 are legal but 00, 0010, and 101001 are not.

© All Rights Reserved

Basic Questions For Scanners



1. How tokens are defined?
⇒ Regular Expressions
2. How tokens are recognized for a stream of character?
⇒ Finite Automata
1. How scanners are coded?

© All Rights Reserved

Set of states and transitions

Finite Automata (FA)



- Assume you are given a string S and a regular expression R that expresses a token class T
- A finite automaton (FA) is a finite state machine that accepts S if $S \in L(R)$, and the token class of S would be T . Otherwise, S is rejected
- By building proper finite automata, a scanner can recognize tokens for a stream of characters, as well as identify erroneous tokens

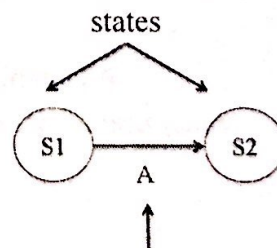
© All Rights Reserved

Accept the token if valid
Reject the token if not valid

Finite State Machines



- First, let us review finite state machines (FSMs)
- A FSM consists of sets and transitions that are event-triggered
 - In scanners, an event occurs when a new input character is consumed
- FSMs memorize previous events (how?)



A transition from state $S1$ to $S2$ occurs when event A is triggered

© All Rights Reserved

Finite Automata (FA)



- A FA consists of
 - A finite set of states S
 - A start state n
 - A finite alphabet Σ
 - A set of accepting states G , where $G \subseteq S$
 - A set of transitions $s_i \xrightarrow{a} s_j$, where $a \in \Sigma$, $s_i, s_j \in S$
- Basically, a transition occurs in the FA when a new input character is consumed
- If the input character does not correspond to any transition, then the FA advances to the error state

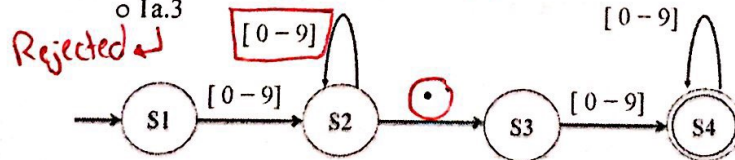
Consists of
 Start (state)
 - transitions
 - accept state

An FA Example



- The below FA is generated for the regular expression:
 $E = [0-9]^+ . [0-9]^+$
 - Specify S , n , Σ , G and all transitions in the FA
 - Does the FA accept or reject the following inputs (show your answer)?

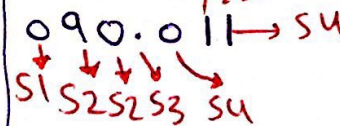
- o 090.011 Accepted token
- o 10. → Rejected token
- o 1a.3 Rejected



لازم عثمان يكون
 Valid token
 ينطقه باخر State

Start state →

الذي يكون داخل على اليمين
 من دلائل (S1 : مثال)



Accepted because
 it finished
 with S4

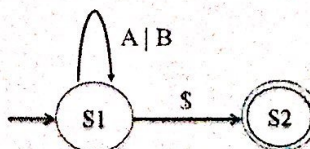
alphabet $\Sigma = \{0, 1, 2, \dots, 9, .\}$

Another FA Example



- The below FA is generated for the regular expression:
 $E = (A|B)^* \$$
 - Specify S, n, Σ, G and all transitions in the FA
 - Does the FA accept or reject the following inputs (show your answer) ?

- A\$ ✓
- \$ ✓
- ABAB\$ ✓
- A\$B ✗
- ABB\$\$



Accept and gives 2 tokens (A\$B\$)

© All Rights Reserved.

Accept on A\$ and \$
 Reject on B\$

①
 ②
 \$)

Finite Automata Types



- Two types of FA:
 - Deterministic Finite Automata (DFA)
 - Non-deterministic Finite Automata (NFA)
- In order to explain each type, we first need to introduce the concept of ϵ -Transitions

© All Rights Reserved

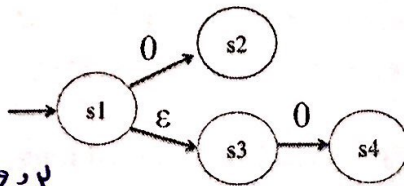
Unconditional transition (no match)

ε-Transitions



- The ϵ -transition $s \xrightarrow{\epsilon} w$ allows moving from state s to state w without reading any alphabet input
 - i.e., ϵ -transition is unconditional move
- ϵ -transitions allow states in a FA to have multiples moves for the same input

بھار ملناں لو
اچانی صہض
صہکن پروج
عی S2 آو



© All Rights Reserved.

(ε) ورجد پیر
عی S4
پروج عی S3 من خلد ل

DFA and NFA



- A deterministic finite automata (DFA) is a FA where all transition are unique, i.e., if $s \xrightarrow{c} w$ and $s \xrightarrow{c} z$, then $w = z$
- DFAs do not have ϵ -transitions
- A FA where states can have multiple moves for the same input is called a non-deterministic finite automata (NFA)
- Unlike DFAs, NFAs can have ϵ -transitions
- Note that NFA is a generalization of DFA, i.e., every DFA is also an NFA

© All Rights Reserved.

Which FA to Use?



- Which FA: DFA or NFA should be used for recognizing tokens?
 - In general, either can be used because both are equivalent in terms of computation power
 - The main difference is their implementation
 - DFAs are usually easier to implement and faster to execute due to their transitions being unique
 - NFAs are generally smaller, i.e., less memory is needed
 - We will learn how to construct both NFAs and DFAs for regular expressions

© All Rights Reserved

Constructing FA For Regular Expressions



- Constructing FA for regular expressions is not trivial
- For example, try guessing the FA for the following regular expression: $(a | (bc)^* d)$
- We need an algorithmic approach for constructing FAs from regular expressions

© All Rights Reserved

Regular expression \rightarrow NFA \rightarrow DFA.

1/20/19

Finite Automata Construction

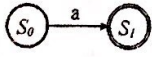
1. Thompson's construction
 - An algorithm that generates an NFA for a regular expression
2. Subset construction
 - An algorithm that converts an NFA into a DFA

© All Rights Reserved

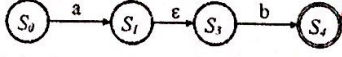
Thompson's Construction

Key idea is simple

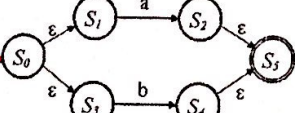
- Draw NFA pattern for each symbol & each operator
- Join them with ϵ moves in precedence order



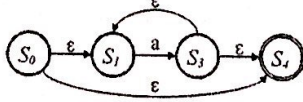
NFA for a



NFA for ab



NFA for a | b



NFA for a*

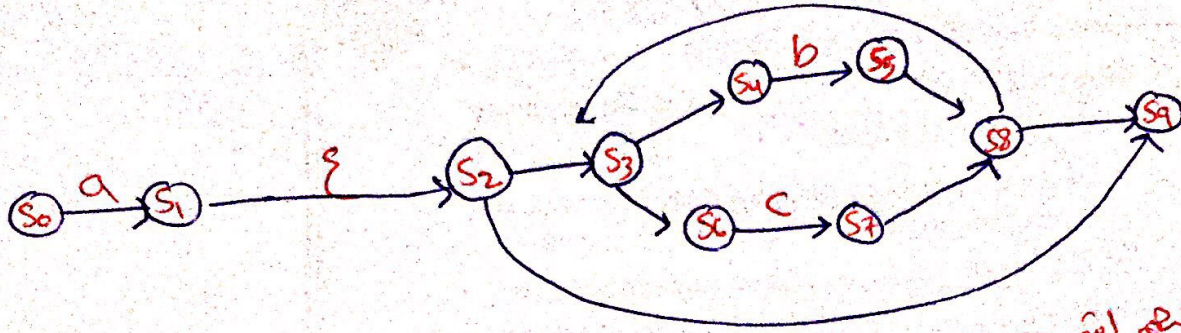
© All Rights Reserved

Concatenation
 لان لازم ايسر
 ب ا بعد ب
 عثمان اوصى س4

هاي OR لان
 بقدر ايسر من قوي
 اوصى كى كا اوصى
 S5

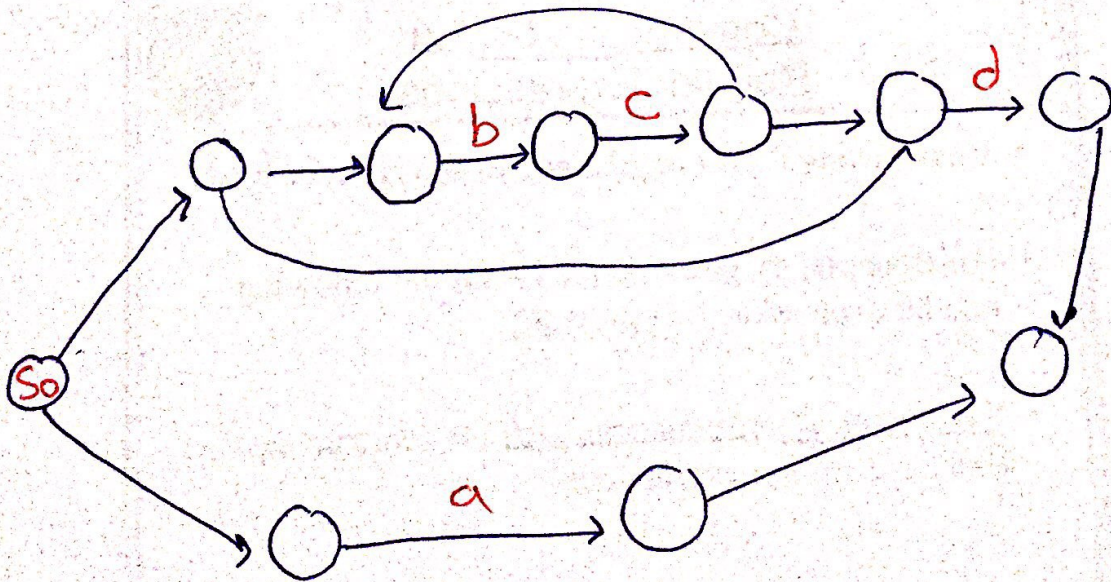
$$R = a(bc)^*$$

علائق نرسه اول اشي يبلش من (b|c)
 من برسماي او default الی قیل
 بعدیه ضبیت * بعدیه a

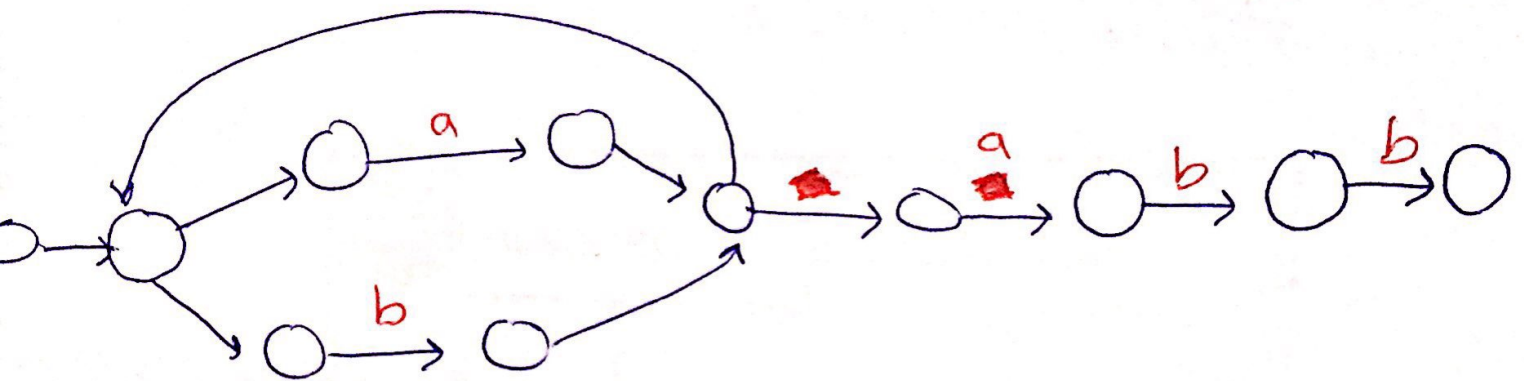


الی ما علیهم اشي عالمهم یعنی ع

$$* R = (a|(bc)^*d)$$



$$R = (a|b)^+ abb$$

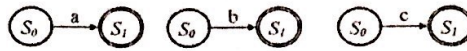


Thompson's Construction Example

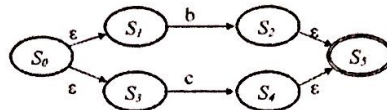


Let's try: $a(b|c)^*$

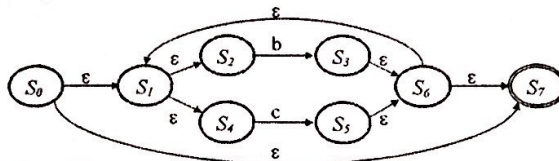
1. a, b, & c



2. $b|c$



3. $(b|c)^*$

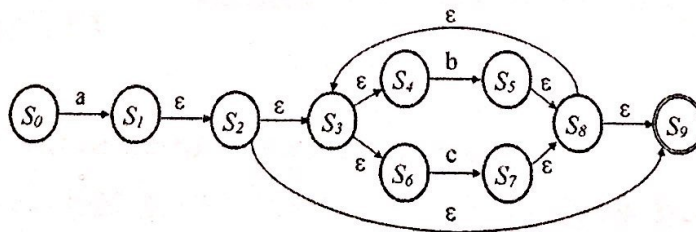


© All Rights Reserved

Thompson's Construction Example (cont')



4. $a(b|c)^*$



© All Rights Reserved

Exercise



- Use Thompson's construction to determine the NFA for the following regular expressions
 - $(a | b)^* abb$
 - $r [0 - 9]^+$
 - $(a | (bc)^* d)$

© All Rights Reserved

Subset Construction



- Transforms an NFA N into an equivalent DFA D
- The algorithm associates each state of D with a set of states of N
- The algorithm uses two key functions:
 - ϵ -Closure(x): returns the set of states reachable from x by ϵ
 - Move(X, a): returns the set of states reachable from X by a , where $a \in \Sigma$ and X is a set of states

