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

* Front-end → check the meaning (syntax error)

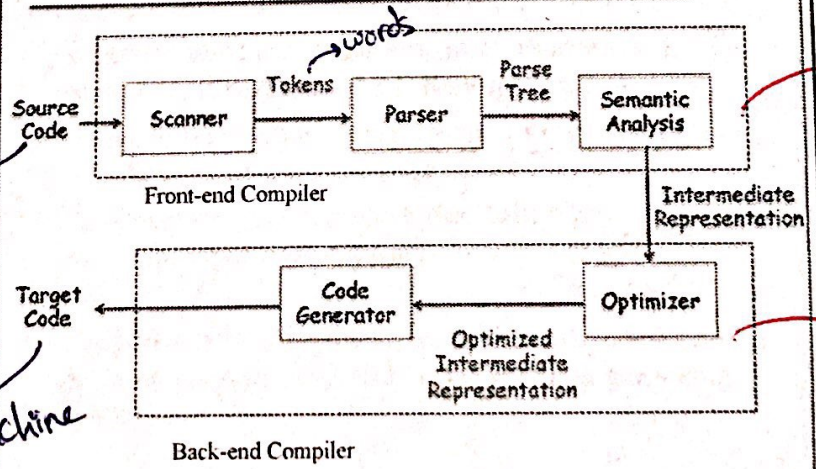
* Back-end → final mapping / translation

1/20/19

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

Compiler Structure



characters عبارة عن
C++ / C فى لغة

assembly / machine

Front-end

Back-end

Scanner → يقرأ الحروف ويحط كل كلمة مع بعض

black box which read a group of characters & يجمع الاحرف المناسبة every عشان يعمل token
كلمة مفيدة
وبصنفها انها keyword أو operator

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

بكون فاهم ال scanner انه f , i عبارة عن if وهي Key word

Tokens (units)

- 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?**

else → right
esle → lexer error
لانه ما نفع هاي الكلمة

I eat apple → structure right
I apple eat → structure illegal

بس هاي مش وظيفة lexer ← يحل هون ما بطلع error
لانه وظيفته يطلع كلمة بعدين النظر عن المعنى

Rules determine the valid & invalid

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

input & tokens

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
     of tokens
  2. Generate a tree representation, called the *parse tree*, of the input code structure
     valid rules
- The parser returns a syntax error if the code structure of the input program is invalid for the given programming language

بشيك اذا مكتوب الكود بـ structure صح

for (i=1 ; i< 3; i++)

بشيك اذا انا كاتبة الجملة هاي صح وحاطة الفواصل وهكذا

وكل لغة لها grammer معين بمشي عليه

I eat apple → valid structure

I apple eat → invalid "

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

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

1/20/19

A Sentence in English

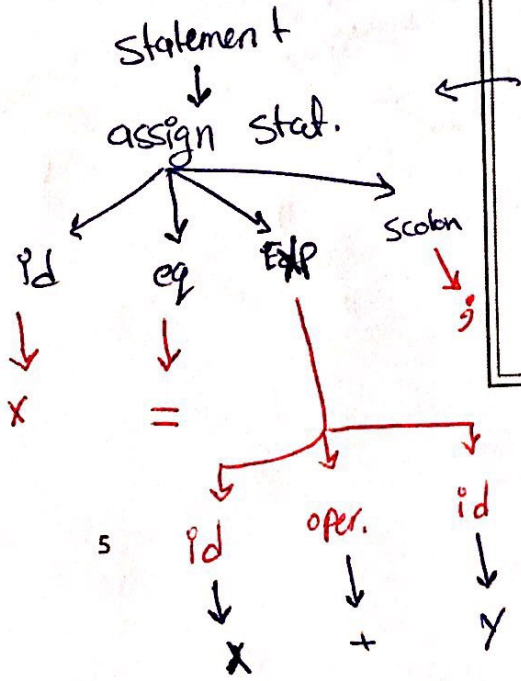
- In English, we have the following production rules
  - A "SENTENCE" can have the structure "SUBJECT VERB OBJECT"
  - A "SUBJECT" can have the structure "PRONOUN"
  - A "VERB" can have the structure "AUXILIARY"
  - An "OBJECT" can have the structure "ADJECTIVE"
  - A "PRONOUN" is "he | she | it | ..."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

زي كأنه عندي Rules جمل بيكل لغة بمشي عليهم

An Assignment Statement In C

مثال

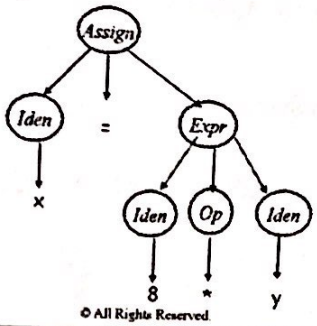
- In C, we have the following production rules
  - A "STATMENT" can have the structure "ASSIGN_STATMENT"
  - A "ASSIGN_STATMENT" can have the structure "IDENTIFIER EQUAL EXPRESSION SCOLON"
  - An "IDENTIFIER " is any sequence of [a-z] characters
  - An "EQUAL" is "="
  - An "EXPRESSION" is "IDENTIFIER OPERATOR IDENTIFIER"
  - An "OPERATOR" is "+ | - | * | ..."
  - An "SCOLON" is ";"
- Exercise: use the above rules to validate the structure of the sentence "x = x + y;"



هاي هي ال parse tree

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:



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

هاي مبينة عملية
طبيعية بس اذا
أكاديمية

int ← X
string ← y

هون وظيفة ال
semantic
error

front end → understand the meaning

I eat car → lexer ✓
→ parser ✓
→ semantic analysis X

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)*

الأساس

بشوف كل . ع
شو نوعه

x	int
y	boolean
fo	func

من هاد الجدول
بشيك على
semantic errors

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

int x
string x
in the same fun.