© All Rights Reserved

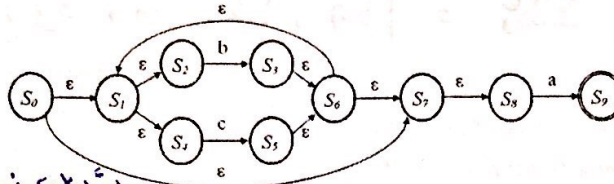
برجع کی لائسم
ادع الی بتوفہین
X د

برجع کی لائسم
الی بتوفہین د X
باستخدام a

ε-Closure Function



- The ε-closure function determines, for a state x , all states that can be reached via ε-transitions in the NFA
- Example:



دائماً لازم نلاحظ نفس اول الشيء

$$\epsilon\text{-closure}(S_0) = \{S_0, S_1, S_2, S_4, S_7, S_8\}$$

$$\epsilon\text{-closure}(S_3) = \{S_3, S_6, S_1, S_2, S_4, S_7, S_8\}$$

$$\epsilon\text{-closure}(S_4) = \{S_4\}$$

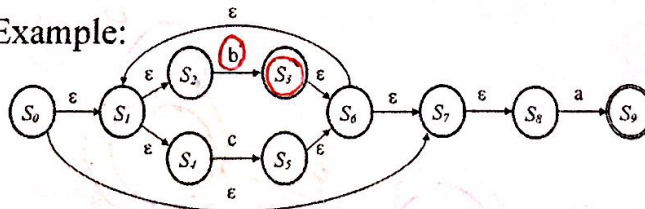
© All Rights Reserved

صرون عن S6
نصي S6

Move (X , a) Function



- The Move(X , a) function determines, for a set of states X, all states that can be reached via "a-transitions"
- Example:



$$\text{Move}(\{S_3, S_6, S_1, S_2, S_4, S_7, S_8\}, b) = \{S_3\}$$

$$\text{Move}(\{S_3, S_6, S_1, S_2, S_4, S_7, S_8\}, c) = \{S_5\}$$

$$\text{Move}(\{S_3, S_6, S_1, S_2, S_4, S_7, S_8\}, a) = \{S_9\}$$

© All Rights Reserved

بدور عوا ستم اسميه
ط و بيثوف سكو
الي بيخول عليها
وستو لتابع و الجواب
بيكون لتابع

كازم التسم الي
بدي ايان ل input

تاعه يكون من ضمنه
arguments تاعون ادر move عشان اقدر اطلع جواب

أول خطوة :

بحسب d_0

$d_0 = \epsilon\text{-closure}(S_0)$

② $\text{Move}(d_0, a) = \{S_1\}$

$\text{Move}(d_0, b) = \emptyset$

$\text{Move}(d_0, c) = \emptyset$

③ $\epsilon\text{-closure}(S_1) =$

$\{S_1, S_2, S_3, S_4, S_6, S_8\}$

$= d_1$ (new state)

Subset Construction Algorithm



Input: NFA N
Output: DFA D

Note that N and D will have the same Σ

```

 $d_0 \leftarrow \epsilon\text{-closure}(\{N.start\_state\})$ 
 $D.states \leftarrow \{d_0\}$ 
 $WorkList \leftarrow \{d_0\}$ 
while ( $WorkList \neq \emptyset$ )
  select and remove  $s$  from  $W$ 
  for each  $\alpha \in \Sigma$ 
     $t \leftarrow \epsilon\text{-closure}(\text{Move}(s, \alpha))$ 
    add  $s \xrightarrow{\alpha} t$  to  $D.transitions$ 
    if ( $t \notin D.states$ ) then
      add  $t$  to  $D.states$ 
      add  $t$  to  $WorkList$ 
  
```

Compute d_0 : the start state of D

For each character in alphabet

Associate a set of states in N with a set in D

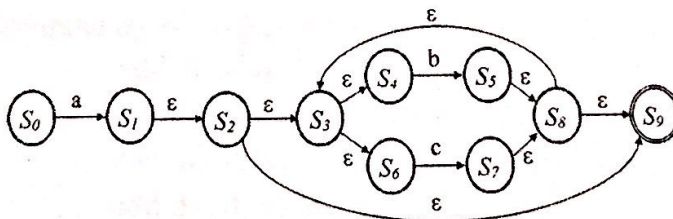
Iterate till no more states are added

© All Rights Reserved

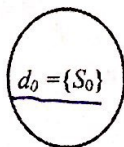
Subset Construction Example



- Let us try the following NFA



- Compute $d_0 \leftarrow \epsilon\text{-closure}(\{S_0\})$



الكل كامل
على لورقة

© All Rights Reserved

① $d_0 = \epsilon \text{ closure}(S_0) = \{S_0\}$

② $\text{Move}(d_0, a) = \{S_1\} \rightarrow \epsilon\text{-closure} = \{S_1, S_2, S_3, S_4, S_6, S_9\}$
 $\text{move}(d_0, b) = \emptyset$
 $\text{move}(d_0, c) = \emptyset$

d_1
(new state)

③ $\text{Move}(d_1, a) = \emptyset$
 $\text{move}(d_1, b) = \{S_5\} \xrightarrow{\epsilon\text{-closure}} = \{S_5, S_8, S_9, S_3, S_4, S_6\}$
 $\text{move}(d_1, c) = \{S_7\} \xrightarrow{\epsilon\text{-closure}} = \{S_7, S_8, S_9, S_3, S_4, S_6\}$

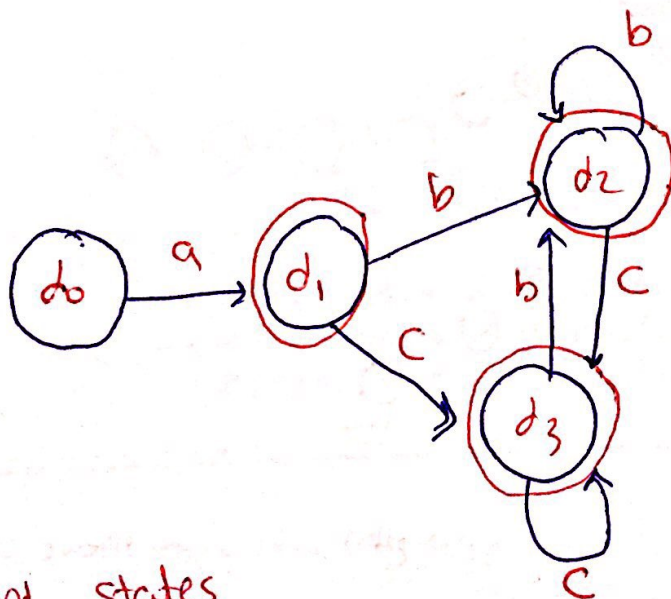
d_2
 d_3

④ $\text{Move}(d_2, a) = \emptyset$
 $\text{move}(d_2, b) = \{S_5\} \rightarrow \{S_5, S_8, S_9, S_3, S_4, S_6\}$ d_2
 $\text{move}(d_2, c) = \{S_7\} \rightarrow \{S_7, S_8, S_9, S_3, S_4, S_6\}$ d_3

⑤ $\text{move}(d_3, a) = \emptyset$
 $\text{move}(d_3, b) = \{S_5\} \xrightarrow{\epsilon\text{-closure}} \{S_5, S_8, S_9, S_3, S_4, S_6\}$ d_2
 $\text{move}(d_3, c) = \{S_7\} \xrightarrow{\epsilon\text{-closure}} \{S_7, S_8, S_9, S_3, S_4, S_6\}$ d_3

Worklist

- ~~d_0~~
- ~~d_1~~
- ~~d_2~~
- ~~d_3~~

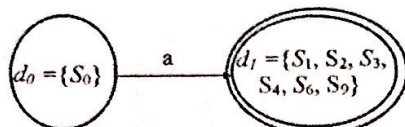


$d_2, d_1, d_3 \rightarrow$ accept states

Subset Construction Example (cont.)



2. Compute $d_1 \leftarrow \varepsilon\text{-closure}(\text{Move}(d_0, a))$
 add $d_0 \xrightarrow{a} d_1$
 add d_1 to WorkList

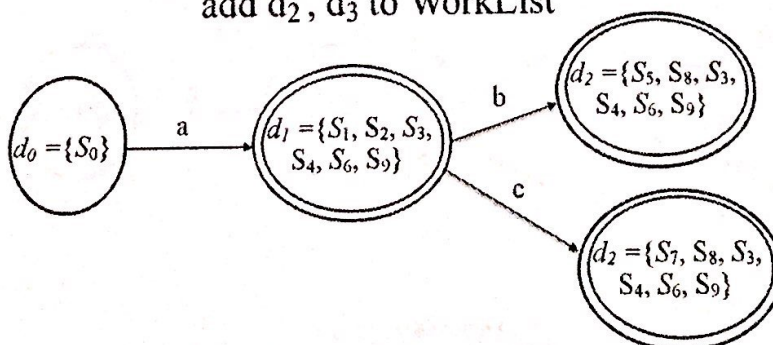


© All Rights Reserved

Subset Construction Example (cont.)



3. Compute $d_2 \leftarrow \varepsilon\text{-closure}(\text{Move}(d_1, b))$
 add $d_1 \xrightarrow{b} d_2$
 $d_3 \leftarrow \varepsilon\text{-closure}(\text{Move}(d_1, c))$
 add $d_1 \xrightarrow{c} d_3$
 add d_2, d_3 to WorkList

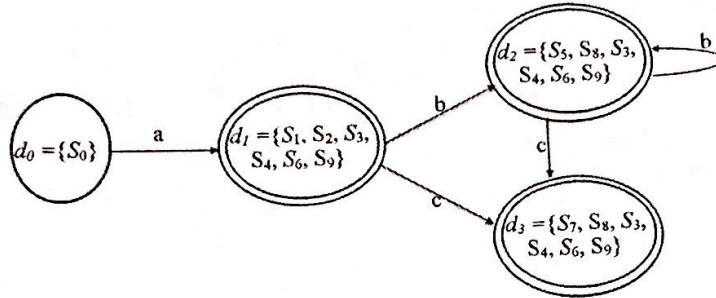


© All Rights Reserved

Subset Construction Example (cont.)



4. Compute $d_4 \leftarrow \epsilon\text{-closure}(\text{Move}(d_2, b))$
 but $d_2 = d_4$, add $d_2 \xrightarrow{b} d_2$
 $d_5 \leftarrow \epsilon\text{-closure}(\text{Move}(d_2, c))$
 but $d_3 = d_5$, add $d_2 \xrightarrow{c} d_3$

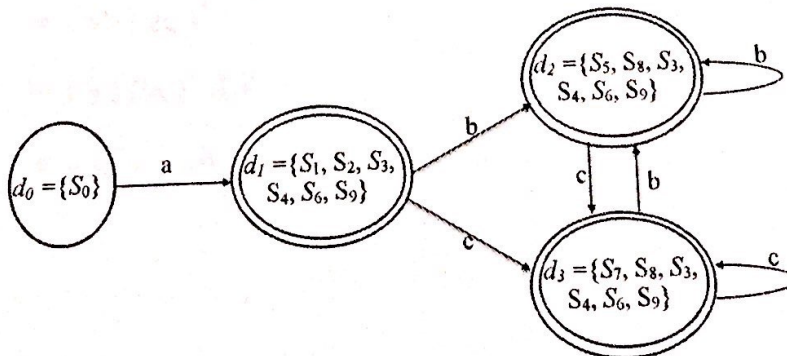


© All Rights Reserved

Subset Construction Example (cont.)



5. Compute $d_6 \leftarrow \epsilon\text{-closure}(\text{Move}(d_3, b))$
 but $d_2 = d_6$, add $d_3 \xrightarrow{b} d_2$
 $d_7 \leftarrow \epsilon\text{-closure}(\text{Move}(d_3, c))$
 but $d_3 = d_7$, add $d_3 \xrightarrow{c} d_3$

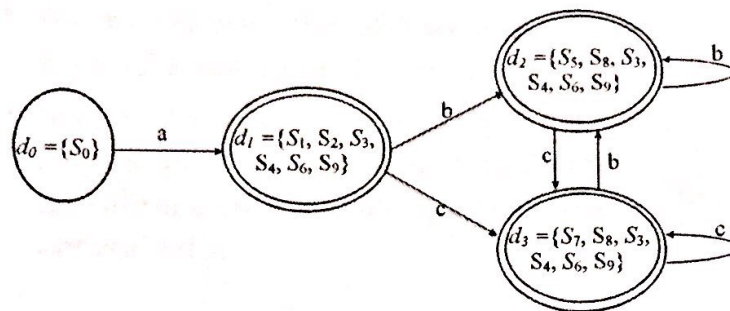


© All Rights Reserved

Subset Construction Example (cont.)



6. WorkList is empty, terminate the algorithm



© All Rights Reserved

Exercises



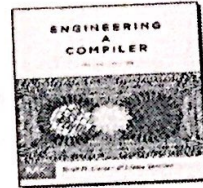
- Construct a DFA for the following regular expressions:
 - $(ab \mid ac)^*$
 - $(a \mid (bc)^* d)$
 - $ab^*c \mid abc^*$

© All Rights Reserved

Bonus: DFA Minimization



- DFAs generated with the Subset construction are not necessarily optimal
- As one example, the DFA we obtained in slide 42 is not optimal (why?)
- Section 2.4.4 introduce the Hopcroft's Algorithm: a minimization algorithm to transform a DFA into an equivalent but optimal DFA



© All Rights Reserved

Basic Questions For Scanners



1. How tokens are defined?

⇒ Regular Expressions

2. How tokens are recognized for a stream of character?

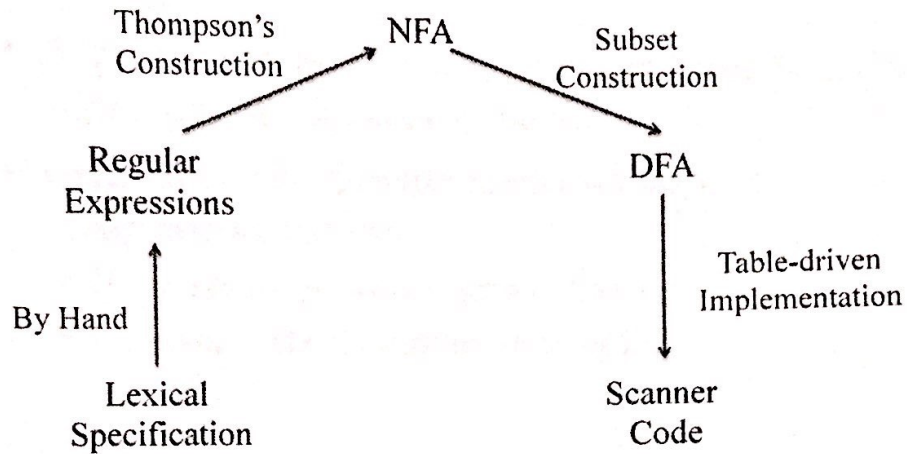
⇒ Finite Automata

1. How scanners are coded?

- ⇒
1. Table-driven implementation
 2. Automatic scanner generators

© All Rights Reserved

Scanner Generation Cycle



© All Rights Reserved

Coding The DFA



- Multiple methods have been used to implement scanners
- We will consider the table-driven form, the more common method for building scanners
- Table-driven scanners take advantage of *transition diagrams*: a tabular representation of FA

© All Rights Reserved

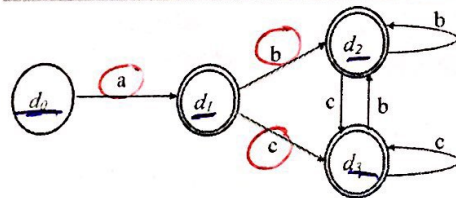
Transition Table



- A transition table T is a 2D array indexed by a FA state s and an alphabet symbol c
- Each entry $T[s,c]$ in the transition table is computed as follows:
 - If the transition $s \xrightarrow{c} w$ exists, then $T[s,c] = w$
 - Otherwise, $T[s,c]$ contains an error flag

© All Rights Reserved

Transition Table Example



By default, blank entries have error flags

Alphabet Characters

States	a	b	c
d_0	d_1		
d_1		d_2	d_3
d_2		d_2	d_3
d_3		d_2	d_3

© All Rights Reserved

اطریق لغافین
یعنی error

من تقاطع بین d_0
و d_1 از a کتبه d_1
و اجتناباً برود به
 d_1

destination
من تقاطع بین State و alphabet بعد از

Table-Driven Implementation



- Direct and simple interpretation of a FA's transition Table T

```

/* Assume CurrentChar contains the first character to be scanned */
State ← StartState
while true do
  NextState ← T[State, CurrentChar]
  if NextState = error
  then break
  State ← NextState
  CurrentChar ← READ()
if State ∈ AcceptingStates
then /* Return or process the valid token */
else /* Signal a lexical error */

```

© All Rights Reserved

Automatic Scanner Generators



Regular
Expressions



Scanner
Code

- Software tools that automatically generates scanners' codes
- As an input, programmers only need to specify regular expressions (usually written inside a text file)
- Internally, these tools will do the transformations we did manually before:

Regular Expressions → NFA → DFA → scanner code

- Two popular scanner generators:

- Lex codes are generated in C/C++
- ANTLR codes are generated in Java

© All Rights Reserved

ANTLR



- <http://www.antlr.org/>
- Very popular parser and scanner generator tool
- As an input, ANTLR reads a text file that contains the language *grammar*, as well as the regular expressions of the language tokens
 - We will describe grammars when we study parsers
- ANTLR has two key components:
 1. ANTLR tool: converts the grammar into a scanner and a parser
 2. ANTLR runtime: set of classes and methods needed when compiling and running the scanner and parser codes
- We will use ANTLR v4 for our course project