ما حدد
نوعه

foo (int x, int y)
foo(3)
X
هو لازم يعطيها

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

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

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
```


my own بعمل
assembly language
بس مش
assembly machine language

⇒ Back-end

Phase 4: Optimizer

- The purpose of this phase is to improve the translated program in some discernable way
  - Minimize execution time
  - Minimize memory footprint
  - 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

يعدل على
الكود بس
- كافظ على نفس
على هاد الكود

* هون ال compile بقدر يعدل
على الكود

* بس الكومبايلر ما بعرف قبل ما ينفذ
وين رح يتخزن ال var. وكم رح ياخذ وقت تنفيذ

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?

b=a → dead statement

↳ sth I write & never read ⇒ Dead

فبشلحها الكومبايلر

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

بعتمد على hardware architecture (accelerators مثلاً)

Ex: a=1
if (x>2)
{ a = 5
}
g = a + 20;

* ما بعرف قيمة a لازم يضل يشيك على x كل مرة

* أما اذا عملت حسابات وقدرت أعرف انه قيمة x أكبر من 2 بلغي كل الجمل اللي بالاول وبعوض g = 5+20

* هاد أهم شي I don't have full time Knowledge ما بعرف التايم كامل

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!

make more money!

* بشو الفائدة من optim.

اهم الكلمات السابقة ثابتة!!

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

بتفترض virtual regesters.

كول لل virtual reg. لل reg. الي عندي كل اللغة

* بمسك الكود الي من my own language

كول لل assembly lang. الصح

Let us Put Everything Together

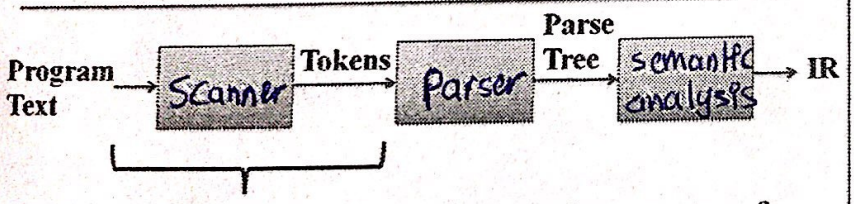
- The compiler is split into two parts: the front-end and back-end
- The front-end (scanner + parser + semantic analyzer) validates the input program and produces an intermediate representation
- The back-end (optimizer + machine code generator) optimizes and maps the intermediate representation into an equivalent machine code
- The selection of the IR affects the back-end design
- We can build multiple front-ends for a particular back-end
- We can build multiple back-ends for a particular front-end

lecture 3

Scanners / Lexers
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 =

they have value & Type

* input for Scanner :- Text (Bunch of characters)

* output " " :- Tokens

Basic Questions For Scanners

1. How tokens are defined?

   ⟹ Regular Expressions (set of Rules)

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

   some algorithms

3. How scanners are coded?

   C++/C / حسب اللغة - جافا

set of Rules to generate strings

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 $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

* each language have alphabits

Ex: binary (0,1) alphabits

010 , 111 , 100 → valid

01ba → invalid

* Some special Characters:

$\phi \rightarrow$ strings لغة ما فيها

$\varepsilon \rightarrow$ " " string فاضي

$S^* \rightarrow$ zero or more characters (عادي يكون فاضي)

$S^+ \rightarrow$ one or more "

Ex $R = 0^* \rightarrow \varepsilon, 0, 00, 000 \ldots$

$R = 0^* 1^* \rightarrow 000111 \ldots$ قد ما بدي أصفار بعدها قد ما بدي واحدات

$R = 0 \Rightarrow$ single char

ε valid

Ex $R = (0/1)^+$ (or)

010110 ✓

0 ✓

1 ✓

1111 ✓

ε X

Basic Regular Languages

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

Basic Operations For Building Regular Expressions

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

a (b/c)*

abcb ✓

abbb ✓

accb ✓

⋮

acab. X

* → Zero occurence or more

+ → one occurence or more

Precedence Rule

1. Parenthesis ( )
2. Kleene Closure * / +
3. Concatenation
4. Alternation

from left to right

Regular Expressions Examples

We can use **algebraic notations** to describe regular expressions

Regular Expressions	Examples on valid string patterns	Examples on non-valid string patterns
(01)*2	2, 012, 01012, 0101012	1, 01, 02, 12, 01201
(0|1)*(a|b)*	0, 1, 0a, 0b, 0abab, 1ab, 1bb, 0aa, 001101, 01, 1111, 101ab	a, aa, aba, 0a1, ε
R1 (0|1|2|3|4|5|6|7|8|9)*	R1, R6, R11, R2, R9, R12, R23, R14, R3, R7, R13, R5, R8, R10, R21, ε	R55, R18, R140, R31
[a-z|A-Z]+ @ gmail	a@gmail, Compiler@gmail, comPutER@gmail	@gmail, ε
[illegible]	[illegible]	[illegible]

(01)*2

ما في تكرار

لازم تكتب 8 مع بعض ما بين 1

لازم ينتهي بـ 2

ε X → لازم يخلص بـ 2 فقش هنا هي

[2-9] ≡ 2|3|4|5|6|7|8|9

[a-Z|A-Z] → a-Z is Char أي small or Capital

Tokens and Regular Expressions

- We will now describe the regular expressions of the following tokens:
  - Literals 1, 2, - - -
  - Identifiers variables
  - Comments
  - Reserved words
  - Operators
  - Punctuations ; , ( ) = - -

Often found in programming languages

Ex 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:

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

  $I = D^+ = [0-9]^+$

  $E = D^+ . D^+ = [0-9]^+ . [0-9]^+$

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

افتراضات

عدد صحيح

عدد كسري

Sol

$0 \mid [1-9][0-9]^*$

صفر أو

اي رقم غير الصفر

٠, ١, ٢, ٣, ٤

حاجي راي اصطهون

لانه التكرار بيصير

+ , *

5 ومن القوى التي

Sol ① [A-Z] ([a-z] | [A-Z] | [0-9])*

② ( )* 00 ( )*

1/20/19

star لأنه هو بس ممكن يكون عندي 1 char وبدي ع

Ex 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])+

بسمحلي أبلش برقم

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

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 = ## ((# | ε) Not(#))* ##

Not(#)* ##

يعني أي شي من Σ ماعدا #

* بس ممكن يكون ال body تاع الكومنت في # وحدة

a#d

* Not → single char

Not (##) X

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 = (')')

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

Keywords = if | for | while - - - -

قوس من اللغة
for (

Exercises

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

① $(a \mid (bc)^* d)^+$ ⟹ aaa-, ad bcd

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.

Sol

Basic Questions For Scanners

1. How tokens are defined?

   ⇨ Regular Expressions

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

   ⇨ Finite Automata FA

1. How scanners are coded?

sol ① $(0 \mid (1-9)(0-9)^*) \cdot (0 \mid (0-9)^*(1-9))$

صفر أو رقم بعده أصفار أو أرقام

صفر أو أرقام أو أصفار بعدين
لازم أرقام

②

$(0|1)^* \; 0 \; (0|1)^* \; 0 \; (0|1)^* \longrightarrow$ على الاقل صفرين

صفرين قبلهم اي شي وبينهم وبعدهم اي شي

$\longrightarrow 1^* \; 0 \; 1^* \; 0 \; 1^* \longrightarrow$ اذا اجبرني بس صفرين

③ $(01 \mid 1)^* \; (0 \mid \varepsilon)$

كل صفر يجي بعده واحد

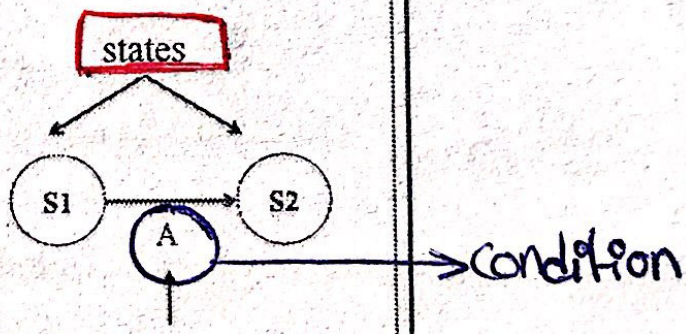
أو ما يجي بصفر

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 ∈ 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

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
- FMSs memorize previous events (how?)



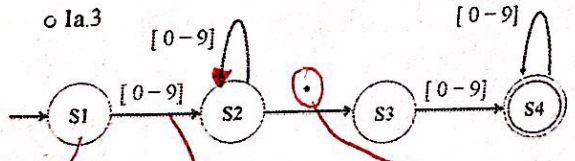
A transition from state S1 to S2 occurs when event A is triggered

Finite Automata (FA)

- A FA consists of
  1. A finite set of states S
  2. A start state n (initial state)
  3. A finite alphabet Σ
  4. A set of accepting states G, where G ⊆ S → End state
  5. A set of transitions $s_i \xrightarrow{a} s_j$, where a∈Σ, s_i, s_j∈ S
     → conditions btw states
- 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

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) ?
    - 090.011
    - 10.
    - 1a.3



initial state

event/condition (أي رقم من 0 – 9)

أي رقم نوقعه

* بس أخلص الرقم اللي عندي اذا كنت بـ S4 ← accept

اذا كنت بوحدة من اللي قبل ← reject.

* في لكل state هم صفني احد Error state ما يسير غلط بروح عليه

Ex ① 09.100 ← وصل S4

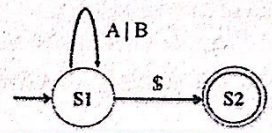
② 100. X
10
وقف عند S3

③ 1a.2 X
وقف عند S2

A$B → رح توصل S2 بس رح تروح على error state

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$$



ليش سواها هيك ؟؟

2 tokens

ABB$ $

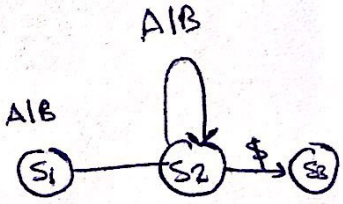
حسب الـ implementation

شلت هاد الكوبايل

space

أو ε

عشان مش مهم يعلش ب (A|B)



هو مجرد يعلش بحرف

$(A|B)^+$ $

Finite Automata Types

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

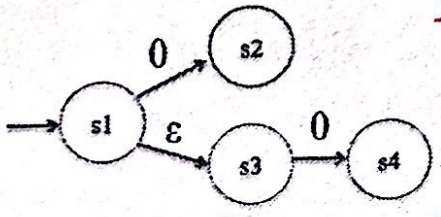
unconditional move

لما كمل الموضوع أكثر من حل بتكون undeterministic

ε–Transitions

- The ε-transition $s \xrightarrow{\varepsilon} w$ 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

مش مبين عندي صفر أو اثنين ثاني



بس يوصل اك بتجوز

DFA and NFA

DFA is simple

- 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

Thompson Subset

R → NFA → DFA

our Target.

Finite Automata Construction

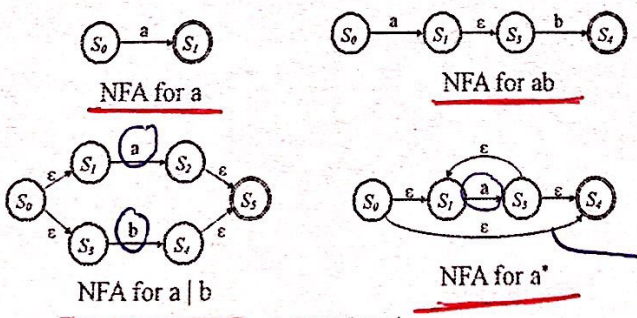
1. Thompson's construction
   - An algorithm that generates an NFA for a regular expression
2. Subset construction
   - An algorithm that converts an NFA into a DFA

Thompson's Construction

we use $\boxed{\varepsilon}$ To make concatenation

Key idea is simple

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



zero occurance

أشكال ثابتة

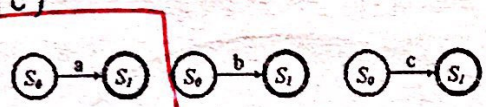
$a^+ \rightarrow$ same a^*

بس بشيل السلك اللي من S_0 لـ S_4

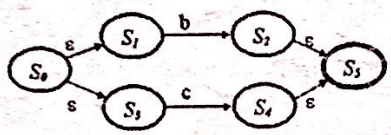
Thompson's Construction Example

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

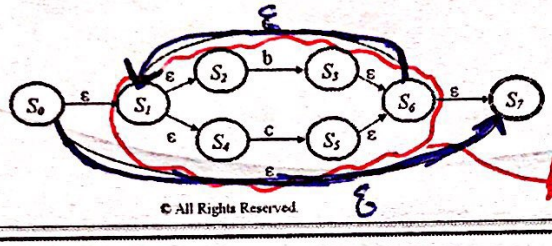
1. a, b, & c



2. b | c



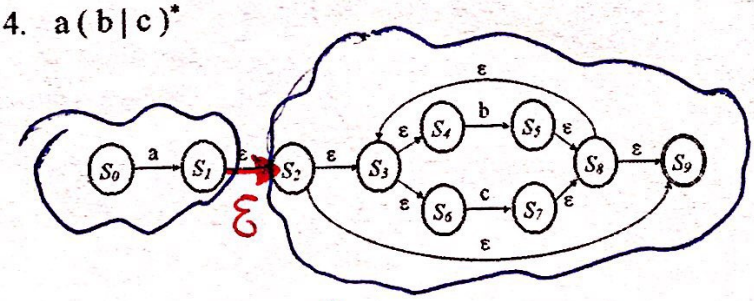
3. (b | c)*



block (b|c)

Thompson's Construction Example (cont')

4. a (b | c)*



concatination

* we can optimize it by removing ε
* both solutions are 100% right

* اختياري نعمل optimize بس لا يفضل

Exercise

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

change from NFA to DFA

Subset Construction

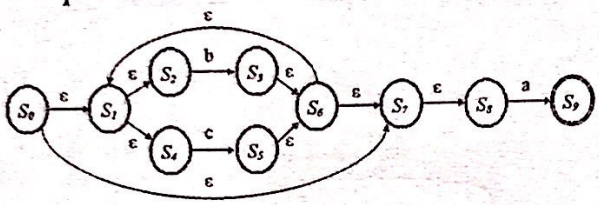
- Transforms an NFA N into an equivalent DFA D
- The algorithm associates each state of D with a set of states of N
- The algorithm uses two key functions:
  - ε–Closure(x): returns the set of states reachable from x by ε
  - Move(X, a): returns the set of states reachable from X by a, where $a \in \Sigma$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

functions

it can reach itself

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:



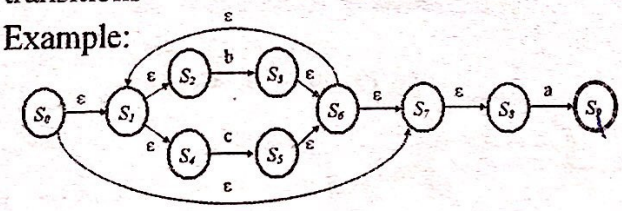
ε-closure (S_0) = { S_0 , S_1 , S_2 , S_4 , S_7 , S_8 }
ε-closure (S_3) = { S_3 , S_6 , S_1 , S_2 , S_4 , S_7 , S_8 }
ε-closure (S_4) = { S_4 }

أي سهم عليه ε +
لا S نفسها

ε-Closure(S_2) = S_2

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:



Move ({S_3 , S_6 , S_1 , S_2 , S_4 , S_7 , S_8} , b) = { S_3 }
Move ({S_3 , S_6 , S_1 , S_2 , S_4 , S_7 , S_8} , c) = { S_5 }
Move ({S_3 , S_6 , S_1 , S_2 , S_4 , S_7 , S_8} , a) = { S_9 }

مين اللي بقدر أوصلهم من خلال a

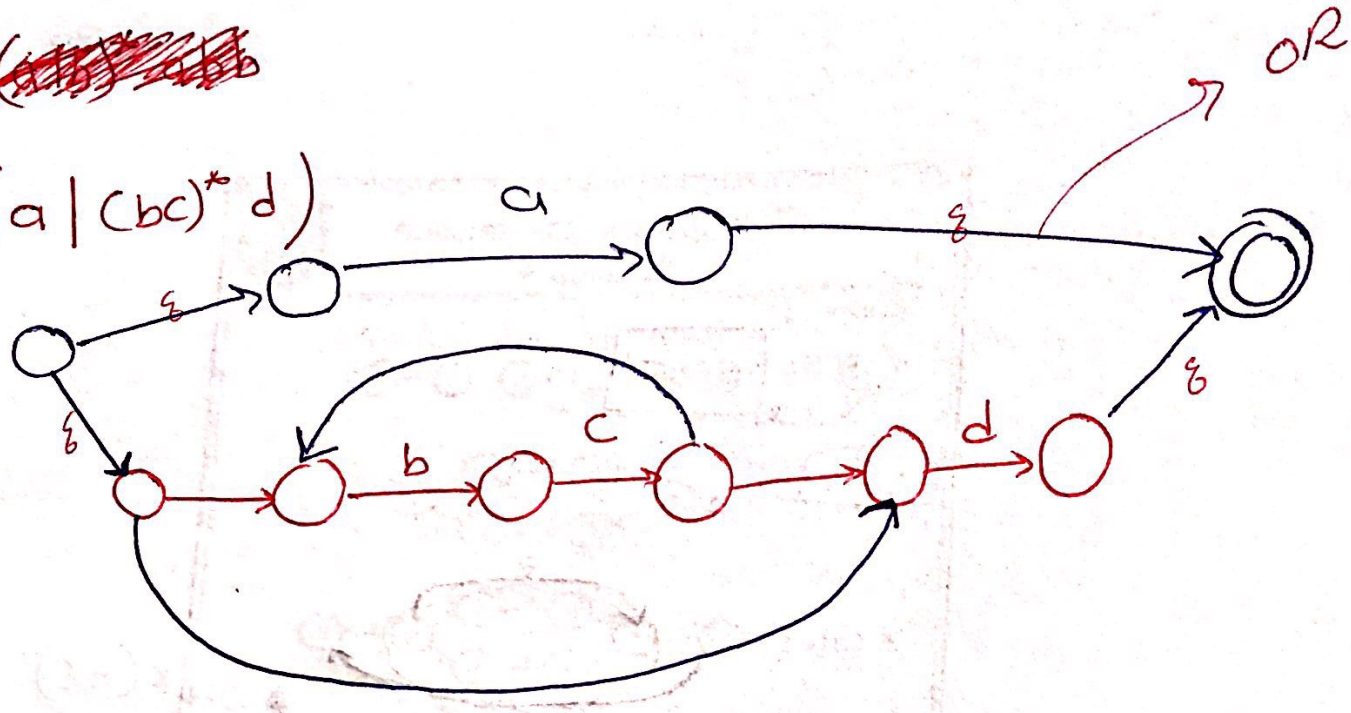
* بطلع على الأسهم اللي طالع منها a وباخذ ال state اللي راح عليها

* عادي تكون empty

اما ε-closure مستحيل تكون empty لانها بتاخذ نفسها

.sol Excersise

$(a \mid (bc)^* d)$



* شلت ε اللي بين D , bc

optimize straight forward يا

sodo code to translate to DFA

Subset Construction Algorithm

Input: NFA N
Output: DFA D

Note that N and D will have the same Σ