© All Rights Reserved

ANTLR Input Example



```
// Define a grammar in a file called Hello.g4
grammar Hello;
r : 'hello' ID ; // match keyword hello followed by an identifier
ID : [a-z]+ ; // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

© All Rights Reserved

ANTLR Tool Output



- Let us run the ANTLR tool on the grammar example
\$ java -jar antlr-4.5.3-complete.jar Hello.g4

- The ANTLR tool generates the following outputs:

Scanner code

- HelloLexer.java
 - HelloParser.java
 - Hello.tokens
 - HelloListener.java
 - HelloLexer.tokens
 - HelloBaseListener.java
- Read more from the ANTLR v4 Book
<https://pragprog.com/book/tpantlr2/the-definitive-antlr-4-reference>

© All Rights Reserved.

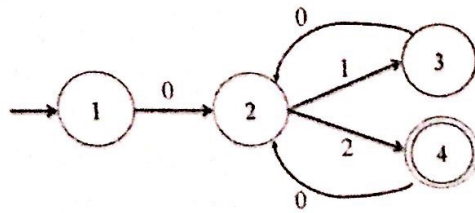
Summary



- Scanning (or lexical analysis) is the first step in the compilation process
- Scanners convert the input program text into a string of tokens
- Tokens define the minimum syntactical unit in programming languages
- Scanners take advantage of the concepts of regular expressions and finite automata in its implementation
- Automatic software tools for generating source codes of scanners are available

© All Rights Reserved

Exercise



Which of the following strings is accepted by the above DFA

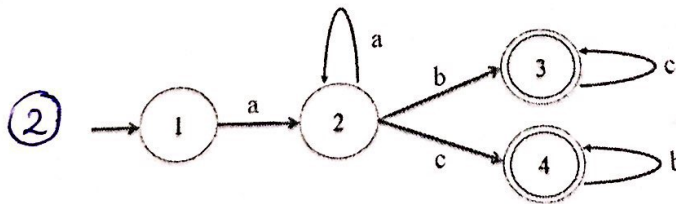
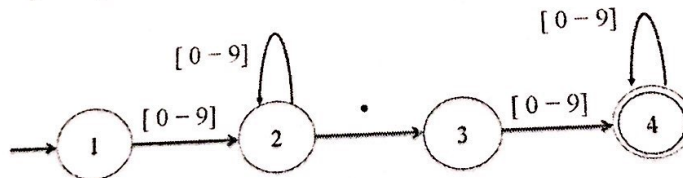
- 0202
- 01010
- 0102012
- 020102

© All Rights Reserved

Exercise



- Specify the transition tables for the following DFAs



All Rights Reserved

	a	b	c
S ₁	S ₂		
S ₂	S ₂	S ₃	S ₄
S ₃			S ₃
S ₄		S ₄	

② regular expression

هذا طلب صفي



Lecture 4

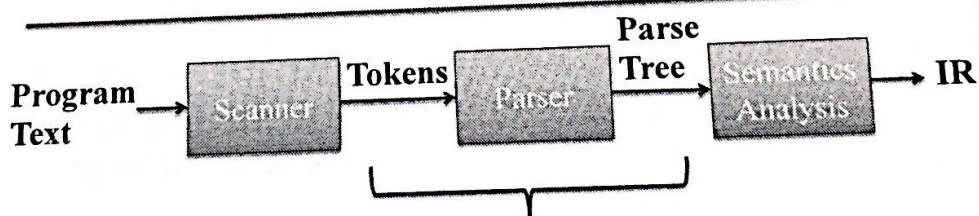
Introduction To Parsing

Spring 2018/2019

Instructor: Dr. Fahed Jubair
 Computer Engineering Department
 University of Jordan



The Parser Role



- The parser role is to
 1. Determine whether or not a string of tokens is a syntactically valid sentence in a programming language
 2. Build the parse tree, a tree representation that describes the code structure of the input program

© All Rights Reserved

grammars, *قواعد*

Language $\rightarrow L$ String of tokens $\rightarrow N$
 Grammar $\rightarrow G$

What is Parsing?



- For a given language L that has a grammar G a string of tokens N is a valid sentence in L if there is a sequence of derivation steps that derives N using the production rules of G
- Parsing is essentially the process of discovering a derivation for some sentence in a language

→ Process of discovering a parse tree

© All Rights Reserved

Roadmap



- First, we will study context-free grammars: a mathematical model for describing syntax in programming languages
- Second, we will study top-down parsing: an algorithm for testing the membership of sentences in a language using the rules of a context-free grammar
 - We will also cover how to write the code of top-down parsers

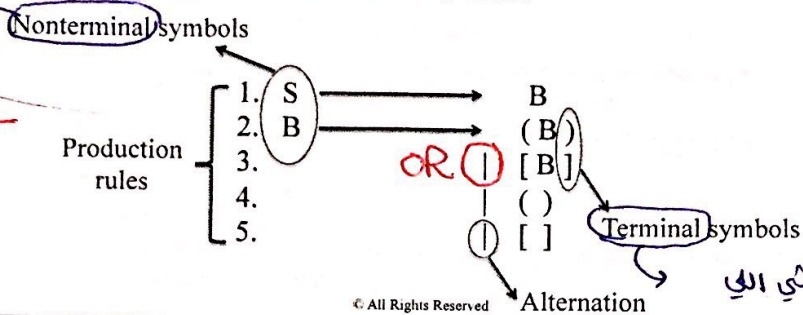
© All Rights Reserved

بكون عندي عالشان
 Symbol
 بالسا واه
 OMC

Context-Free Grammar (CFG)



- A set of **production rules** that **abstractly** describes how strings are derived in a language
- For example, the following grammar describes how strings of brackets can be formed



لاشي الي عالشان
 (بديكم تعريضا)
 شرحه
 S هون هي لبدارة وبتعطي
 B , B عكس تعطي
 حدة من كياران

Tokens ≡
 Terminals

(B) OR [B] OR () OR []

عالمين
 (ما بتعريفوا)

Deriving Sentences with A CFG



- S is called the start symbol because its where the derivation starts
- Let us try deriving the string "[()]"

1.	S	→	B
2.	B	→	(B)
3.		→	[B]
4.		→	()
5.		→	[]

S ⇒ B Start with rule 1
 ⇒ [B] Use rule 3
 ⇒ [[B]] Use rule 3
 ⇒ [[()]] Use rule 4

Components of A CFG



- Formally, a context-free grammar G is a quadruple (T, NT, S, P) where:
 - T** is the set of terminals
 - Terminals are basically the syntactic categories returned by the scanner
 - NT** is the set of nonterminals
 - Nonterminals are syntactic variables introduced to provide abstraction and structure in the productions
 - S** is the non-terminal designated as the start symbol
 - P** is the set of productions in G
 - Each rule in P has the form $NT \rightarrow (T \cup NT)^+$; that is, it replaces a single nonterminal with a string of one or more grammar symbols

© All Rights Reserved

CFG Examples



Specify the terminals and the nonterminals for each grammar

1.	E	→	<u>num</u> Op num Etail \$
2.	Etail	→	Op num Etail
3.			ε
4.	Op	→	⊕ token
5.			⊖ token

~~Terminals~~ Terminals

1.	S	→	(Arg_list)
2.	Arg_list	→	id Arg_Tail
3.	Arg_Tail	→	, id Arg_Tail
4.			ε

© All Rights Reserved

Non terminals

Non terminals

Derivation Process



- A derivation consists of a series of rewrite steps

$$S \Rightarrow Y_1 \Rightarrow Y_2 \Rightarrow \dots \Rightarrow Y_{n-1} \Rightarrow Y_n \Rightarrow N$$

1. The derivation always starts with the start symbol S
2. To get Y_i from Y_{i-1} , expand some nonterminal $A \in Y_{i-1}$ by using production rule $A \rightarrow \alpha$
3. Repeat (2) until there are no terminals
 - The derivation terminates with N : a valid sentence in the language $L(G)$

© All Rights Reserved

Terminology for Derivation



- A sentential form is a string of terminal & nonterminal symbols that is a valid step in some derivation
- The derivation $S \Rightarrow^* N$ denotes the start symbol S derives the sentence N in zero or more steps
- The derivation $S \Rightarrow^+ N$ denotes the start symbol S derives the sentence N in one or more steps

© All Rights Reserved

Parse Tree



- Parse tree: a directed graph that represents a derivation

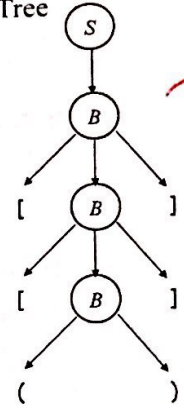
Grammar

1.	S	→	B
2.	B	→	(B)
3.			[B]
4.			()
5.			[]

$$S \Rightarrow^* [[()]]$$

Rule	Sentential Form
—	S
1	B
3	[B]
3	[[B]]
4	[[()]]

Parse Tree



الذي جوا لاندرة
 non terminal
 ليون
 terminals
 ليون

© All Rights Reserved

Parse Tree Properties



- A parse tree has
 - The start symbol at the root
 - Terminals at the leaves
 - Nonterminals at the interior nodes
- A post-order traversal of the leaves yields the original input string
- The parse tree shows which operands associate with which operations

© All Rights Reserved

Derivation Types



1. Leftmost derivation: replace, at each derivation step, the leftmost nonterminal
 2. Rightmost derivation: replace, at each derivation step, the rightmost nonterminal
- Of course, replacing nonterminals can occur in any order but the above two orders are the most commonly used

© All Rights Reserved

A Derivation Example



We will use the below grammar to show a leftmost derivation and a rightmost derivation for the string "x + 8 * y"

1.	S	→	E
2.	E	→	E Op E
3.			id
4.			num
5.	Op	→	plus
6.			mul

- The terminals are described by the following regular expressions:

id: $([a-z] | [A-Z])^+$

num: $[0-9]^+$

plus: '+'

mul: '*'

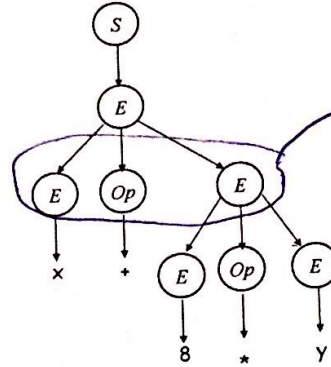
© All Rights Reserved

S/E/Op
Non terminals
id/num/plus/mul
terminals.

Leftmost Derivation

$$S \Rightarrow^* x + 8 * y$$

Rule	Sentential Form
-	S
1	E
2	E Op E
3	<id,x> Op E
5	<id,x> + E
2	<id,x> + E Op E
4	<id,x> + <num,8> Op E
6	<id,x> + <num,8> * E
3	<id,x> + <num,8> * <id,y>



A post-order tree walk of this parse tree evaluates as $x + (8 * y)$

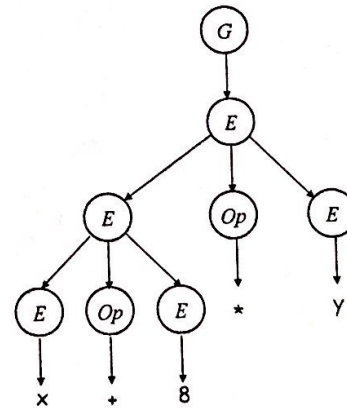
عكس ترتيب
اكثر من احوال
(E Op E / id / num)
هذه اختارنا
expression
التي عندها فيه args

non terminals
منه يوسع
expand
اول S هو بتجزي
Leftmost derivation
right most derivation

Another Leftmost Derivation

$$S \Rightarrow^* x + 8 * y$$

Rule	Sentential Form
-	S
1	E
2	E Op E
2	E Op E Op E
3	<id,x> Op E Op E
5	<id,x> + E Op E
4	<id,x> + <num,8> Op E
6	<id,x> + <num,8> * E
3	<id,x> + <num,8> * <id,y>



A post-order tree walk of this parse tree evaluates as $(x + 8) * y$

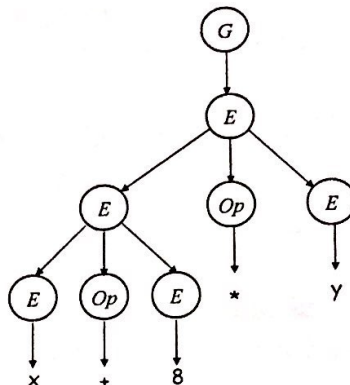
ambiguity -> left most

Rightmost Derivation



$$S \Rightarrow^* x + 8 * y$$

Rule	Sentential Form
—	S
1	E
2	$E Op E$
3	$E Op \langle id, y \rangle$
6	$E * \langle id, y \rangle$
2	$E Op E * \langle id, y \rangle$
4	$E Op \langle num, 8 \rangle * \langle id, y \rangle$
5	$E + \langle num, 8 \rangle * \langle id, y \rangle$
3	$\langle id, x \rangle + \langle num, 8 \rangle * \langle id, y \rangle$



A post-order tree walk of this parse tree evaluates as $(x + 8) * y$

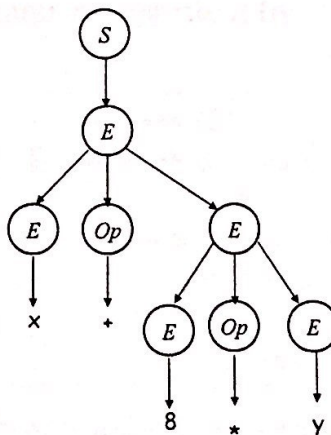
© All Rights Reserved

Another Rightmost Derivation



$$S \Rightarrow^* x + 8 * y$$

Rule	Sentential Form
—	S
1	E
2	$E Op E$
2	$E Op E Op E$
3	$E Op E Op \langle id, y \rangle$
6	$E Op E * \langle id, y \rangle$
4	$E Op \langle num, 8 \rangle * \langle id, y \rangle$
5	$E + \langle num, 8 \rangle * \langle id, y \rangle$
3	$\langle id, x \rangle + \langle num, 8 \rangle * \langle id, y \rangle$



A post-order tree walk of this parse tree evaluates as $x + (8 * y)$

© All Rights Reserved

Ambiguity



- A context-free grammar G is ambiguous if it has multiple leftmost (or multiple rightmost) derivations for some string in $L(G)$
- Equivalently, a context-free grammar G is ambiguous if there is multiple parse trees for some string in $L(G)$
- Therefore, a context-free grammar G is not ambiguous if all strings in $L(G)$ have unique parse trees
- Ambiguity is bad in a programming language because it can lead the compiler to interpret different meanings for the same program

© All Rights Reserved

Eliminating Ambiguity



- To disambiguate an ambiguous grammar, rewrite it by hand

1.	$S \longrightarrow E$
2.	$E \longrightarrow E \text{ Op } E$
3.	id
4.	num
5.	$Op \longrightarrow plus$
6.	mul

Ambiguous Grammar

1.	$S \longrightarrow E$
2.	$E \longrightarrow E \text{ plus } \hat{E}$
3.	\hat{E}
4.	$\hat{E} \longrightarrow id \text{ mul } \hat{E}$
5.	$num \text{ mul } \hat{E}$
6.	id
7.	num

Rewritten Grammar: we gave multiplication precedence over summation

© All Rights Reserved

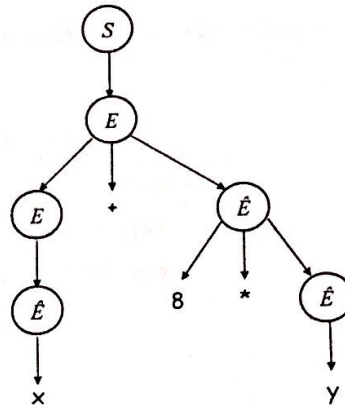
grammer, les
is the
ambiguity of the

Let us Try Leftmost Derivation



$$S \Rightarrow^* x + 8 * y$$

Rule	Sentential Form
—	S
1	E
2	$E + \hat{E}$
3	$\hat{E} + \hat{E}$
6	$\langle id, x \rangle + \hat{E}$
5	$\langle id, x \rangle + \langle num, 8 \rangle * \hat{E}$
6	$\langle id, x \rangle + \langle num, 8 \rangle * \langle id, y \rangle$



Leftmost derivation of $S \Rightarrow^* x + 8 * y$ is unique

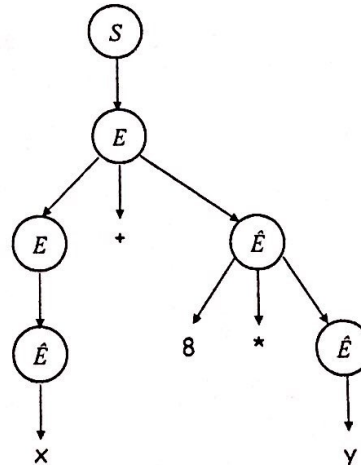
© All Rights Reserved

Let us Try Rightmost Derivation



$$S \Rightarrow^* x + 8 * y$$

Rule	Sentential Form
—	S
1	E
2	$E + \hat{E}$
5	$E + \langle num, 8 \rangle * \hat{E}$
6	$E + \langle num, 8 \rangle * \langle id, y \rangle$
3	$\hat{E} + \langle num, 8 \rangle * \langle id, y \rangle$
6	$\langle id, x \rangle + \langle num, 8 \rangle * \langle id, y \rangle$



Rightmost derivation of $S \Rightarrow^* x + 8 * y$ is unique

Note that leftmost and rightmost derivations yield the same parse tree

© All Rights Reserved

Adding Precedence to Grammars



- Adding precedence to grammars removes ambiguity
- General guidelines to adding precedence:
 - Create a nonterminal for each *level of precedence*
 - Isolate the corresponding part of the grammar
 - Force the parser to recognize high precedence subexpressions first

© All Rights Reserved

Example: Algebraic Expression Grammar



Straightforward grammar is ambiguous

1	<i>Start</i>	→	<i>Expr</i>
2	<i>Expr</i>	→	(<i>Expr</i>)
3			<i>Expr Op Expr</i>
4			<i>number</i>
6			<i>id</i>
8	<i>Op</i>	→	+
9			-
10			*
11			/

Exercise: show that parsing the string “ (x + 1) / y - 2 ” is ambiguous

© All Rights Reserved

Adding Precedence to Algebraic Expression Grammar



	1	<i>Start</i>	→	<i>Expr</i>
Addition and subtraction, last {	2	<i>Expr</i>	→	<i>Expr + Term</i>
	3			<i>Expr - Term</i>
	4			<i>Term</i>
	5	<i>Term</i>	→	<i>Term * Factor</i>
Multiplication and division, next {	6			<i>Term / Factor</i>
	7			<i>Factor</i>
	8	<i>Factor</i>	→	<i>(Expr)</i>
Parentheses have highest precedence {	9			number
	10			id

3
2
1

صدا
قسم اوليات
کی واحد
non terminal
لحاظ

Exercise: show the parse tree for the string “ (x + 1) / y - 2 ”

© All Rights Reserved

If-then-else Problem



- Another classic ambiguity example
- Consider the following straightforward grammar:

1.	<i>STMT</i>	→	if <i>EXPR</i> then <i>STMT</i>	
2.			if <i>EXPR</i> then <u><i>STMT</i></u> else <u><i>STMT</i></u>	
3.			... other statements ...	

- Let us inspect whether leftmost derivation for the following string is unique:

“ if *expr*₁ then if *expr*₂ then *stmt*₁ else *stmt*₂ ”

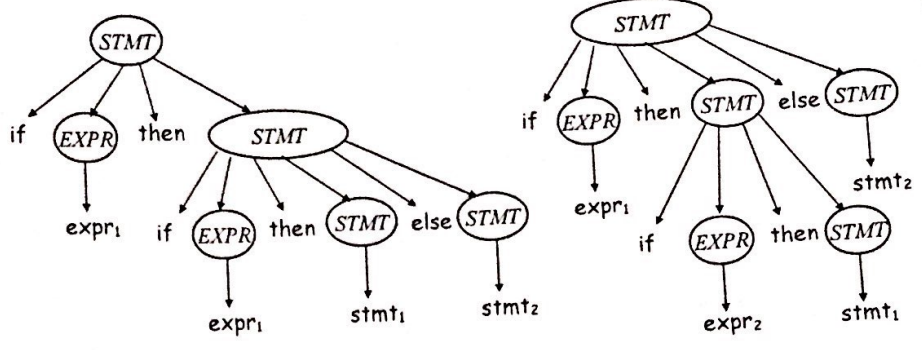
© All Rights Reserved

Symmetry لہاے
|
عینے
ambiguity



Leftmost Derivations

$STMT \Rightarrow^*$ if $expr_1$ then if $expr_2$ then $stmt_1$ else $stmt_2$



Two parse trees \rightarrow meaning is ambiguous

© All Rights Reserved



Solution

- Rewrite grammar to remove ambiguity by matching each "else" to innermost unmatched "if"

1	$STMT \rightarrow$	if $EXPR$ then $STMT$
2		if $EXPR$ then $STMT^\#$ else $STMT$
3		Other Statements
4	$STMT^\# \rightarrow$	if $EXPR$ then $STMT^\#$ else $STMT^\#$
5		Other Statements

Intuition: once into $STMT^\#$, we cannot generate an unmatched else

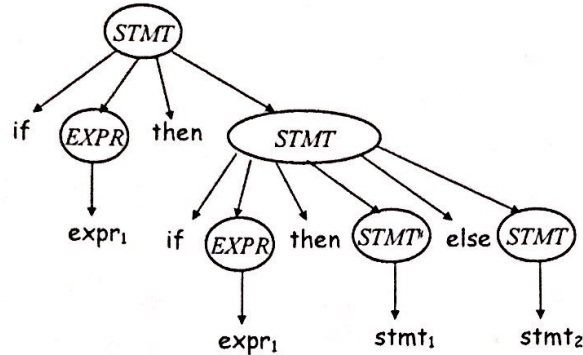
© All Rights Reserved

Breaking the symmetry led to breaking the ambiguity.

Parse Tree With Rewritten Grammar



$STMT \Rightarrow^*$ if $expr_1$ then if $expr_2$ then $stmt_1$ else $stmt_2$



Parse tree is now unique

© All Rights Reserved

Is There An Algorithm To Do it?

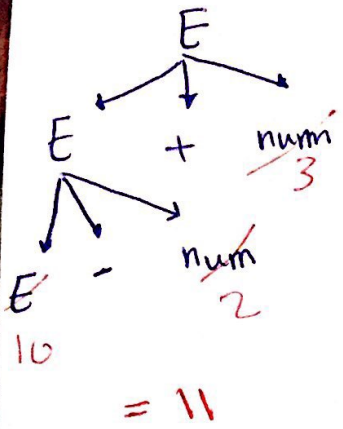


- There are no known algorithms to disambiguate ambiguous context-free grammars
- In fact, the problem of deciding if a context-free grammar G is ambiguous or not is undecidable
- To deal with ambiguous grammars, compiler writers:
 1. Modify context-free grammars by hand and ensure their unambiguity
 2. Or, allow compilers to accept ambiguous context-free grammars
 - Compiler writers include “guidelines” that tell the compiler which parse tree to choose when multiple trees can be generated

© All Rights Reserved

We usually choose left associative

Left grammar



Associativity

- We already saw examples on how parse trees determine the evaluation order
- We use the term grammar *associativity* to describe the evaluation order direction in parse trees
- Example: compare the evaluation order when parsing the string $10 - 2 + 3$ in the following grammars

1	$E \rightarrow E + \text{num}$
2	$E \rightarrow E - \text{num}$
3	$E \rightarrow \text{num}$

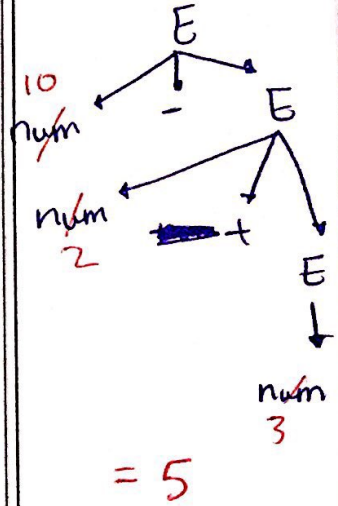
Left-associative grammar

1	$E \rightarrow \text{num} + E$
2	$E \rightarrow \text{num} - E$
3	$E \rightarrow \text{num}$

Right-associative grammar

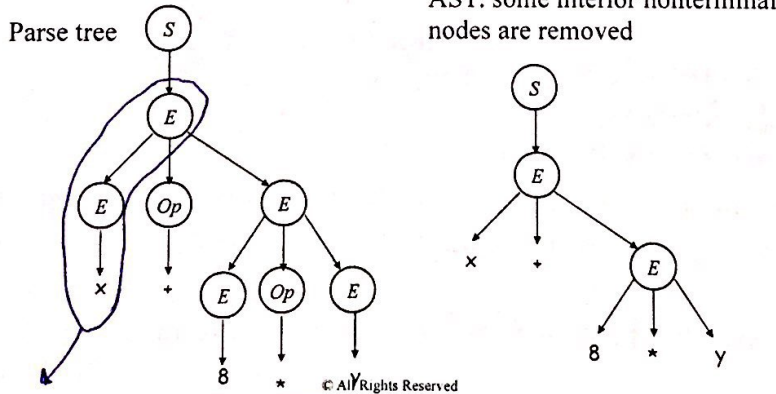
© All Rights Reserved

Right grammar



Abstract Syntax Tree (AST)

- Abstract syntax trees are parse trees but may ignore some details



Less memory and less complexity.

Parse tree
 optimized
 parse tree

Handwritten notes in Arabic script.

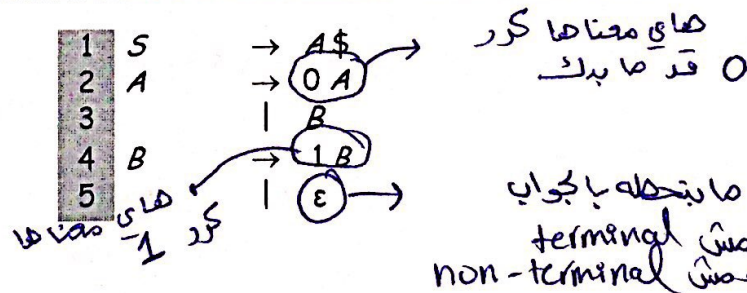
Summary



- Context-free grammars are powerful mathematical model of syntax in programming languages
- For a given context-free grammar G with a start symbol S , the language $L(G)$ is all strings N such that:
 - N contains only terminal symbols, i.e., legal tokens in the language
 - $S \Rightarrow^* N$, i.e, there is a derivation of N in $L(G)$ using the production rules of G
- A Parse tree is a directed graph that represents the derivation of a string in $L(G)$
- Ambiguity in context-free grammars is undesirable

© All Rights Reserved

Exercise



- Specify the terminals and the nonterminals of this grammar
- Write a regular expression that can generate the language described by this grammar
- Using leftmost derivation, draw the parse tree for the string "011\$"

© All Rights Reserved

(tokens)
 ① terminals : ~~0, 1, \$~~ 0, \$, 1
 non-terminals : S, A, B

② $0^* 1^* \$$

① $S \rightarrow 0A1$
 $A \rightarrow 0A$
 $A \rightarrow \epsilon$

2 % Non terminals جملات
 terminals

Exercise

- ① Write a context-free grammar that describes the same language as the regular expression 0^*1
- ② Write a context-free grammar that describes the same language as the regular expression 01^* up to 2 non-terminals
- ③ Write a context-free grammar that describes the same language as the regular expression $01|01^*$

لو جملات مني ا جملات
 non terminal 1
 $\rightarrow S01$
 $S \rightarrow 0A01$

②
 $S \rightarrow 01A$
 $A \rightarrow 1A$
 $A \rightarrow \epsilon$

③
 $S \rightarrow A|B$
 $A \rightarrow 0A1$
 $A \rightarrow \epsilon$
 $B \rightarrow 01B$
 $B \rightarrow 1B$
 $B \rightarrow \epsilon$

© All Rights Reserved

Exercise

- Is the following grammar ambiguous? Justify your answer

1	S	→	XaaX
2			aX
3	X	→	Xa
4			Xb
5			ε

ambiguous, because of symmetry.
 كيف بتبين؟ من خلال مثال

© All Rights Reserved

Lecture 5

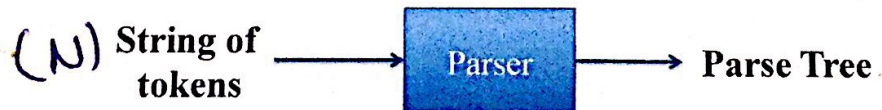
Top-Down Parsing

Spring 2018/2019

Instructor: Dr. Fahed Jubair
Computer Engineering Department
University of Jordan



The Parser Role



- The parser processes all tokens returned from the scanner and produces a parse tree that represents the input program structure (or a syntax error if an invalid structure is found)
- In this lecture, we will study top-down parsing: a computer algorithm that builds the parse tree using the derivation rules of a context-free grammar
- There is also bottom-up parsing but it will not be covered by this course

Parse Tree Properties



- A parse tree has
 - The start symbol at the root
 - Nonterminals at the interior nodes
 - Terminals at the leaves
- A derivation is discovered if a post-order traversal of the leaves (i.e., terminal nodes) match the tokens returned by the scanner

Top-Down Parsing



- A basic top-down parsing algorithm:
 1. Construct the root node of the parse tree
 - Loop* 2. Repeat until lower fringe of the parse tree matches the string of tokens
 - i. At a node labeled A, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child
 - ii. When a terminal symbol is added to the fringe and it doesn't match the fringe, backtrack
 - iii. Find the next node to be expanded

Recall The Algebraic Expression Grammar



- 1 $S \rightarrow Expr$
- 2 $Expr \rightarrow Expr + Term$
- 3 $Expr \rightarrow Expr - Term$
- 4 $Expr \rightarrow Term$
- 5 $Term \rightarrow Term * Factor$
- 6 $Term \rightarrow Term / Factor$
- 7 $Term \rightarrow Factor$
- 8 $Factor \rightarrow (Expr)$
- 9 $Factor \rightarrow number$
- 10 $Factor \rightarrow id$

Let us try deriving $S \Rightarrow^* x - 2 * y$
using the basic top-down parsing algorithm

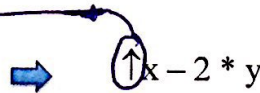
© All Rights Reserved.

5

$$S \Rightarrow^* x - 2 * y$$



Input Stream (the arrow points to the next input token):



The root is the start symbol

Pick rule 1

randomly

Pick rule 2

Pick rule 4

Pick rule 7

Pick rule 10

"x" matches "id" type → advance arrow to the next input token

© All Rights Reserved.

غیر ماہی
فاختا، ماہی

randomly expansion
Term, فاختا
بیشک عسوائی

Term expansion
بیشک عسوائی
فاختا, Factor

Factor expand
<id>
واختا

کون عی match بی id و x و انتقل ل token الی جز

$$S \Rightarrow^* x - 2 * y$$

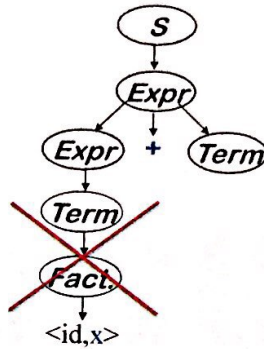


Input Stream (the arrow points to the next input token):

$x \uparrow - 2 * y$

"+" does not match "-"

The algorithm backtracks and try a different rule while reversing focus arrow



© All Rights Reserved

هون
 + 11 rule
 ما بقا
 match
 back track
 لسا و ريتا

id . x

هون كوك ايسم لا نه صا ، ا match

$$S \Rightarrow^* x - 2 * y$$



Input Stream (the arrow points to the next input token):

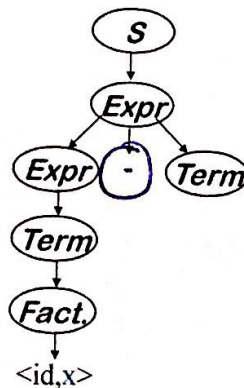
$\uparrow x - 2 * y$

Pick rule 3

Pick rule 4

Pick rule 7

Pick rule 10



"x" matches "id" type → advance to the next input token

© All Rights Reserved

8

$S \Rightarrow^* x - 2 * y$

Input Stream (the arrow points to the next input token):

→ $x \uparrow - 2 * y$

“-” matches “-”

→

advance to the next input token