```
d0 ← ε-closure({N.start_state})
D.states ← { d0 }
WorkList ← { d0 }
while ( WorkList ≠ ∅ )
  select and remove s from W
  for each α ∈ Σ
    t ← ε-closure(Move(s,α))
    add s --a--> t to D.transitions
    if ( t ∉ 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

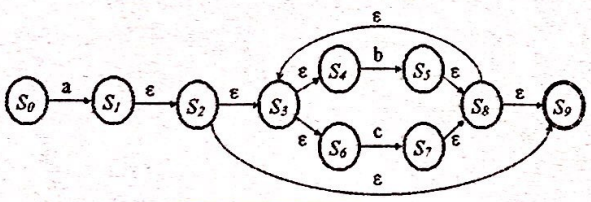
Associate a set of states in N with a set in D

Iterate till no more states are added

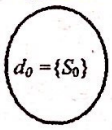
2-fun الأعلى

Subset Construction Example

- Let us try the following NFA



1. Compute $d_0 \leftarrow \varepsilon\text{-closure}(\{S_0\})$



① كتب الـ do

do = ε-closure (So)

= So

worklist = do

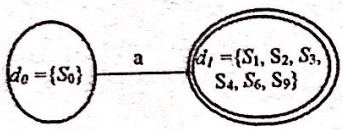
DFA	NFA
Do	So

DFA	NFA
d_0	S_0
d_1	$S_1, S_2, S_3, S_4, S_6, S_9$

1/20/19

Subset Construction Example (cont.)

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



move$(d_0, a) = S_1$

move$(d_0, b) = \phi$

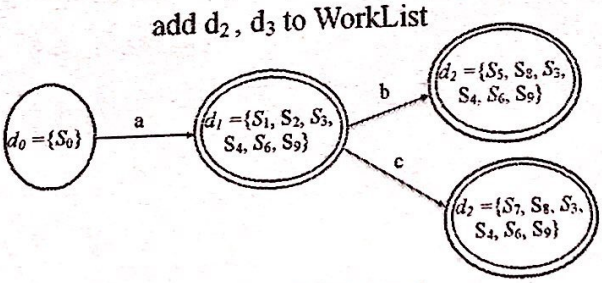
move$(d_0, c) = \phi$

$S_1 \xrightarrow{\varepsilon\text{-closure}} S_1, S_2, S_3, S_4, S_6, S_9 = d_1$

* worklist = ~~d_0~~, d_1

Subset Construction Example (cont.)

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



DFA	NFA
d_0	— —
d_1	— —
d_2	— — —
d_3	

move$(d_1, a) = \phi$ X

move$(d_1, b) = S_5 \xrightarrow{\varepsilon\text{-closure}} \{S_5, S_8, S_9, S_3, S_4, S_6\}$ d_2

move$(d_1, c) = S_7 \xrightarrow{\varepsilon\text{-closure}} \{S_7, S_8, S_9, S_3, S_4, S_6\}$ d_3

* worklist = ~~d_0~~, ~~d_1~~, d_2, d_3

move$(d_2, a) = \phi$

move$(d_2, b) = S_5 \xrightarrow{\varepsilon\text{-closure}} \{$

move$(d_2, c) = S_7 \xrightarrow{\varepsilon\text{-closure}} \{$

$\rightarrow d_2$

$\}$

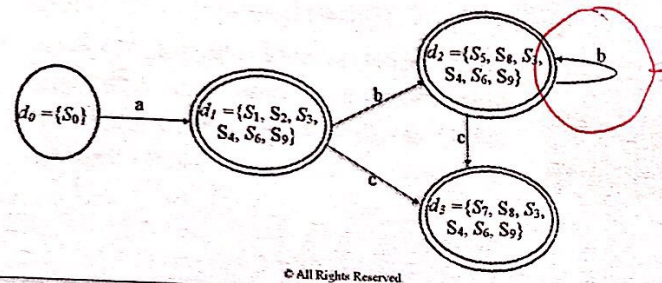
$\}$

$\rightarrow d_3$

1/20/19

Subset Construction Example (cont.)

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

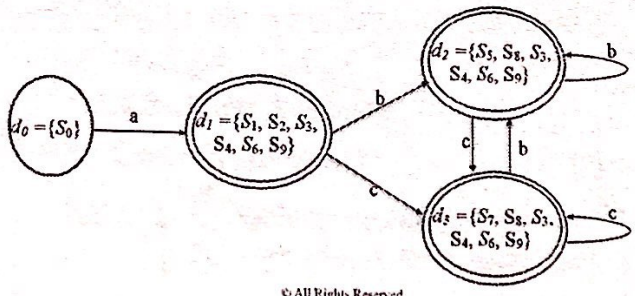


اذا انا بـ d_2 و بجيلي b برجع على d_2

work list = $\cancel{d_0}, \cancel{d_1}, d_2, d_3$

Subset Construction Example (cont.)

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



move$(d_3, a) = \phi$

move$(d_3, b) = S_5 \xrightarrow{\varepsilon\text{-closure}} d_2$

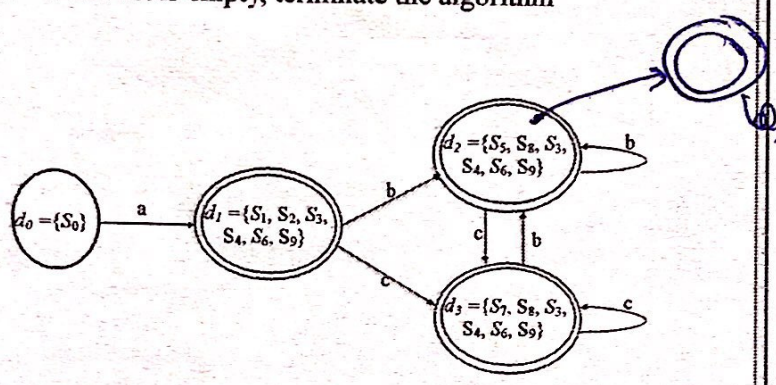
move$(d_3, c) = S_7 \longrightarrow d_3$

worklist = $\cancel{d_2}, d_3$

work list = ~~d3~~ (emptyset)

Subset Construction Example (cont.)

6. WorkList is empty, terminate the algorithm



accept

اللي بقدر ارجع عليها قد ما بدي

* الجواب مش شرط يطلع مصغر ⇐ ممكن أبسط أكثر
humen

Exercises

- Construct a DFA for the following regular expressions:
  - (ab | ac)*
  - (a | (bc)* d)
  - a b* c | a b c*

* بس يجي سؤال بالامتحان :-

خط الجدول

مش مطلوب

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

Basic Questions For Scanners

1. How tokens are defined?

   ⇒ Regular Expressions

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

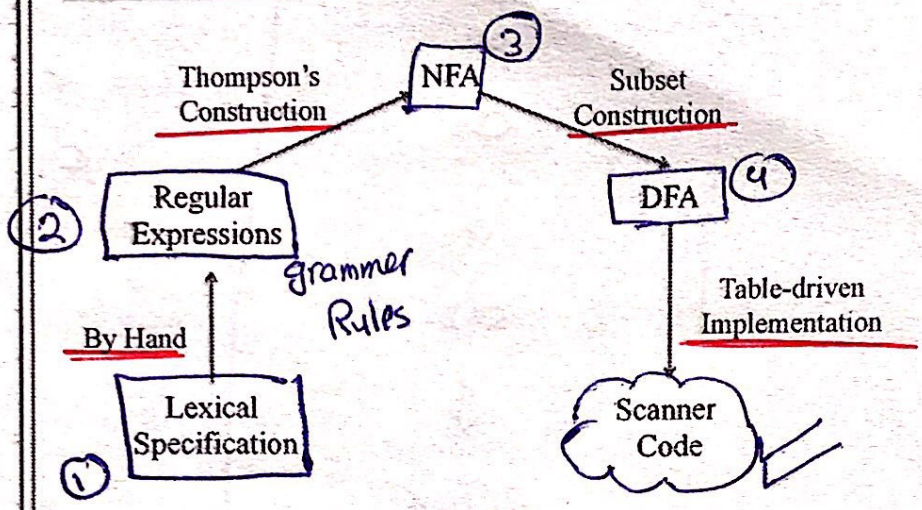
   ⇒ Finite Automata

1. How scanners are coded?

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

كيف نكتب كود يمثل ال
regular expressions

Scanner Generation Cycle



Coding The DFA

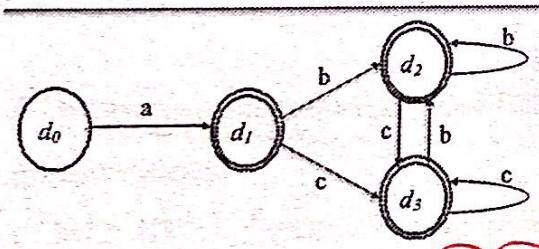
- Multiple methods have been used to implement scanners
- We will consider the table-driven form, the more common method for building scanners
- Table-driven scanners take advantage of *transition diagrams*: a tabular representation of FA