```

graph TD
    S((S)) --> E1((Expr))
    E1 --> E2((Expr))
    E1 --> T1((Term))
    E2 --> T2((Term))
    T2 --> F1((Fact))
    F1 --> ID("<id,x>")
            
```

© All Rights Reserved

$S \Rightarrow^* x - 2 * y$

Input Stream (the arrow points to the next input token):

→ $x - \uparrow 2 * y$

match *هو*

Pick rule 7 →

Pick rule 9 →

“2” matches “number” type → advance to the next input token

```

graph TD
    S((S)) --> E1((Expr))
    E1 --> E2((Expr))
    E1 --> T1((Term))
    E2 --> T2((Term))
    T2 --> F1((Fact))
    F1 --> ID("<id,x>")
    T1 --> F2((Fact))
    F2 --> NUM("<number,2>")
            
```


expand *هو*

1 *هو* *هو*

match

هو

© All Rights Reserved



S ⇒ * x - 2 * y

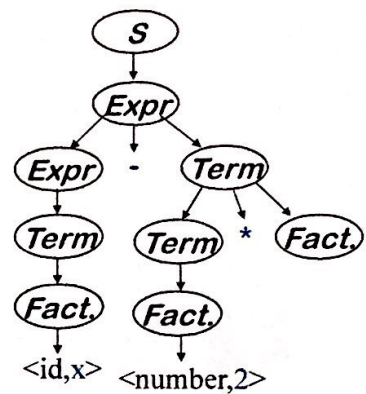
Input Stream (the arrow points to the next input token):

→ x - 2↑ * y


“*” matches “*”

→

advance to the next input token



© All Rights Reserved 13



S ⇒ * x - 2 * y

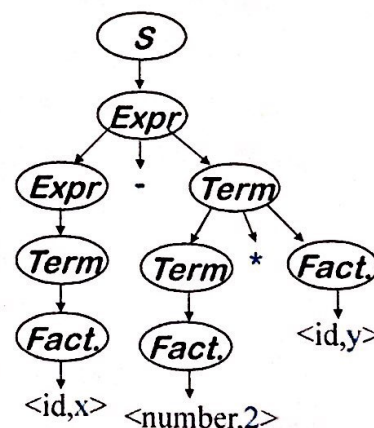
Input Stream (the arrow points to the next input token):

→ x - 2 * ↑y

Pick rule 10

→

“y” matches “id” type → advance to the next input character

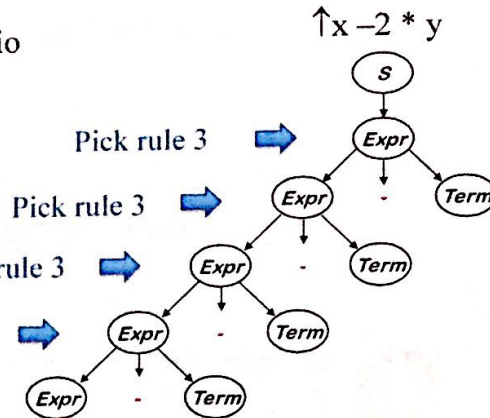


© All Rights Reserved 14

One Possible Bad Scenario



Consider the following scenario when deriving $S \Rightarrow^* x - 2 * y$



المسألة 8
 left recursive ← grammar
 top-down parsing
 leftmost derivation
 Pick rule 3

Because rule 3 is left-recursive, top-down parsing has the possibility of infinite execution

الحل
 grammar
 left recursive
 rightmost derivation
 leftmost derivation
 infinite loop
 right recursive
 left most derivation

The Left Recursion Problem



1	S	\rightarrow	$Expr$
2	$Expr$	\rightarrow	$Expr + Term$
3		$ $	$Expr - Term$
4		$ $	$Term$
5	$Term$	\rightarrow	$Term * Factor$
6		$ $	$Term / Factor$
7		$ $	$Factor$
8	$Factor$	\rightarrow	$(Expr)$
9		$ $	number
10		$ $	id

- Recursive use of rules 2, 3, 5 and 6 leads to an infinite sequence of expansions
- Non-termination is definitely a bad property for compilers
- We refer to this problem as the left recursion problem

هذا كل اللي اع
 نفعه

Eliminating Left Recursion



- Solution: rewrite grammars so that they are right-recursive
 - By hand
 - Using an automatic tool
- Consider the following simple left recursive grammar:

$$A \rightarrow A\alpha \quad \beta\alpha^*$$

$$| \beta$$

- We (or an automatic tool) can rewrite the grammar as follows:

$$\left. \begin{array}{l} A \rightarrow \beta \hat{A} \\ \hat{A} \rightarrow \alpha \hat{A} \\ | \epsilon \end{array} \right\} \text{The new grammar is} \\ \text{right-recursive}$$

© All Rights Reserved

19

right recursive
 ωωε ←
 Non-terminal
 جديد

Let us Generalize



A general left-recursive grammar

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n$$

$$| \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$$



The equivalent right-recursive grammar

$$A \rightarrow \beta_1 \hat{A} \mid \beta_2 \hat{A} \mid \dots \mid \beta_m \hat{A}$$

$$\hat{A} \rightarrow \alpha_1 \hat{A} \mid \alpha_2 \hat{A} \mid \dots \mid \alpha_n \hat{A} \mid \epsilon$$

© All Rights Reserved

20

Eliminating Left Recursion in the Expression Grammar



The expression grammar contain the following left recursion cases

$$\begin{array}{l}
 \text{Expr} \rightarrow \text{Expr} + \text{Term} \\
 \quad | \quad \text{Expr} - \text{Term} \\
 \quad | \quad \text{Term} \\
 \hat{A} \text{Term} \rightarrow \text{Term} * \text{Factor} \\
 \quad | \quad \text{Term} / \text{Factor} \\
 \quad | \quad \text{Factor}
 \end{array}$$



Applying the transformation yields

$$\begin{array}{l}
 \text{Expr} \rightarrow \text{Term} \hat{E} \\
 \hat{E} \rightarrow + \text{Term} \hat{E} \\
 \quad | \quad - \text{Term} \hat{E} \\
 \quad | \quad \epsilon \\
 \text{Term} \rightarrow \text{Factor} \check{T} \\
 \check{T} \rightarrow * \text{Factor} \check{T} \\
 \quad | \quad / \text{Factor} \check{T} \\
 \quad | \quad \epsilon
 \end{array}$$

The Right-Recursive Algebraic Expression Grammar



Right-Recursive as S is 1st

1	S	\rightarrow	Expr
2	Expr	\rightarrow	$\text{Term} \hat{E}$
3	\hat{E}	\rightarrow	$+ \text{Term} \hat{E}$
4		$ $	$- \text{Term} \hat{E}$
5		$ $	ϵ
6	Term	\rightarrow	$\text{Factor} \check{T}$
7	\check{T}	\rightarrow	$* \text{Factor} \check{T}$
8		$ $	$/ \text{Factor} \check{T}$
9		$ $	ϵ
10	Factor	\rightarrow	(Expr)
11		$ $	number
12		$ $	id

- A top-down parser will always terminate when using this grammar
- Exercise: show the parse tree of $S \Rightarrow^* x - 2 * y$ using the basic top-down parsing algorithm

How to Code Top-Down Parsers?



- Multiple approaches have been introduced in the literature to implement top-down parsers
- A popular implementation is the recursive-descendent parser
- A recursive-descendent parser comprises a set of mutually recursive routines that cooperate to parse a string of tokens
 - Each routine typically corresponds to a single production rule

A Basic Function For The Recursive-Descendent Parser



- Let (next) be a pointer that points to the next token to be consumed in the input stream
- Define a boolean function that checks for a match of a token with the next input token:

```

bool MATCH (Token t) {
    if (t == *next)
        IsEqual = true ;
    else
        IsEqual = false ;
    next ++;
    return IsEqual;
}

```

جب تک لا حوالہ
 match کر لا
 بڑھ (pointer) سوڈا

non terminal
 rule بنجره
 و يكون انه boolean function

Top-Down Parser Code For A Simple Grammar

1	S	→	BCA
2	A	→	S
3			ε
4	B	→	€
5			£
6			ε
7	C	→	number
8			id

هناي مبرورين لان كلهم لازم يتحققوا

```

Rule S
bool S () { return B() && C() && A(); }
            A / C / B J expand
Rule A
bool A1 () { return S(); }
bool A2 () { return true; }
bool A () {
    Token *save = next;
    return (A1(); // try A1 rule first
           || (next = save; A2();); // then try A2 rule
           )
}
    
```

اذا اوله true مايجل لثانية
 Backtrack by reversing pointer
 Only tries rule A2 if A1 fails

اذا اوله true مايجل لثانية

Save هياي بتقل
 token جوا next
 لسه Save

اذا اوله وحده true يرجع false
 وبعث back track

Top-Down Parser Code For A Simple Grammar

1	S	→	BCA
2	A	→	S
3			ε
4	B	→	€
5			£
6			ε
7	C	→	number
8			id

```

Rule B
bool B1 () { return MATCH(€); }
bool B2 () { return MATCH(£); }
bool B3 () { return true; }

bool B () {
    Token *save = next;
    return (B1(); // try B1 rule first
           || (next = save; B2();); // then try B2 rule
           || (next = save; B3();); // then try B3 rule
           )
}
    
```

© All Rights Reserved

Top-Down Parser Code For A Simple Grammar



Rule C

```
bool C1 () { return MATCH( number ); }
bool C2 () { return MATCH( id ); }
```