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

عندي state بدي احولها ل table

Transition Table Example



By default, blank entries have error flags

Alphabet Characters

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

* التقاطع بين do & a ← d1

* الفاضيين ← error state

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 */
```

invalid.

sudy code

Automatic Scanner Generators

Regular Expressions → Scanner Generator → **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

كيف عكن أشيك من ال table اذا ال String

مع ؟؟

① ~~a~~ ~~b~~ ~~b~~ ~~c~~

d_0
↓
d_1
↓
d_2
↓
d_2
↓
d_3

Valid ✓

② ~~a~~ ~~b~~ ~~a~~ c

d_0
↓
d_1
↓
d_2
↓
error

X
invalid.

③ a

d_0
↓
d_1 ✓

* اذا وصلت لا accept state بيكون valid

* ، ما ، ، ، ، ، او وصلت لـ error

↓

invalid.

code generater

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

في grammer Rules

بيعطيني الكود جاهز

لل Scanne & parser

input:- hello fahed ✓ accept

-hello # ✗

ANTLR Input Example

بدي خايك يكتب

hello ___ +

```
// 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
```

v4

ثابت دائماً

اي حرف بعد hello

take comments

هاي الاشياء ما بيشوفها ال scanner

ANTLR Tool Output

- Let us run the ANTLR tool on the grammar example

$ java –jar antlr-4.5.3-complete.jar Hello.g4

input

جافا
تعمل run
لـ jar

- The ANTLR tool generates the following outputs:

Scanner code

Scanner

- HelloLexer.java
- Hello.tokens
- HelloLexer.tokens

Parser

- HelloParser.java
- HelloListener.java
- HelloBaseListener.java

- Read more from the ANTLR v4 Book

https://pragprog.com/book/tpantlr2/the-definitive-antlr-4-reference

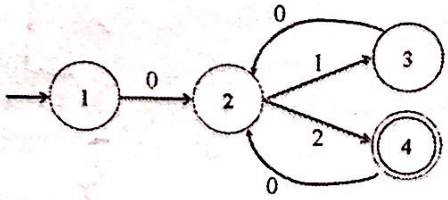
ملخص

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

امتحان

Exercise



alphabits → 0, 1, 2

states → 1, 2, 3, 4

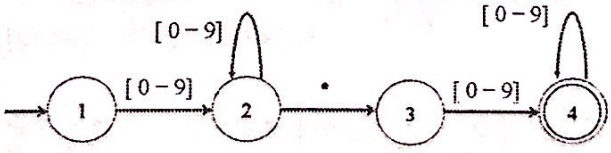
Which of the following strings is accepted by the above DFA

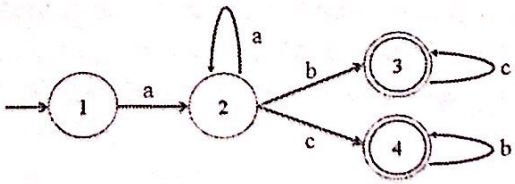
- 0202
- 01010
- 0102012
- 020102

sol

Exercise

- Specify the transition tables for the following DFAs





raw → 4 (states)

Colomn → 11

(0 – 9) + •

10 + 1 = 11

Sol Exersise

ololo

1
↓
2
↓
3
↓
2
↓
3
↓
2

X reject

لأنه ما وصلت 4

الي هي accept

020102

1
↓
2
↓
4
↓
2
↓
3
↓
2
↓
4

✓ accept

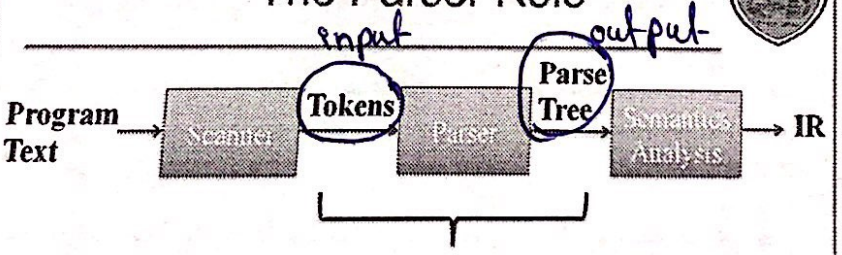
lecture 4

Introduction To Parsing
Spring 2018/2019

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

step 2 of front end.

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

I eat apple ⟹ pronoun → verb → object.

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

الفكرة الأولى
derivation
لكل ال sentence

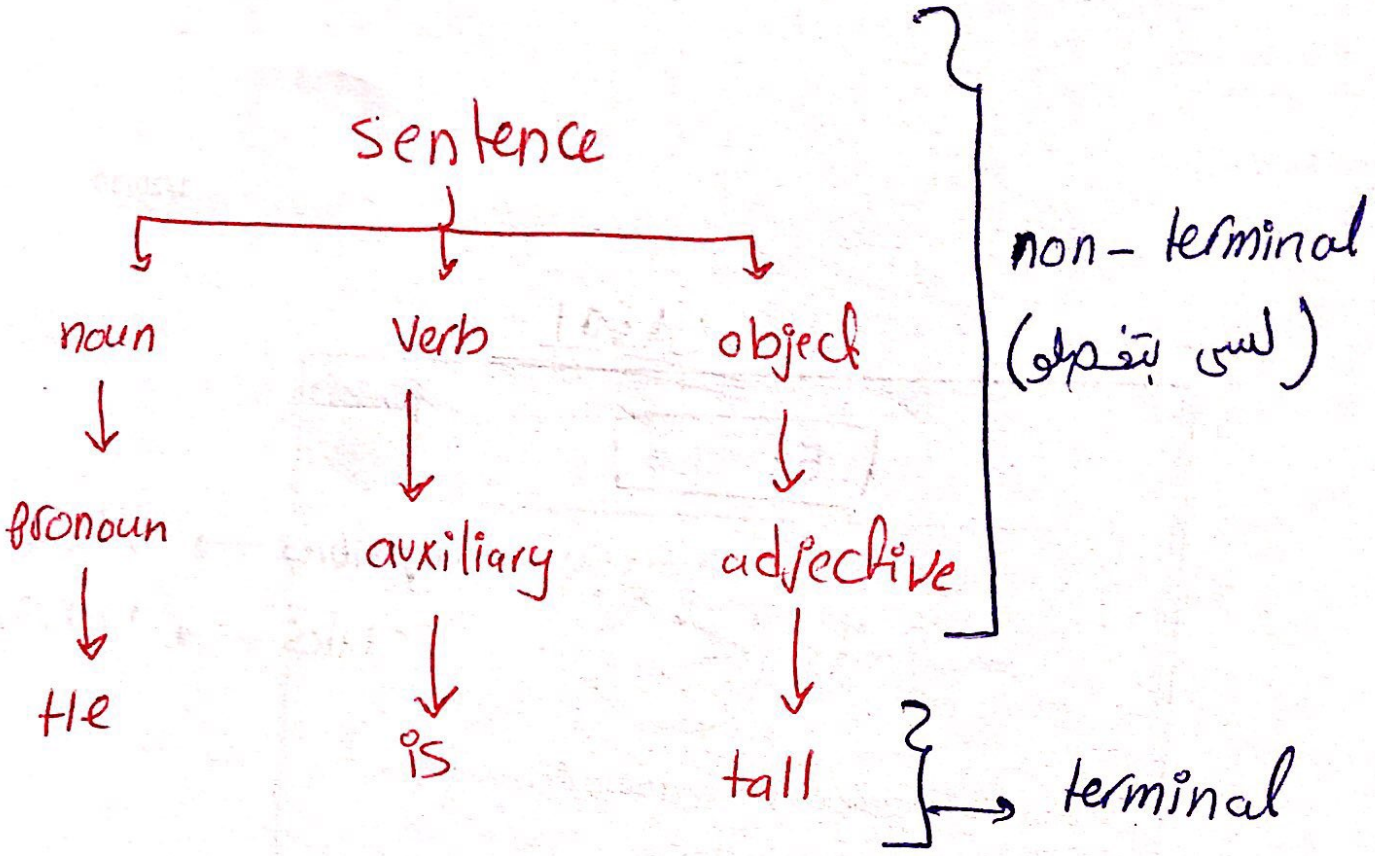
*ex

خطوتين لل Parser

Roadmap

- First, we will study context-free grammars, a mathematical model for describing syntax in programming languages → cfg
- 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

Ex he is tall



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

Nonterminal symbols

Production rules

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

Terminal symbols

Alternation

Start symbol

حط أقواس قدما بدك

بقدر أستبدل B بأي حرف منه هدول

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.	|	[ ]

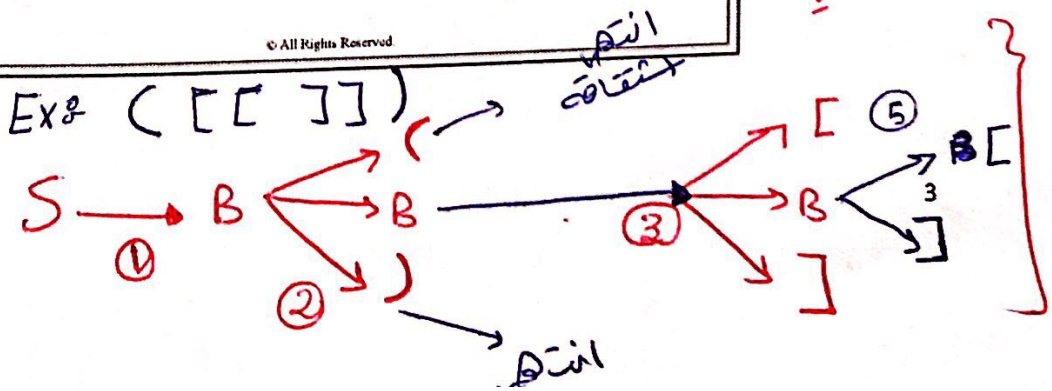
S ⇒ B	Start with rule 1
⇒ [B]	Use rule 3
⇒ [[B]]	Use rule 3
⇒ [[( )]]	Use rule 4

الي بلش فوق منه

مثال لفت اقواس

خلص اشتقاق كل حرف

Ex: ([[ ]])



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 → (T ∪ NT)+; that is, it replaces a single nonterminal with a string of one or more grammar symbols

CFG Examples

Specify the terminals and the nonterminals for each grammar

1.	E	→	*num* Op *num* Etail $
2.	Etail	→	Op *num* Etail
3.		|	ε
4.	Op	→	+
5.		|	−

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

قد ما بدي أرقام
وكل الأول اتمين
25 X

* terminal → +, −, ε, num, $
* non terminal → E, Etail, op

Hw → (X, Y, Z)

(*) Ex slide 3

[[()]]

Sol

$$S \xrightarrow{1} B \xrightarrow{3} [B] \xrightarrow{3} [[B]] \xrightarrow{4} [[()]]$$

* Rule 4 & 5 ⟶ when there is no recursion
* " 2 & 3 ⟶ " " " "

(*) Ex slide 4

2 - 3 + 1 $

$E \xrightarrow{1}$ num op num Etail $ $\xrightarrow{5}$ num - num Etail $

$\xrightarrow{2}$

num - num op num Etail $

$\xrightarrow{4}$

num - num + num ~~Etail~~ $

ε

$\xrightarrow{3}$

num - num + num $

* بنتظر الـ derivation بس توصل لحد N

Derivation Process

- A derivation consists of a series of rewrite steps

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

final sentention (terminal)

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)

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

Parse Tree

- Parse tree: a directed graph that represents a derivation

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

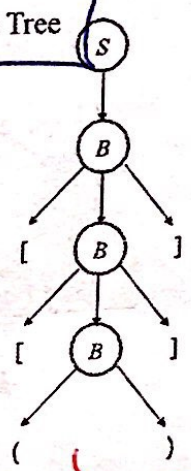
Grammar

1.	S	$\longrightarrow$	B
2.	B	$\longrightarrow$	(B)
3.		|	[B]
4.		|	( )
5.		|	[ ]

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

Tabular represet.

Parse Tree



tree representatio

Parse Tree Properties

- A parse tree has
  - The start symbol at the root
  - Terminals at the leaves
  - Nonterminals at the interior nodes

الاب
childes

- A post-order traversal of the leaves yields the original input string
- The parse tree shows which operands associate with which operations

Parse Tree

- Parse tree: a directed graph that represents a derivation

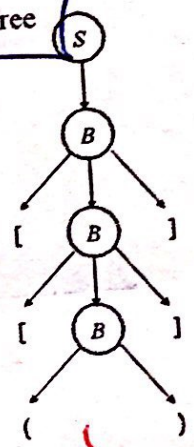
$S \Rightarrow^* [[()]]$

Grammar

1.	S	$\longrightarrow$	B
2.	B	$\longrightarrow$	(B)
3.		|	[B]
4.		|	( )
5.		|	[ ]

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

Parse Tree



Tabular represet.

tree representation

Parse Tree Properties

- A parse tree has
  - The start symbol at the root
  - Terminals at the leaves
  - Nonterminals at the interior nodes

الـ childes

- A post-order traversal of the leaves yields the original input string