```
bool C () {
    Token *save = next;
    return ( C1(); ) // try C1 rule first
           || ( next = save; C2(); ); // then try C2 rule
}
```

1	S	\rightarrow	BCA
2	A	\rightarrow	S
3		$ $	ϵ
4	B	\rightarrow	ϵ
5		$ $	ϵ
6		$ $	ϵ
7	C	\rightarrow	$number$
8		$ $	id

Exercise



- Write the code of a recursive-descendent parser for the expression grammar

1	S	\rightarrow	$Expr$
2	$Expr$	\rightarrow	$Term \hat{E}$
3	\hat{E}	\rightarrow	$+ Term \hat{E}$
4		$ $	$- Term \hat{E}$
5		$ $	ϵ
6	$Term$	\rightarrow	$Factor \check{T}$
7	\check{T}	\rightarrow	$* Factor \check{T}$
8		$ $	$/ Factor \check{T}$
9		$ $	ϵ
10	$Factor$	\rightarrow	$(Expr)$
11		$ $	$number$
12		$ $	id

Invoking A Recursive-Descendent Parser



- To start a recursive-descendent parser:
 - Initialize next to point to the first token
 - Invoke the start symbol routine
- Easy to implement by hand
- Similar to scanners, there are software tools that generate the code of parsers automatically
- Example: ANTLR



*lexer.java
Parser.java*

© All Rights Reserved

29

Summary



- Top-down parsers find a derivation for a string of tokens by building a parse tree
 - The parser starts at the root and then extends the tree downward (hence the name top-down parsing)
 - The parser terminates when the leaves matches the tokens returned by the scanner
 - Leftmost derivation is used
- Backtracking is needed when a “bad” pick of a production rule is used
- Top-down parsers cannot handle left recursive grammars
 - Solution: rewrite grammars to be right recursive

© All Rights Reserved

30

Summary (cont.)



- Recursive-descendent parsers are popular implementation for top-down parsers
 - However, a major source of inefficiency is the need to backtrack
 - Luckily, there are algorithms for backtrack-free top-down parsing 😊
Performance (أداء)
- **The topic of our next lecture**

Exercise



1. S	→	(A) × (B)
2. A	→	T num
3. T	→	T num +
4.		ε
5. B	→	B + num
6.		num

- Rewrite the above grammar into a right-recursive grammar
- Using your new grammar, show the parse tree for the following strings:
 - (1 + 2 + 3) × (2)
 - (2) × (1 + 2 + 3)

Predictive Parsing Spring 2018/2019

Instructor: Dr. Fahed Jubair
Computer Engineering Department
University of Jordan



Review: The Problem of Backtracking



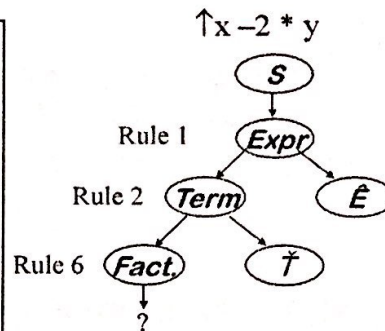
- Backtracking makes top-down parsers really slow and inefficient
- This inefficiency arises from the parser's lack of knowledge of which production rule is the correct one
 - Therefore, it tries all rules till finding the correct one
- This lecture introduces LL parsing: a computer algorithm for performing backtrack-free top-down parsing

عنوان نحس مسلكه
 في performance
 backtrack

Key Idea: Looking Ahead Helps



1	S	\rightarrow	$Expr$
2	$Expr$	\rightarrow	$Term \hat{E}$
3	\hat{E}	\rightarrow	$+ Term \hat{E}$
4		$ $	$- Term \hat{E}$
5		$ $	ϵ
6	$Term$	\rightarrow	$Factor \check{T}$
7	\check{T}	\rightarrow	$* Factor \check{T}$
8		$ $	$/ Factor \check{T}$
9		$ $	ϵ
10	$Factor$	\rightarrow	$(Expr)$
11		$ $	number
12		$ $	id



A smart parser would **lookahead** at the next input token "x" and conclude that the rule $Factor \rightarrow id$ is the correct rule to choose

© All Rights Reserved

3

Backtrack-Free Parsers



- Given $A \rightarrow \alpha \mid \beta$, a backtrack-free top-down parser should be able to choose between α & β without the need to backtrack
- The key idea is to "look ahead" at the next input token when selecting the production rule
 - Let us call this token the lookahead token
- We refer to such parsers as predictive parsers because they predict the "correct" rule to use
- Predictive top-down parsers are also called LL parsers
 - They read the input stream from left to right (hence the first "L") and they use leftmost derivation (hence the second "L")

© All Rights Reserved.

4

non terminal نون

↑ rules رولز

unique start نون عنيق

LL(1) Grammar



- A grammar for which a top-down parser that reads the input from left to right and uses leftmost derivation needs a lookahead of at most one token to always predict the correct rule
- Using LL(1) grammars, a top-down parser can always predict the correct rule every time it expands a nonterminal

© All Rights Reserved

5

Predictive Top-Down Parsing



- Consider a top-down parser that uses an LL(1) grammar
- Let the next nonterminal node to be expanded by the parser be A
- Let the lookahead token be t
- When expanding A , the LL(1) grammar has the property that there is a unique production rule $A \rightarrow \alpha$ such that $\alpha \Rightarrow^* t \beta$, i.e., there is only one rule that can derive token t in the first position
- Therefore, LL(1) grammars enable top-down parsers to be predictive

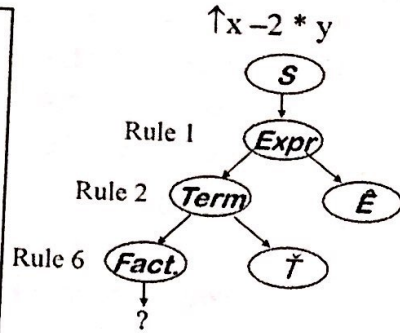
© All Rights Reserved

6

LL(1) Grammar Example



1	S	\rightarrow	$Expr$
2	$Expr$	\rightarrow	$Term \hat{E}$
3	\hat{E}	\rightarrow	$+ Term \hat{E}$
4		$ $	$- Term \hat{E}$
5		$ $	ϵ
6	$Term$	\rightarrow	$Factor \check{T}$
7	\check{T}	\rightarrow	$* Factor \check{T}$
8		$ $	$/ Factor \check{T}$
9		$ $	ϵ
10	$Factor$	\rightarrow	$(Expr)$
11		$ $	number
12		$ $	id



Lookahead token is x
 $Factor \rightarrow id$ is the **only** rule that derives x in the first position

Prediction Criteria



- A production rule $A \rightarrow \alpha$ can derive a terminal t in the first position under one of the following two conditions:
 - $t \in FIRST(\alpha)$, OR
 - $\epsilon \in FIRST(\alpha)$ and $t \in FOLLOW(\alpha)$

Let us define FIRST
and FOLLOW sets

First set
 \downarrow
 token ϵ \in α
 rule J
 تَبَيَّنَ فِيهَا

FIRST Sets



- Let $A \rightarrow \alpha$ be a production rule in an LL(1) grammar
- $FIRST(\alpha)$ is the set of all **terminals** that appear as the first token in some string that derives from α
 $FIRST(\alpha) = \{ \text{all terminal tokens } t \text{ such that } \alpha \Rightarrow^* t \beta \}$
- $FIRST$ sets have the following properties:
 - $FIRST(t) = \{ t \}$, where t is a terminal
 - $\epsilon \in FIRST(\alpha)$ if any of the following holds:
 - $\alpha \rightarrow \epsilon$, OR
 - $\alpha \rightarrow X_1 X_2 \dots X_n$ and $X_i \rightarrow \epsilon$ for all $i: 1 \leq i \leq n$
 - $FIRST(\alpha) \subseteq FIRST(\beta)$
 if $\beta \rightarrow X_1 X_2 \dots X_n \alpha$ and $X_i \rightarrow \epsilon$ for all $i: 1 \leq i \leq n$

© All Rights Reserved

9

Example 1



كل token بيلى بنفسه

- $FIRST(+)=\{+\}$
- $FIRST(-)=\{-\}$
- $FIRST(*)=\{*\}$
- $FIRST(/)=\{/ \}$

مجموعة tokens التي بيلى فيه Factor

- $FIRST(Factor) = \{ (, \text{number}, \text{id} \}$
- $FIRST(* Factor \check{T}) = \{ * \}$
- $FIRST(\check{T}) = \{ *, /, \epsilon \}$
- $FIRST(Term) = \{ (, \text{number}, \text{id} \}$
- $FIRST(\hat{E}) = \{ +, -, \epsilon \}$
- $FIRST(Expr) = \{ (, \text{number}, \text{id} \}$
- $FIRST(S) = \{ (, \text{number}, \text{id} \}$

1	S	$\rightarrow Expr$
2	$Expr$	$\rightarrow Term \hat{E}$
3	\hat{E}	$\rightarrow + Term \hat{E}$
4		$ - Term \hat{E}$
5		$ \epsilon$
6	$Term$	$\rightarrow Factor \check{T}$
7	\check{T}	$\rightarrow * Factor \check{T}$
8		$ / Factor \check{T}$
9		$ \epsilon$
10	$Factor$	$\rightarrow (Expr)$
11		$ \text{number}$
12		$ \text{id}$

الاشي التي بيلى فيه Term هو نفسه التي بيلى فيه Factor.

© All Rights Reserved

10

Example 2



1	S	→	BCA
2	A	→	S
3			ε
4	B	→	€
5			£
6			ε
7	C	→	number
8			id

- Terminals are €, £, number, id
- Nonterminals are A, B, C, S

- FIRST (C) = { number, id }
- FIRST (B) = { €, £, ε }
- FIRST (S) = { €, £, number, id }
- FIRST (A) = { €, £, number, id, ε }

هدون كيف اجو S
 ار First ل S فهو نفسه ار First ل B الی هو € و £
 بینما بنحط € ضمنه ار First ل S بنحسب لانه مش موجود

© All Rights Reserved.

11

€
 متى ممكن تكون
 First(S) من ضمنه
 اذا كانت € من ضمنه
 First(C) and
 First(B) and
 First(A)

number, id. وينقل للي بعدهما (C) وينحط ل First ايه الی هو

FOLLOW Sets



- Let A be nonterminal in an LL(1) grammar that has start symbol S
- $FOLLOW(A)$ is the set of all terminals that follow A in some sentential form

$$FOLLOW(A) = \{ \text{all terminals } t \text{ such that } S \Rightarrow^* \alpha A t \beta \}$$

- $FOLLOW$ sets have the following properties:
 1. $\$ \in FOLLOW(S)$, where $\$$ is a special end-of-input token
 2. Ignore ϵ when computing $FOLLOW$ sets
 3. If $A \rightarrow \alpha \beta$, then $FIRST(\beta) \subseteq FOLLOW(\alpha)$
 4. If $A \rightarrow \alpha \beta$, then $FOLLOW(A) \subseteq FOLLOW(\beta)$
 5. If $A \rightarrow \alpha \beta$ and $\beta \Rightarrow^* \epsilon$, then $FOLLOW(A) \subseteq FOLLOW(\alpha)$

© All Rights Reserved

12

Example 1

$FOLLOW(S) \subseteq FOLLOW(Expr)$
 $FOLLOW(Expr) \subseteq FOLLOW(\hat{E})$
 $FOLLOW(Expr) \subseteq FOLLOW(Term)$
 $FIRST(\hat{E}) \subseteq FOLLOW(Term)$
 $FOLLOW(\hat{E}) \subseteq FOLLOW(Term)$
 $FIRST(Term) \subseteq FOLLOW(+)$
 $FIRST(Term) \subseteq FOLLOW(-)$
 $FOLLOW(Term) \subseteq FOLLOW(\check{T})$
 $FOLLOW(Term) \subseteq FOLLOW(Factor)$
 $FIRST(\check{T}) \subseteq FOLLOW(Factor)$
 $FOLLOW(\check{T}) \subseteq FOLLOW(Factor)$
 $FIRST(Factor) \subseteq FOLLOW(*)$
 $FIRST(Factor) \subseteq FOLLOW(/)$
 $FIRST(Expr) \subseteq FOLLOW(')$

1	S	→	$Expr$
2	$Expr$	→	$Term \hat{E}$
3	\hat{E}	→	$+ Term \hat{E}$
4			$- Term \hat{E}$
5			ϵ
6	$Term$	→	$Factor \check{T}$
7	\check{T}	→	$* Factor \check{T}$
8			$/ Factor \check{T}$
9			ϵ
10	$Factor$	→	$(Expr)$
11			number
12			id

© All Rights Reserved 13

بدور علی ایس
 بندهی ل token
 بندور علیہ بالینہ
 ویشون شوقیہ
 (~~علیہ~~ آخرہ)

Example 1 (cont.)

$FOLLOW(S) = \{ \$ \}$
 $FOLLOW(Expr) = \{ \$, , \}$
 $FOLLOW(\hat{E}) = \{ \$, , \}$
 $FOLLOW(Term) = \{ \$, , +, - \}$
 $FOLLOW(\check{T}) = \{ \$, , +, - \}$
 $FOLLOW(Factor) = \{ \$, , +, -, *, / \}$
 $FOLLOW(+) = \{ number, id, (\}$
 $FOLLOW(-) = \{ number, id, (\}$
 $FOLLOW(*) = \{ number, id, (\}$
 $FOLLOW(/) = \{ number, id, (\}$
 $FOLLOW(' (') = \{ number, id, (\}$
 $FOLLOW(') ') = \{ \$, , +, -, *, / \}$
 $FOLLOW(number) = \{ \$, , +, -, *, / \}$
 $FOLLOW(id) = \{ \$, , +, -, *, / \}$

1	S	→	$Expr$
2	$Expr$	→	$Term \hat{E}$
3	\hat{E}	→	$+ Term \hat{E}$
4			$- Term \hat{E}$
5			ϵ
6	$Term$	→	$Factor \check{T}$
7	\check{T}	→	$* Factor \check{T}$
8			$/ Factor \check{T}$
9			ϵ
10	$Factor$	→	$(Expr)$
11			number
12			id

© All Rights Reserved 14

صموج تگون E
 موجودہ پار First
 لیس صموج تگون
 موجودہ پار Follow

\$: special end-of-stream character (دائماً جز من جواب)

Example 2



- Terminals are $\epsilon, \text{£}, \text{number}, \text{id}$
- Nonterminals are A, B, C, S

1	S	\rightarrow	BCA
2	A	\rightarrow	S
3		$ $	ϵ
4	B	\rightarrow	£
5		$ $	£
6		$ $	ϵ
7	C	\rightarrow	number
8		$ $	id

$\text{FOLLOW}(S) = \text{FOLLOW}(A)$ (why?)
 $\text{FOLLOW}(S) \subseteq \text{FOLLOW}(C)$
 $\text{FIRST}(C) \subseteq \text{FOLLOW}(B)$
 $\text{FIRST}(A) \subseteq \text{FOLLOW}(C)$

$\text{FOLLOW}(S) = \{ \$ \}$
 $\text{FOLLOW}(A) = \{ \$ \}$
 $\text{FOLLOW}(B) = \{ \text{number}, \text{id} \}$
 $\text{FOLLOW}(C) = \{ \$, \epsilon, \text{£}, \text{number}, \text{id} \}$
 $\text{FOLLOW}(\epsilon) = \text{FOLLOW}(\text{£}) = \{ \text{number}, \text{id} \}$
 $\text{FOLLOW}(\text{number}) = \text{FOLLOW}(\text{id}) = \{ \$, \epsilon, \text{£}, \text{number}, \text{id} \}$

© All Rights Reserved

15

المصطلحات (مجموعة) التي يمكن ان ياتي بها -

PREDICT Sets



- We now merge FIRST and FOLLOW sets into a single set, called the PREDICT set
- For each production rule $A \rightarrow \alpha$ in the LL(1) grammar, we define $PREDICT(\alpha)$ as the set of all terminals that appear as the first token in some string that derives from α
- $PREDICT(\alpha)$ is computed as follows:

$\text{Predict}(\alpha) = \text{FIRST}(\alpha) - \{ \epsilon \}$
 if $(\epsilon \in \text{FIRST}(\alpha))$
 $\text{Predict}(\alpha) = \text{Predict}(\alpha) \cup \text{FOLLOW}(\alpha)$

© All Rights Reserved

16

$\rightarrow \text{Predict}(\alpha)$ is $\text{First}(\alpha)$ if α doesn't contain ϵ
 \rightarrow if α contains ϵ
 $\text{Predict}(\alpha)$ is ~~$\text{First}(\alpha)$~~ $\text{Predict}(\alpha) \cup \text{Follow}(\alpha)$

Example 1



$PREDICT(S) = \{ \text{number, id, (} \}$
 $PREDICT(Expr) = \{ \text{number, id, (} \}$
 $PREDICT(\hat{E}) = \{ +, -, \cdot, \$ \}$
 $PREDICT(Term) = \{ \text{number, id, (} \}$
 $PREDICT(\check{T}) = \{ +, -, *, /, \cdot, \$ \}$
 $PREDICT(Factor) = \{ \text{number, id, (} \}$

1	S	$\rightarrow Expr$
2	$Expr$	$\rightarrow Term \hat{E}$
3	\hat{E}	$\rightarrow + Term \hat{E}$
4		$ - Term \hat{E}$
5		$ \epsilon$
6	$Term$	$\rightarrow Factor \check{T}$
7	\check{T}	$\rightarrow * Factor \check{T}$
8		$ / Factor \check{T}$
9		$ \epsilon$
10	$Factor$	$\rightarrow (Expr)$
11		$ \text{number}$
12		$ \text{id}$

© All Rights Reserved

17

Example 2



- Terminals are €, £, number, id
- Nonterminals are A, B, C, S

1	S	$\rightarrow BCA$
2	A	$\rightarrow S$
3		$ \epsilon$
4	B	$\rightarrow \text{€}$
5		$ \text{£}$
6		$ \epsilon$
7	C	$\rightarrow \text{number}$
8		$ \text{id}$

- $PREDICT(C) = \{ \text{number, id} \}$
- $PREDICT(B) = \{ \text{€, £, number, id} \}$
- $PREDICT(S) = \{ \text{€, £, number, id} \}$
- $PREDICT(A) = \{ \text{€, £, number, id, \$} \}$

© All Rights Reserved

18

LL(1) Parsing Table



- For a given LL(1) grammar G , a parsing table T is a 2D array that informs the parser of which production rule to use when expanding a nonterminal A in a parse tree
- Hence, if A is a non-terminal and t is a terminal in G :
 $T[A, t] = \alpha$ such that $A \rightarrow \alpha$ and $t \in \text{PREDICT}(\alpha)$
- $T[A, t]$ is set to an error flag if no such production rule existed

LL(1) Parsing Table For The Expression Grammar



tokens من

	+	-	*	/	id	num	()	\$
S	-	-	-	-	1	1	1	-	-
Expr	-	-	-	-	2	2	2	-	-
E	3	4	-	-	-	-	-	5	5
Term	-	-	-	-	6	6	6	-	-
f	9	9	7	8	-	-	-	9	9
Factor	-	-	-	-	12	11	10	-	-

من non terminals

بشرف ليقام بينه اد tokens واد non terminals و بخط رقم ال rule

الي بتوري عليه غير هيك - خط (-)
(error)

Example 2



	€	£	num	id	\$
S	1	1	1	1	-
A	2	2	2	2	3
B	4	5	6	6	-
C	-	-	7	8	-

1	S	→	BCA
2	A	→	S
3			ε
4	B	→	€
5			£
6			ε
7	C	→	number
8			id

© All Rights Reserved

21

Example 2 (cont.)



We can also write the LL(1) parsing table as follows:

	€	£	num	id	\$
S	BCA	BCA	BCA	BCA	-
A	S	S	S	S	ε
B	€	£	ε	ε	-
C	-	-	num	id	-

1	S	→	BCA
2	A	→	S
3			ε
4	B	→	€
5			£
6			ε
7	C	→	number
8			id

Handwritten note: *↓ i case
doS expr ji
rulest per u y*

© All Rights Reserved

22

Predictive Top-Down Parsing



- The algorithm of predictive top-down parsing :
 1. Construct the parse table T
 2. Construct the root node of the parse tree
 3. Repeat until lower fringe of the parse tree matches the input string
 - i. At a node labeled A, select the production rule $T[A,t]$ from the parse table (where t is the lookahead token), and for each symbol on the rhs of this rule, construct the appropriate child
 - ii. Find the next node to be expanded

© All Rights Reserved.

23

Let us Redo $S \Rightarrow^* x - 2 * y$



An LL(1) version of the algebraic expression grammar

1	S	\rightarrow	$Expr$
2	$Expr$	\rightarrow	$Term \hat{E}$
3	\hat{E}	\rightarrow	$+ Term \hat{E}$
4		$ $	$- Term \hat{E}$
5		$ $	ϵ
6	$Term$	\rightarrow	$Factor \check{T}$
7	\check{T}	\rightarrow	$* Factor \check{T}$
8		$ $	$/ Factor \check{T}$
9		$ $	ϵ
10	$Factor$	\rightarrow	$(Expr)$
11		$ $	number
12		$ $	id

The parse table is shown on slide 20

Let us take it to the board

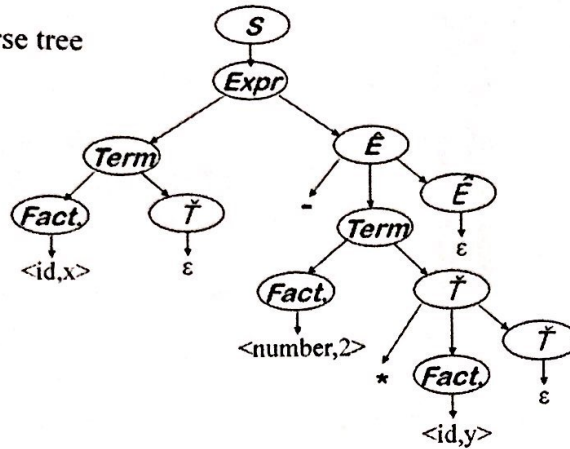
© All Rights Reserved

24

Let us Redo $S \Rightarrow^* x - 2 * y$



Final parse tree



© All Rights Reserved

25

Recursive-Descendent LL(1) Parsers



- In previous lecture, we studied recursive-descendent parsers: a popular implementation for top-down parsers
- We introduced:
 - *next*: a pointer that points to the next token to be consumed in the input stream
 - *MATCH*(Token *t*): a function that checks if a token *t* matches the next token to be consumed in the input stream
- We now introduce one more function:
 - *PEEK*() as the function that peeks at the input stream and returns the lookahead token

© All Rights Reserved

26

next
token

برجع ال

Predictive Top-Down Parser Code For A Simple Grammar



Rule S This is the PREDICT set
 bool $S()$ {
 Token $t = PEEK()$;
 if ($t \in \{\epsilon, \text{£}, \text{number}, \text{id}\}$)
 return $B() \ \&\& \ C() \ \&\& \ A()$;
 else
 return *false*; // parse error
 }

1	S	\rightarrow	BCA
2	A	\rightarrow	S
3			ϵ
4	B	\rightarrow	£
5			£
6			ϵ
7	C	\rightarrow	<i>number</i>
8			<i>id</i>

أولاً صورة رطلع سوار lookahead token
 وبتوف إذا موجود بار predict table إذا

© All Rights Reserved.

27

بعد من طبق عليها match إذا

Predictive Top-Down Parser Code For A Simple Grammar



Rule A
 bool $A()$ {
 Token $t = PEEK()$;
 if ($t \in \{\epsilon, \text{£}, \text{number}, \text{id}\}$)
 return $S()$;
 else if ($t \in \{\text{\$}\}$)
 return *true*;
 else
 return *false*; // parse error
 }

1	S	\rightarrow	BCA
2	A	\rightarrow	S
3			ϵ
4	B	\rightarrow	£
5			£
6			ϵ
7	C	\rightarrow	<i>number</i>
8			<i>id</i>

Note that Backtracking is not
 needed in this version of code

© All Rights Reserved.

28

Predictive Top-Down Parser Code For A Simple Grammar



Rule B

```
bool B () {
    Token  $t = PEEK ()$ ;
    if ( $t \in \{ \epsilon \}$ )
        return MATCH( $\epsilon$ );
    else if ( $t \in \{ \text{£} \}$ )
        return MATCH( $\text{£}$ );
    else if ( $t \in \{ \text{number}, \text{id} \}$ )
        return true;
    else
        return false; // parse error
}
```

1	S	\rightarrow	BCA
2	A	\rightarrow	S
3		$ $	ϵ
4	B	\rightarrow	ϵ
5		$ $	£
6		$ $	ϵ
7	C	\rightarrow	number
8		$ $	id

Predictive Top-Down Parser Code For A Simple Grammar



Rule C