- The parse tree shows which operands associate with which operations

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

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: '*'

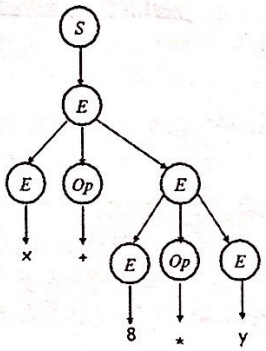
$S \rightarrow E \xrightarrow{5} E op E \rightarrow id\ op\ E \rightarrow id + E$

$\rightarrow id + E op E \rightarrow id + \underset{num}{\cancel{\ }} * id$

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>



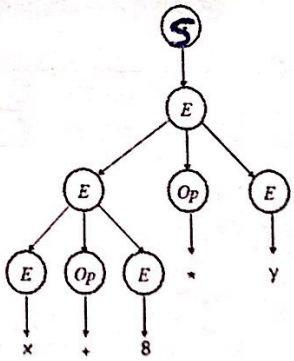
A post-order tree walk of this parse tree evaluates as x + (8 * y)

هون بحل الضرب بعد ين الجمع.

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

طريقة حل ثانية

بجمع بعدين بضرب

⇓

صار عنا ambiguas Grammer

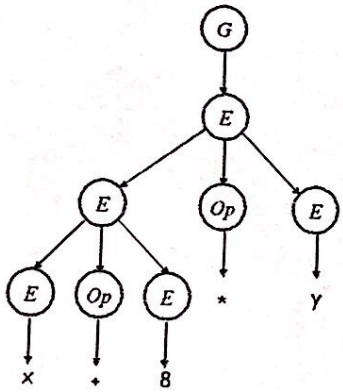
واضح (الكل طريقة و طريقة)

بعين غلط يطلع عنا أكثر من شجرة

Rightmost Derivation

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

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

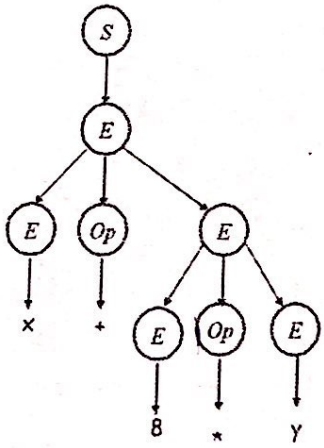


A post-order tree walk of this parse tree evaluates as (x + 8) * y

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 <id,y>
6	E Op E * <id,y>
4	E Op <num,8> * <id,y>
5	E + <num,8> * <id,y>
3	<id,x> + <num,8> * <id,y>



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

2 parse tree

أكثر من

⟹ Problem

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

compiler don't Know what to do

Eliminating Ambiguity

- To disambiguate an ambiguous grammar, rewrite it **by hand**

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

Ambiguous Grammar

1.	S	→	E
2.	E	→	E *plus* Ê
3.		|	Ê
4.	Ê	→	*id mul* Ê
5.		|	*num mul* Ê
6.		|	*id*
7.		|	*num*

Rewritten Grammar: we gave multiplication precedence over summation

E → +

Ê → *

بسيرله expansion

بعد E بين أولويتها أعلى

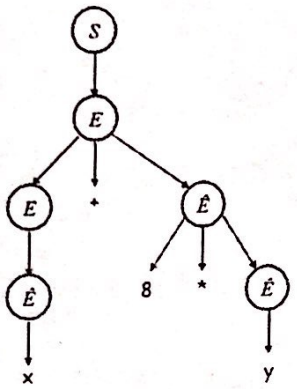
* اللي بكون تحت بالشجرة هو ال high priority

* دائماً اللي له أولوية يكون $\hat{E}$
بحيث أوصله قبل.

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	$\langle id,x\rangle + \langle num,8\rangle * \langle id,y\rangle$

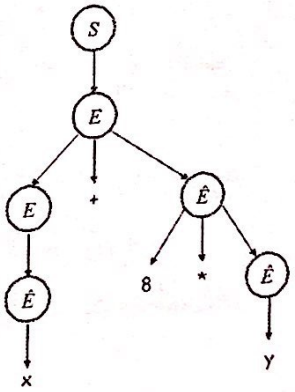


Leftmost derivation of $S \Rightarrow^* x + 8 * y$ is unique

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

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

Example: Algebraic Expression Grammar

Straightforward grammar is ambiguous

1	*Start*	→	*Expr*
2	*Expr*	→	(*Expr*)
3		|	*Expr* Op *Expr* → red flag (left & right side equal)
4		|	*number*
6		|	*id*
8	*Op*	→	+
9		|	-
10		|	*
11		|	/

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

( ) *

↓

* \

↓

+ −

* عندي ( )

فبدي أضيف الشيء

+ − و *\

صفت Term و factor هشان يسير
عندي ٣ non-terminal يعبرو عن الاولويات اللي عنا

Adding Precedence to Algebraic Expression Grammar

1	Start	→	Expr
2	Expr	→	Expr + Term
3		|	Expr - Term
4		|	Term
5	Term	→	Term * Factor
6		|	Term / Factor
7		|	Factor
8	Factor	→	(Expr)
9		|	number
10		|	id

Addition and subtraction, last

Multiplication and division, next

Parentheses have highest precedence

حكي أرجع لفوق

لازم بنهاية الجملة

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

واضح

If-then-else Problem

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

1.	STMT →	if EXPR then STMT
2.	|	if EXPR then STMT else STMT
3.	|	… *other statements* …

Key word

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

" if $expr_1$ then if $expr_2$ then $stmt_1$ else $stmt_2$ "

if _____
if _____
else _____

مش مبين ال else طين تابعة (أي if)

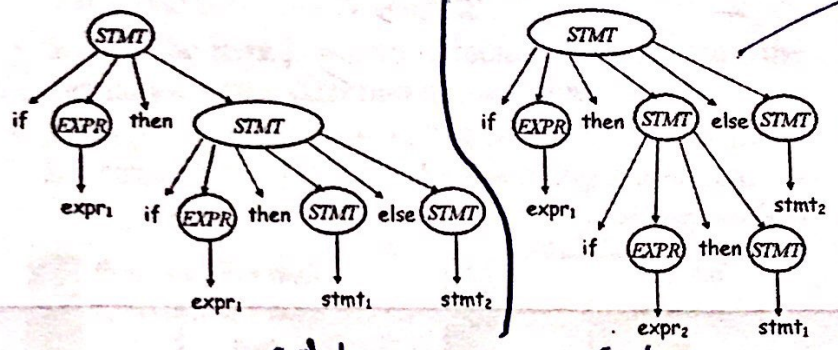
⇓

ambiguous

الحل احتمالين

Leftmost Derivations

STMT ⇒* if $expr_1$ then if $expr_2$ then $stmt_1$ else $stmt_2$



sol 1 sol 2

Two parse trees → meaning is ambiguous

else تابعة لل if الأولى

Solution

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

1	*STMT*	→	if *EXPR* then *STMT*
2		|	if *EXPR* then *STMT#* else *STMT*
3		|	*Other Statements*
4	*STMT#*	→	if *EXPR* then *STMT#* else *STMT#*
5		|	*Other Statements*

Intuition: once into *STMT#*, we cannot generate an unmatched else

أعدل على grammar

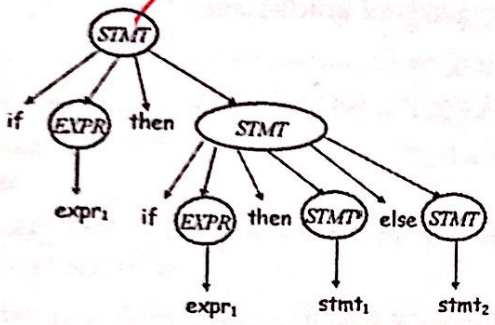
*in general

بعتبر ال else تابعة لل if الأقرب (الثانية)

الها قاعدتين بشوف مين بتعيش معي
⇐ قاعدة واحد بتزيلها معي لانه فيها statent
بتسهلي اختار القابلة

Parse Tree With Rewritten Grammar

STMT ⇒* if $expr_1$ then if $expr_2$ then $stmt_1$ else $stmt_2$



Parse tree is now unique

Is There An Algorithm To Do it?

→ No

- 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* → has no general solution
- To deal with ambiguous grammars, compiler writers:
  1. Modify context-free grammars **by hand** and ensure their unambiguity (sol)
  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

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	→	E + num	1	E	→	num + E
2		|	E - num	2		|	num - E
3		|	num	3		|	num

Left-associative grammar Right-associative grammar

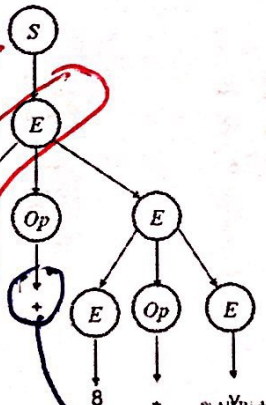
we usually choose left assosiative لأنها تجنب من الاشتباه اليمين

Abstract Syntax Tree (AST)

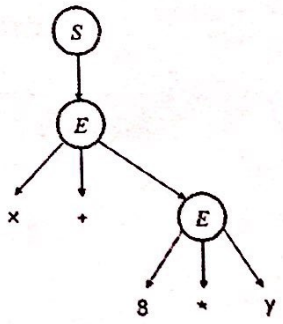
optimized of parse tree

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

Parse tree



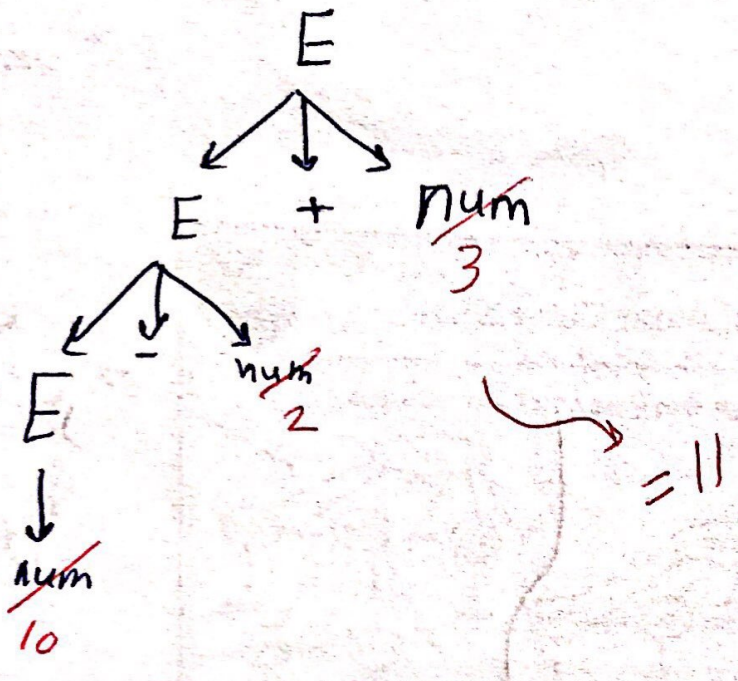
AST: some interior nonterminal nodes are removed



هون فيش داعي لل E طالبة إلا

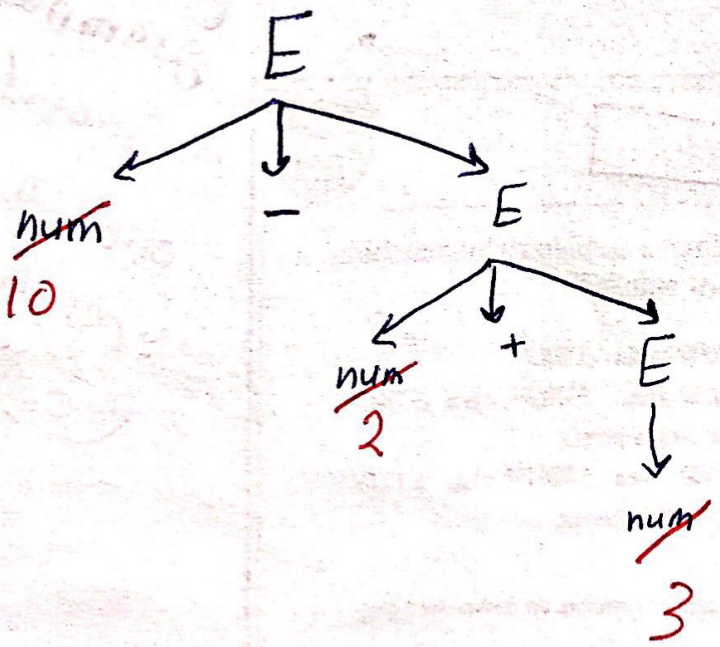
بعد ارتفع زائد لفوق

* left grammer (10 − 2 + 3)



= 11

* right grammer



= 5

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

Exercise

1	S	$\rightarrow$	$A\$$
2	A	$\rightarrow$	$0A$
3		$\mid$	B
4	B	$\rightarrow$	$1B$
5		$\mid$	ε

0 و 1 و $

S , A , B

- 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$"

H.W

$0^*\ 1^*\ \$$

لانه ممكن A
تروح على B

ممكن B
تروح على ε

$S \rightarrow 0A1$

0^* $A \rightarrow 0A$ → recursion

$\mid \varepsilon$

$S \rightarrow 0 \mid A$