```
bool C () {
    Token  $t = PEEK ()$ ;
    if ( $t \in \{ \text{number} \}$ )
        return MATCH(number);
    else if ( $t \in \{ \text{id} \}$ )
        return MATCH(id);
    else
        return false; // parse error
}
```

1	S	\rightarrow	BCA
2	A	\rightarrow	S
3		$ $	ϵ
4	B	\rightarrow	ϵ
5		$ $	£
6		$ $	ϵ
7	C	\rightarrow	number
8		$ $	id

Exercise



- Write the code of a recursive-descendent LL(1) parser for the expression grammar

1	S	\rightarrow	$Expr$
2	$Expr$	\rightarrow	$Term \hat{E}$
3	\hat{E}	\rightarrow	$+ Term \hat{E}$
4		$ $	$- Term \hat{E}$
5		$ $	ϵ
6	$Term$	\rightarrow	$Factor \check{T}$
7	\check{T}	\rightarrow	$* Factor \check{T}$
8		$ $	$/ Factor \check{T}$
9		$ $	ϵ
10	$Factor$	\rightarrow	$(Expr)$
11		$ $	number
12		$ $	id

© All Rights Reserved

31

The Problem is That Not All Grammars are LL(1)



- Assume a context-free grammar G with the production rules $A \rightarrow \alpha \mid \beta$
- Remember that for G to be an LL(1) grammar, then there is at most one production rule in $A \rightarrow \alpha \mid \beta$ that derives a Token t in the first position
- Therefore, G is LL(1) grammar if $PREDICT(\alpha) \cap PREDICT(\beta) = \emptyset$
- Otherwise, G is not an LL(1) grammar
 - The predictive top-down parsing algorithm shown in slide 22 cannot be used with grammars that are not LL(1)

© All Rights Reserved

32

کیف آعرف انہ ہمار LL(1) grammar ہو گا؟

← حسب اہتمام بینہ LL(1) predict ہو گا کہ نہ ہون LL(1) grammar

A Non-LL(1) Grammar Example



1	<i>Function</i>	\rightarrow	<i>id</i>
2			<i>id (ArgList)</i>
3			<i>id [ArgList]</i>
4	<i>ArgList</i>	\rightarrow	<i>id MoreArgs</i>
5	<i>MoreArgs</i>	\rightarrow	<i>, id MoreArgs</i>
6			ϵ

- Rules 1, 2 and 3 can all derive token *id* in the first position from nonterminal *Function*
- Can we rewrite this non-LL(1) grammar so that it is LL(1) grammar? **not LL(1) grammar.**

© All Rights Reserved.

33

Left Factoring



- Consider the following non-LL(1) grammar G

$$A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \alpha \beta_3 | \dots | \alpha \beta_n | \gamma$$

- The problem is that token α is a common prefix
- Solution: let us factor token α out
- Therefore, we can obtain the equivalent LL(1) version of G by introducing a new nonterminal \hat{A} :

$$A \rightarrow \alpha \hat{A} | \gamma$$

$$\hat{A} \rightarrow \beta_1 | \beta_2 | \beta_3 | \dots | \beta_n$$

© All Rights Reserved

34

Left Factoring Example

1	<i>Function</i>	→	<i>id</i>
2			<i>id (ArgList)</i>
3			<i>id [ArgList]</i>
4	<i>ArgList</i>	→	<i>id MoreArgs</i>
5	<i>MoreArgs</i>	→	<i>, id MoreArgs</i>
6			ϵ

1	<i>Function</i>	→	<i>id X</i>
2	<i>X</i>	→	<i>(ArgList)</i>
3			<i>[ArgList]</i>
4			ϵ
5	<i>ArgList</i>	→	<i>id MoreArgs</i>
6	<i>MoreArgs</i>	→	<i>, id MoreArgs</i>
7			ϵ

Non-LL(1) Grammar ? كيف هذا LL(1) Grammar LL(1) Grammar

© All Rights Reserved 35

أخذ id عامل مشترك و عد non terminal جديد اسما (X)
 Predict(x) = { ε, [, \$ } → Unique → LL(1) grammar

Left Factoring Doesn't Always Work

- Even with left factoring, some grammars still cannot be converted into an LL(1) grammar
- Possible solution: use LL(*k*) parsing, an advanced top-down parsing that uses *k* lookahead characters
- Another possible solution: use bottom-up parsing, another algorithm for parsing that covers a bigger class of grammars than top-down parsing

© All Rights Reserved 36

Summary



- Predictive top-down parsing is an efficient parsing technique that does not require backtracking
- To use predictive top-down parsing, context-free grammars must be rewritten as LL grammars
 - This can be done by hand or by an automatic software tool
- Predictive top-down parsers can be implemented as recursive-descendent parsers
- Most programming languages can be parsed using LL(1) parsers

© All Rights Reserved.

37

Exercise



1	$S \rightarrow AB\$$
2	$A \rightarrow xB$
3	$\quad \quad \quad xw$
4	$B \rightarrow xyA$
5	$\quad \quad \quad z$

- The above grammar is not LL(1). Explain why.
- Use left factoring to rewrite the grammar into an LL(1) grammar
- Show the LL(1) parsing table for your grammar in part ii
- Using the parsing table, show the derivation steps for the string $xyxwz\$$

© All Rights Reserved

38

i. not

A بتبليس X باطرسية

* ما يقارن A مع B ، يطلع على A كما انه اذا تقاطع زرع
 ويطلع على B لانه اذا تقاطع اوع .

$$ii. 1 S \rightarrow AB\$$$

$$2 A \rightarrow x\hat{A}$$

$$3 \hat{A} \rightarrow B$$

$$4 \quad | \quad w$$

$$5 B \rightarrow xyA$$

$$6 \quad | \quad z$$

بعد ما آخذ حاصل
مشترك لازم ارجع
أسسك إذا عندني
نقاط ارجع.

iii. Parsing table

	w	x	y	z	\$
S	-	1	-	-	-
A	-	2	-	-	-
\hat{A}	4	3	-	3	-
B	-	4	-	5	-

Lecture 7

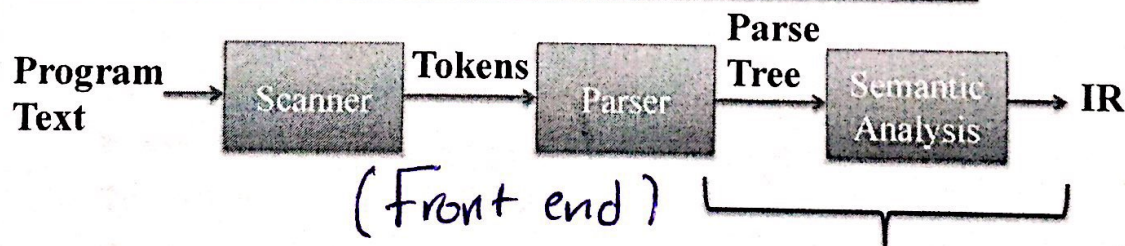
Semantic Actions

Spring 2018/2019

Instructor: Dr. Fahed Jubair
Computer Engineering Department
University of Jordan



Semantic Actions' Role



- The role of semantic actions (or semantic analysis) is to analyze the parse tree in order to
 1. Build the symbol table
 2. Check for semantic errors
 3. Generate the intermediate representation (IR)

Semantic Errors



- Some errors are beyond syntax analysis
- For example, what is wrong with the following C code?

```
float foo (int n, int m){
    int n;
    ...
}

void main (){
    int a, b ;
    float c [4];
    char *p;
    c[8] = 1 ;
    d = 2;
    p = foo(b);
    b = p + d ;
}
```

Syntax error

هون ما عندي

عندي semantic error

مثلا الفينكشن foo ال parameters 2

وهون نارديها باستخدام parameter 1

multiple variable declaration
↳ semantic error

P is char
P يخنز جوا
↳ semantic error

d is not declared → semantic errors .

All Rights Reserved

Semantic Actions



- Semantics actions are routines that are invoked while traversing the parse tree to examine the meaning of the program
- These routines check the meaning by applying a verity of correctness checks
- In the end of the semantic analysis, the parse tree is traversed in order to construct the IR

All Rights Reserved

- when visiting the parse tree we do more actions .

Visit children
 root
 د بقل
 لد
 تا عرنه
 من لشمال للصيه
 (post order)

← The Visitor Pattern

- We will define a *visitor* pattern that traverses all the nodes of a parse tree
- Each node recursively visits its children
- Default behavior: do nothing
- For example, consider the shown parse tree example

```
Visit S() {
  visit (X);
  visit (Y);
}
```

```
Visit X() {
  visit (A);
}
```

```
Visit Y() {
  visit (B);
  visit (C);
  visit (D);
}
```

© All Rights Reserved

S بتنادي ل children
 (x,y) و س
 children
 its بتنادي children

children (X,Y)
 س له صون

Parse Trees With The Same Children Type

- In parse trees, multiple instances of the same node may appear in the same level (e.g., see the parse tree below)
- To distinguish between these nodes, the visitor pattern uses an array

```
Visit X() {
  visit (A[0]);
  visit (B);
  visit (A[1]);
}
```

Ambiguous grammar

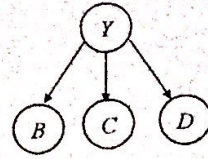
لین لو صہر، عنہ ی صہر لاشی
 بناد یلم باستخدا ام . Array

© All Rights Reserved

Integrating Semantic Actions With The Visitor Pattern



- Compiler writers **override** the visitor pattern functions to insert semantic actions



```

Visit Y() {
  #action1
  visit ( B );
  visit ( C );
  visit ( D );
  #action2
}
  
```

← Insert code here to perform actions **before** visiting the children nodes
 ← Insert code here to perform actions **after** visiting the children nodes

All Rights Reserved

Example 1



- The below grammar describes a list of identifier, which is often needed in programming languages
- Write semantic actions to count the number of IDs
 - E.g., when traversing the parse tree of (x, y, z), the count of IDs is 3

```

1 S      → ( id_list )
2 id_list → id list_tail
3 list_tail → , id list_tail
4         | ε
  
```

4 rules 4 visits

All Rights Reserved

```

Visit rule1() {
  visit ( c );
  visit ( id_list );
  visit ( ) );
}
  
```


Example 1 (cont.)



The Visitor pattern: default routines

```
Visit Rule1 () {
    visit (id_list);
    return ;
}
```

```
Visit Rule2 () {
    visit (id);
    visit (list_tail);
    return ;
}
```

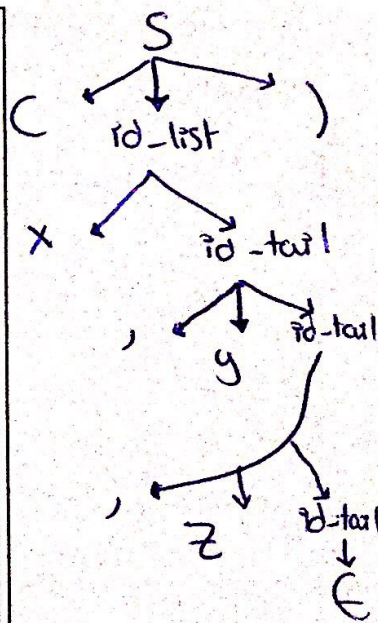
list_tail has two rules
Which one is invoked?

```
Visit Rule3 () {
    visit (id);
    visit (list_tail);
    return ;
}
```

```
Visit Rule4 () {
    return ;
}
```

© All Rights Reserved

Parse tree



Example 1 (cont.)



We write semantic action pass that overrides the routines of the Visitor pattern

```
int count; // global variable
```

```
Visit Rule1 () {
    count = 0;
    visit (id_list);
    return ;
}
```

```
Visit Rule2 () {
    count++;
    visit (list_tail);
    return ;
}
```

```
Visit Rule3 () {
    count++;
    visit (list_tail);
    return ;
}
```

```
Visit Rule4 () {
    return ;
}
```

Note that count is passed down from parents to children while traversing the parse tree

© All Rights Reserved

رجوع عدد (rules)
اسم id, Visit
ممكن اسم id Rule
هون بدل
Visit (id)
حطيا count++
رغبت فينا
Variable
هو (2) بيزيد
ار count

هون بيكون
باسم non terminal

هون ما احتجت غير اشي على هاد لفيكشن

القيمة النهائية لـ count = 3

Parse tree

Example 2



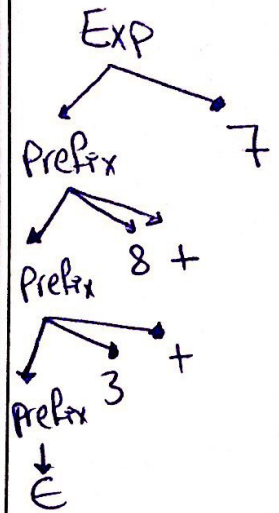
- The grammar describes an algebraic expression of integers that are added to each other
- Write semantic actions to evaluate the expression
 - E.g., when traversing the parse tree of $3 + 8 + 7$, the expression evaluates to 18
 - Assume $GetValue(Token t)$ is a function that returns the integer value of t , where t is a token with the type INT

```

1 Expr      → Expr_prefix INT
2 Expr_prefix → Expr_prefix INT +
3           | ε

```

© All Rights Reserved



Example 2 (cont.)



```

Visit Rule1 () {
    int sum = visit (Expr_prefix);
    sum = sum + GetValue(INT);
    print sum;
    return;
}

```

```

Visit Rule2 () {
    int sum = visit (Expr_prefix);
    sum = sum + GetValue(INT);
    return sum;
}

```

```

Visit Rule3 () {
    return 0;
}

```

Note that sum is passed up
from children to parents while
traversing the parse tree

© All Rights Reserved

default -

```
Visit Rule 1 ( ) {  
    Visit (prefix)  
    Visit (INT)  
}
```

```
Visit Rule 2 ( ) {  
    Visit (prefix)  
    Visit (INT)  
    Visit (+)  
}
```

```
Visit Rule 3 ( )  
{  
}
```

التعديل

```
Visit Rule 3 ( )  
{ return 0; }
```

لأنه آخر مرة يتنادى
يعني 0

```
Visit Rule 2 ( ) {  
    int x = Visit (prefix)  
    x = x + GetValue (INT)  
return x;  
}
```

```
Visit Rule 1 ( ) {
```

```
    int x = Visit (prefix)
```

```
    x = x + GetValue (INT)
```

```
    return x;
```

```
}
```

اطلعومات الي Code: Semantic Attributes



- Semantics attributes are information that describe some meaning of a terminal or a non-terminal in the parse tree
- Information passed from children nodes to parent nodes are called synthesized attributes
- Information passed from parent nodes to children nodes are called inherited attributes

© All Rights Reserved

Semantic Action Passes



- The number and functionality of semantic action passes vary from a compiler to a compiler
- In this course, we will consider the following traditional semantic action passes:
 - First Pass: building the symbol Table (key, value)
 - Second Pass: performing type checking
 - Third Pass: generating the IR

© All Rights Reserved