$A \rightarrow 1A$

$\mid \varepsilon$

Exercise

- Write a context-free grammar that describes the same language as the regular expression 0^+1
- Write a context-free grammar that describes the same language as the regular expression $0\,1^+$
- Write a context-free grammar that describes the same language as the regular expression $0^+1 \mid 0\,1^+$

$S \rightarrow 0A1 \mid 0 \mid B$

$A \rightarrow 0A$

$\mid \varepsilon$

$B \rightarrow 1B$

$\mid \varepsilon$

Exercise

- Is the following grammar ambiguous? Justify your answer

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

sol string that have 2 parse trees

Top-Down Parsing
Spring 2018/2019

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

The Parser Role

Input

output

- 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

* ما بيكون عندي leaf لا شي non-terminal

Top-Down Parsing

- A basic top-down parsing algorithm:
  1. Construct the root node of the parse tree
  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*	→	*Expr*
2	*Expr*	→	*Expr + Term*
3		|	*Expr - Term*
4		|	*Term*
5	*Term*	→	*Term * Factor*
6		|	*Term / Factor*
7		|	*Factor*
8	*Factor*	→	*(Expr)*
9		|	number
10		|	id

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

left recursion

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

Input Stream (the arrow points to the next input token): ⇨ ↑x – 2 * y

next char.

The root is the start symbol ⇨ S

Pick rule 1 ⇨ Expr

Pick rule 2 ⇨ Expr + Term

Pick rule 4 ⇨ Term

Pick rule 7 ⇨ Fact.

Pick rule 10 ⇨ <id x>

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

ما الها rule وحدة

ال fact اعطاني id. كاملة

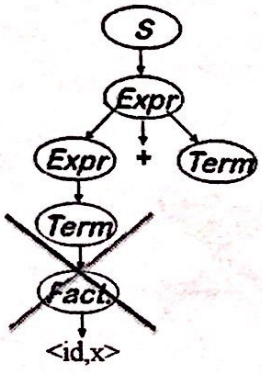
value → X

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

Input Stream (the arrow points to the next input token): ⇨ x↑ – 2 * y

"+" does not match "-" ⇨

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



صار عندي
missmatch
بين + و -

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

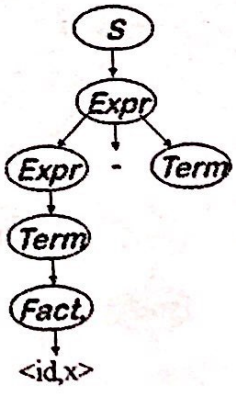
Input Stream (the arrow points to the next input token): ⇨ ↑x – 2 * y

Pick rule 3 ⇨

Pick rule 4 ⇨

Pick rule 7 ⇨

Pick rule 10 ⇨



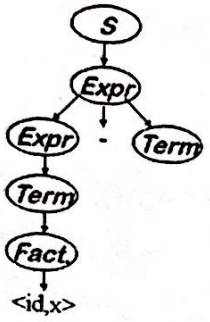
"x" matches "id" type → advance to the next input token

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

Input Stream (the arrow points to the next input token): ⇨ x↑ – 2 * y

"-" matches "-" ⇨

advance to the next input token



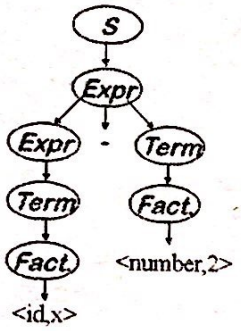
match عندي
عند الطالب

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

Input Stream (the arrow points to the next input token): ⇨ x – ↑2 * y

Pick rule 7 ⇨

Pick rule 9 ⇨



"2" matches "number" type → advance to the next input token

وهل pointer عند 2

Term

اختار بالصدفة Factorial

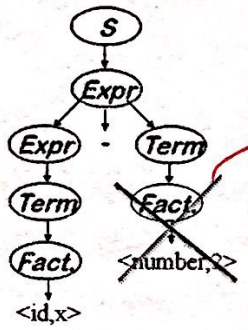
بس لما اختارها خلصت والمعادلة الي عندي من خالصة

Back track ⇐ parss tree

parss tree خلص

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

Input Stream (the arrow points to the next input token): ⇨ x – 2↑ * y



No nonterminals left to expand ⇨

input stream is not fully consumed yet

Parse tree terminated too soon

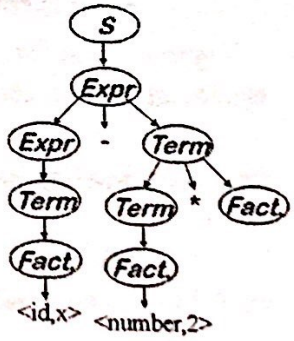
The algorithm backtracks

 11

يرجع
بدل ما اختار
fact
بختار
term

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

Input Stream (the arrow points to the next input token): ⇨ x – ↑2 * y



Pick rule 5 ⇨

Pick rule 7 ⇨

Pick rule 9 ⇨

"2" matches "number" type → advance to the next input token

 12

* this is very Bad for performance

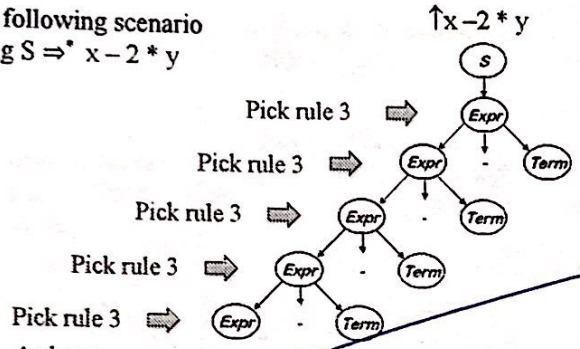
→ optimized. في رح نعدلها بعدين

بس تجي يسوي back track يرجع يختار الطرق

infinite recursion يسير.

One Possible Bad Scenario

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



And so on

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

17

not good idea for top down parsing!

The Left Recursion Problem

1	*S*	→	*Expr*
2	*Expr*	→	*Expr + Term*
3		|	*Expr - Term*
4		|	*Term*
5	*Term*	→	*Term * Factor*
6		|	*Term / Factor*
7		|	*Factor*
8	*Factor*	→	*(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

18

اللي يخلي مشاكل لازم احولها لـ right

Left

$A \to A\alpha \to A\alpha\alpha \ldots$

$\beta\alpha\alpha$

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:

$$\boxed{\begin{array}{lcl} A & \to & A\,\alpha \\ & \mid & \beta \end{array}}$$

كرر الفا قدما بدل

وانظر لـ β

left recursive

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

$$\left.\begin{array}{lcl} A & \to & \beta\,\hat{A} \\ \hat{A} & \to & \alpha\,\hat{A} \\ & \mid & \varepsilon \end{array}\right\} \boxed{\text{The new grammar is right-recursive}}$$

$\beta\alpha^*$

 19

كيف اكتبه بطريقة ثانية

Let us Generalize

A general left-recursive grammar

$$A \to A\,\alpha_1 \mid A\,\alpha_2 \mid \ldots \mid A\,\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

The equivalent right-recursive grammar

$$\begin{array}{lcl} A & \to & \beta_1\hat{A} \mid \beta_2\hat{A} \mid \ldots \mid \beta_m\hat{A} \\ \hat{A} & \to & \alpha_1\hat{A} \mid \alpha_2\hat{A} \mid \ldots \mid \alpha_n\hat{A} \mid \varepsilon \end{array}$$

 20

Eliminating Left Recursion in the Expression Grammar

The expression grammar contain the following left recursion cases

$\hat{A}$

Expr → *Expr* + *Term*
| *Expr* - *Term*
| *Term*

$\hat{A}$ *Term* → *Term* * *Factor*
| *Term* / *Factor*
| *Factor* β

Applying the transformation yields

Expr → *Term* $\hat{E}$
$\hat{E}$ → + *Term* $\hat{E}$
| - *Term* $\hat{E}$
| ε

Term → *Factor* $\hat{T}$
$\hat{T}$ → * *Factor* $\hat{T}$
| / *Factor* $\hat{T}$
| ε

top down parsing لل اكل ,ه

The Right-Recursive Algebraic Expression Grammar

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

 23

اتفاق

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;
}
```

 24

* pointer → next
  next token to be consumed
* function → Match
  بقارن بين t و next
* t → token

هاد الكود اللي بدي استخدمه بالـ code الكبير

non terminal ← نادي (S)

Top-Down Parser Code For A Simple Grammar

terminal → يوقف

non terminal → expand يعمل

Rule *S*

```
bool S ( ) { return B( ) && C( ) && A( ) ;}
```

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

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
}
```

Only tries rule A2 if A1 fails

Backtrack by reversing pointer

25

داخل " true يرجع لأنها ع

نحفظ ال next في save temp.

لو هاي اجت true خلص ما يكمل السطر الثاني

- واذا ما تحقق الشرط يروح عالسطر الثاني

- واذا ما تحقق A1 و A2 بعطي false ويرجع يعمل 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
}
```

نادي terminal بيستخدم match

26

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 *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
}
```


Exercise H.W

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

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

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

Input Grammar by user → ANTLR → Java codes for a scanner and a parser

lexer.Java
parser.Java

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

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 ☺
  - The topic of our next lecture

 31

عشان
كل ال
performance

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)

 32

** كون الامتحان
الفيرست

Exercise

$S \longrightarrow (A) \times (B)$

$A \longrightarrow T \text{ num}$

$T \longrightarrow T \underbrace{\text{num} +}_{\alpha}$
$\quad | \underbrace{\varepsilon}_{\beta}$

$\Longrightarrow$

$$\left\{ \begin{array}{l} T \longrightarrow \hat{T} \\ \hat{T} \longrightarrow \text{num} + \hat{T} \\ \quad | \ \varepsilon \end{array} \right\} \Longrightarrow \begin{array}{l} T \longrightarrow \text{num} + T \\ \quad | \ \varepsilon \end{array}$$

$B \longrightarrow B + \text{num}$
$\quad | \text{num}$

$\Longrightarrow \{ \qquad \}$

lec 6

1/25/19

Predictive Parsing
Spring 2018/2019

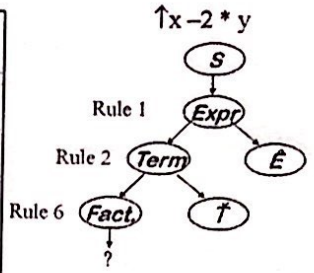
Instructor: Dr. Fahed Jubair
Computer Engineering Department
University of Jordan

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 role 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

Key Idea: Looking Ahead Helps

1	*S*	→	*Expr*
2	*Expr*	→	*Term Ê*
3	*Ê*	→	*+ Term Ê*
4		|	*- Term Ê*
5		|	ε
6	*Term*	→	*Factor Ť*
7	*Ť*	→	** Factor Ť*
8		|	*/ Factor Ť*
9		|	ε
10	*Factor*	→	*(Expr)*
11		|	number
12		|	id



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

3

كل ال Rule بتو grammar تلبس باسي مختلف

Back free grammer

Backtrack-Free Parsers

- Given A → α | β, 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")

4

اذا في حل بوصل ل parsetree

واذا ماني حل رح ارفع ل syntax error

وما رح يسوي Back-track

grammar
منها جزء أول Rule لـ
مختلف

بتستخدم 1 look ahead character
عشان يميز بين Rule

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

في
LL2
LL3
⋮

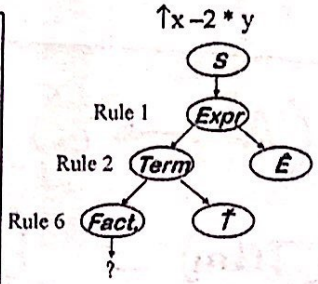
Predictive Top-Down Parsing

- Consider a top-down parser that uses an LL(1) grammar
- Let the next nonterminal node to be expanded by a the parser be A
- Let the lookahead token be $\boxed{t}$
- 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

$A \rightarrow t\alpha_1 \rightarrow$ هاي بتبلش بـ t
$\mid \alpha_2$
$\mid \alpha_3$ ما بيروح Rule
⋮ تانية تبلش بـ t

LL(1) Grammar Example

1	*S*	→	*Expr*
2	*Expr*	→	*Term Ê*
3	*Ê*	→	*+ Term Ê*
4		|	*- Term Ê*
5		|	ε
6	*Term*	→	*Factor Ť*
7	*Ť*	→	** Factor Ť*
8		|	*/ Factor Ť*
9		|	ε
10	*Factor*	→	*(Expr)*
11		|	number
12		|	id



Lookahead token is *x*
Factor → *id* is the **only** rule that derives *x* in the first position

 7

Prediction Criteria

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

Let us define FIRST and FOLLOW sets

 8

first set
↓
token عم
و
Rule
تبلش فيها

FIRST Sets

- Let $A \rightarrow \alpha$ be a production rule in an LL(1) grammar
- *FIRST*(α) is the set of all **terminals** that appear as the first token in some string that derives from α

  FIRST(α) = { all terminal tokens t such that $\alpha \Rightarrow^* t\,\beta$ }
- *FIRST* sets have the following properties:

  1. *FIRST* (t) = { t }, where t is a terminal
  2. $\varepsilon \in$ *FIRST*(α) if any of the following holds:
     - $\alpha \rightarrow \varepsilon$, OR
     - $\alpha \rightarrow X_1 X_2 \ldots X_n$ and $X_i \rightarrow \varepsilon$ for all i: $1 \le i \le n$
  3. *FIRST*(α) $\subseteq$ *FIRST*(β)

     if $\beta \rightarrow X_1 X_2 \ldots X_n\, \alpha$ and $X_i \rightarrow \varepsilon$ for all i: $1 \le i \le n$

اذا اخر بروح
عالع بعدها روح
عالي بعدها

Example 1

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

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

* First (factor → (expr)) = { (}
  Rule 10

* " (rule 11) = { number }
  " (" 12) = { id. }

→ factor

5

first(B) C first(S)

Example 2

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

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

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

 11

first (S) = { € , £ , ~~ε~~ }

اذا اختار ε

فانا بروح على C

⇒ { € , £ , num, id }

FOLLOW Sets

ممنوع تكون ال ε موجودة بال set

- 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*) = {all terminals *t* such that $S \Rightarrow^* \alpha A t \beta$ }
- *FOLLOW* sets have the following properties:
  1. $ ∈ *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* (β) ⊆ *FOLLOW* (α)
  4. If $A \rightarrow \alpha\beta$, then *FOLLOW* (*A*) ⊆ *FOLLOW* (β)
  5. If $A \rightarrow \alpha\beta$ and $\beta \Rightarrow^* \varepsilon$, then *FOLLOW*(A) ⊆ *FOLLOW*(α)

 12

اي اشي بيجي ورا S بيجي ورا expr

Example 1

FOLLOW (S) ⊆ FOLLOW ($Expr$)
FOLLOW ($Expr$) ⊆ FOLLOW ($\hat{E}$)
FOLLOW ($Expr$) ⊆ FOLLOW ($Term$)
FIRST($\hat{E}$) ⊆ FOLLOW ($Term$)
FOLLOW($\hat{E}$) ⊆ FOLLOW ($Term$)
FIRST($Term$) ⊆ FOLLOW (+)
FIRST($Term$) ⊆ FOLLOW (-)
FOLLOW ($Term$) ⊆ FOLLOW ($\check{T}$)
FOLLOW ($Term$) ⊆ FOLLOW ($Factor$)
FIRST($\check{T}$) ⊆ FOLLOW ($Factor$)
FOLLOW($\check{T}$) ⊆ FOLLOW ($Factor$)
FIRST($Factor$) ⊆ FOLLOW (*)
FIRST($Factor$) ⊆ FOLLOW (/)
FIRST($Expr$) ⊆ 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

كيف اسوي follow

بشوف مين في وراي

بعدين الي بيجي

وراي بكون جزء منه

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

ووراه Term بشوف

↳ $\hat{E}$

بالتالي follow(term) = {+ و - و (و $}

① $A \longrightarrow \alpha\beta$

$follow(A) \subseteq follow(\beta)$

---

② $A \longrightarrow \alpha\beta$

$first(\beta) \subseteq follow(\alpha)$

* في هدا بيجي ورا ي

* في هدا ~~[illegible]~~ بروح على

$Follow(\hat{T}) = \{\$,), +, -\}$ لانه نفس term

$follow(factor) = \{*, /, \$,), +, -\}$ +ي بيلزق في $\hat{T}$ لانه جاي ورا factor

---

$\hat{T}$ بتروح على ع

فبتروح term

Example 2

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

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

FOLLOW (*S*) = FOLLOW (*A*) (why?)
FOLLOW (*S*) ⊆ FOLLOW (*C*)
FIRST (*C*) ⊆ FOLLOW (*B*)
FIRST (*A*) ⊆ FOLLOW (*C*)

FOLLOW (*S*) = { $ }
FOLLOW (*A*) = { $ }
FOLLOW (*B*) = { number, id }
FOLLOW (*C*) = { $, €, £, number, id }
FOLLOW (€) = FOLLOW (£) = { number, id }
FOLLOW (number) = FOLLOW (id) = { $, €, £, number, id }

 15

first & follow

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*(α) as the set of all terminals that appear as the first token in some string that derives from α
- PREDICT(α) is computed as follows:

Predict (α) = FIRST (α) – { ε }
if (ε ∈ FIRST (α))
    Predict (α) = Predict (α) U FOLLOW(α)

 16

parse table

	£	E	num	id	$
S	1	1	1	1	error
A	2	2	2	2	3
B	4	5	6	6	error
C	error	error	7	8	error

1/

Example 1

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

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

Example 2

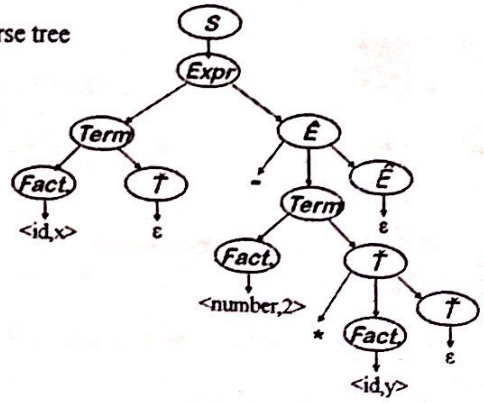
- Terminals are €, £, number, id
- Nonterminals are *A*, *B*, *C*, *S*

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

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

Let us Redo S ⇒* x – 2 * y

Final parse tree



next Chara. to be consume

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

فنكشن بكلي بطلع
على اول Token في الـ input
بـ match بعمل أو false

A Non-LL(1) Grammar Example

1	*Function*	→	*id*
2		|	*id (ArgList)*
3		|	*id [ArgList]*
4	*ArgList*	→	id MoreArgs
5	*MoreArgs*	→	, id MoreArgs
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?

Left Factoring

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

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \alpha\beta_n \mid \gamma$$

- 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 $\hat{A}$:

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

$$\hat{A} \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$$

اخذ α
عامل مشترك
وتكون
$\hat{A}$
هي
$(\beta_1 \mid \beta_2 \ldots)$

Left Factoring Example

1	*Function*	→	*id*
2		|	*id (ArgList)*
3		|	*id [ArgList]*
4	*ArgList*	→	id MoreArgs
5	*MoreArgs*	→	, id MoreArgs
6		|	ε

Non-LL(1) Grammar

1	*Function*	→	*id X*
2	*X*	→	*(ArgList)*
3		|	*[ArgList]*
4		|	ε
5	*ArgList*	→	id MoreArgs
6	*MoreArgs*	→	, id MoreArgs
7		|	ε

LL(1) Grammar

 35

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

more powerful

1 char

 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

 37

Exercise

1	S	→	A B $
2	A	→	x B
3		|	x w
4	B	→	x y A
5		|	z

i. The above grammar is not LL(1). Explain why.

ii. Use left factoring to rewrite the grammar into an LL(1) grammar

iii. Show the LL(1) parsing table for your grammar in part ii

iv. Using the parsing table, show the derivation steps for the string xxyxwz$

 38

A يتزوج على x مرتين

يحوله بس اخذ x عامل مشترك

Exercise

- Construct the LL(1) parsing table for the shown LL(1) grammar
- Using the parsing table, show the derivation steps for the string
  $id - id((id))$

1	E	$\rightarrow$	$-E$
2		$\mid$	(E)
3		$\mid$	$V\,Etail$
4	V	$\rightarrow$	$id\,Vtail$
5	$Vtail$	$\rightarrow$	(E)
6		$\mid$	ε
7	$Etail$	$\rightarrow$	$-E$
8		$\mid$	ε

39

	(	)	id	−	$
E					
V					
Vtail					
Etail					

exercise

	x	w	y	z	$
S					
A					
$\hat{A}$					
B					

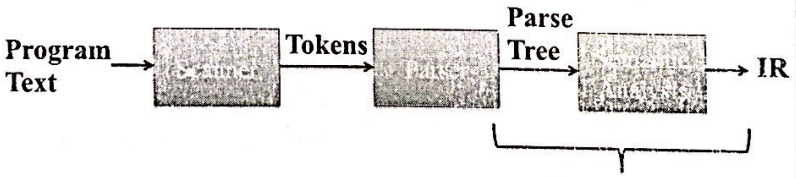
لازم
اضيف
الاستي
الجديد

Lec 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)

3 basics

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 ;
}
```

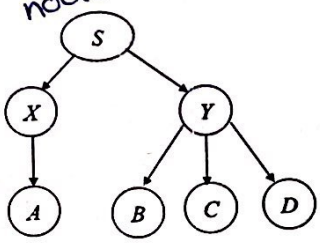

Semantic Actions

- 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

The Visitor Pattern

يمر على كل ال node

- 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);
}
```

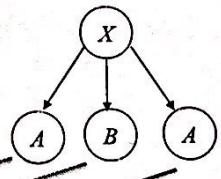
childes of S

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

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


Parse Tress 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



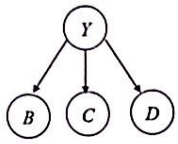

```
Visit X() {
    visit(A[0]);
    visit(B);
    visit(A[1]);
}
```

if we have 2 A

parser ال بفرقهم

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

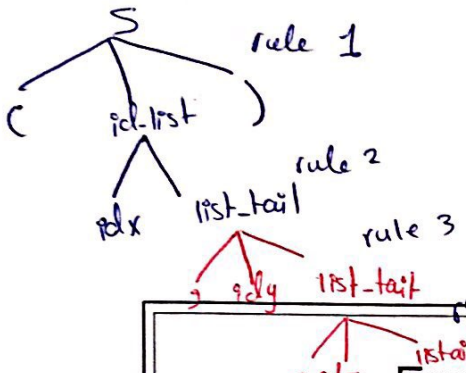
Example 1

- 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

we have 4

Rule → 4 visits



Example 1 (cont.)

The Visitor pattern: default routines

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

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

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

Visit Rule4 ( ) {
    return ;
}
```

list_tail has two rules
Which one is invoked ?

ε

@ All Rights Reserved.

بناديها باسمها
ال non terminal
مش باسم ال Rule2

هون
بس بنادي ال
list_tail
او هي لانه في عندي
الها Rule 3,4

ما بزبطها احطهم التين
فا حسن اناد ي بال
non - terminal.

Example 1 (cont.)

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

int count; // global variable accessed everywhere

```
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.

بعدال id

visit (،)
visit (id_list)
visit (،)

بدل
visit id

او كمط visit(id)
و عند visit(id)
count++ كمط.

بتقلل وقت

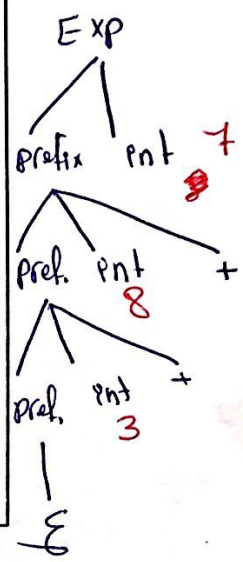
* semantic attribute → count of id.

Dummy example

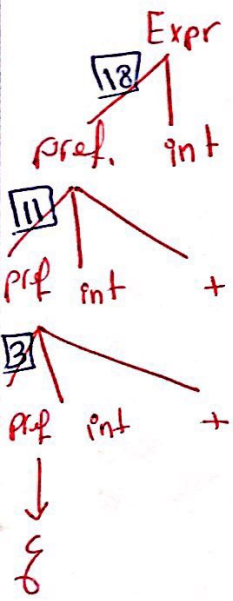
Example 2

- 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* → terminal
2	*Expr_prefix*	→	*Expr_prefix INT +*
3		|	ε



ما في attributes بقدر اجمعها إلا اذا عدلت الكود



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 ;
}
```

token من نوع int

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

الكود الاصلي :

```
visit Rule 1 ()
{ visit (pref.)
  visit (INT)
}

visit Rule 2()
{ visit (pref)
  visit (int)
  visit (+)
}

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

bottom up

أفضل بالمشروع

لأنه ما في global variable

حل

Top down

```
Visit Rule 1()
{ sum = getvalue(int)
  Visit (prefix)
}
```

→ اعمل visit لل int اول

```
Visit Rule 2()
{ sum = sum + getvalue(INT)
  visit (prefix)
}

visit rule3()
{ return;
}
```

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*

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 Tab
  - Second Pass: performing type checking
  - Third Pass: generating the IR

step 3 in project

step 4 in project

table
تجمع معلومات
مشان تستخدمها
ال 2nd
بس pass
check تسوي

First Pass: Building The Symbol Table

- A data structure that the compiler uses to store information about declared identifiers
- Information in symbol tables are typically obtained by *processing declarations*
  - Function declarations
  - Variable (scalars, arrays, pointers, etc) declarations
  - Class declarations
- Building symbol tables is typically the first pass in the semantic action passes

Scope in Java

- Each identifier in the symbol table corresponds to a *scope*
- An identifier's scope is the portion of the program where this identifier is **visible**
- *Global variables*: variables that are visible to all scopes
- *Local variables*: variables that are declared by a particular scope, i.e, only visible within this scope
- Note that scopes can be nested within each other
- Are variables visible across nested scopes? The answer depends on the programming language specification

A C Code Example

```
#include "stdio.h"

int x, y;

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

void main (){

  int x, w ;
  ...
  {
    int z;
    ...
  }
  ...
}
```

Global Scope

ID Name	ID Type
x	int
y	int

هاد ال table
Global scopes

A C Code Example

```
#include "stdio.h"

int x, y;

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

void main (){

  int x, w ;
  ...
  {
    int z;
    ...
  }
  ...
}
```

foo Scope

ID Name	ID Type
n	int
m	string
s	float

& the table of globale scope also available for foo

In C, variables in the *global* scope are also visible to *foo* scope

A C Code Example

```
#include "stdio.h"

int x, y;

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

void main (){

  int x, w ;
  ...
  {
    int z;
    ...
  }
  ...
}
```

main Scope

ID Name	ID Type
x	int
w	int

A C Code Example

```
#include "stdio.h"

int x, y;

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

void main (){

  int x, w ;
  ...
  {
    int z;
    ...
  }
  ...
}
```

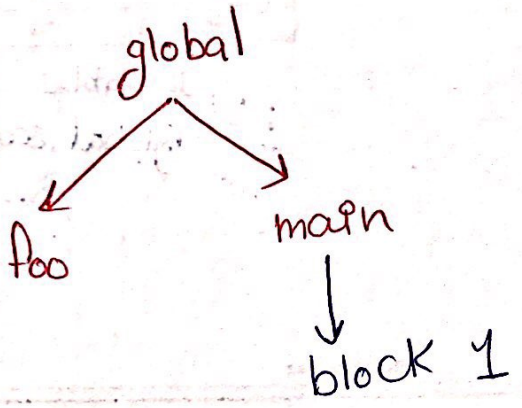
inner
بشوف outer
بس العكس ✗

Block 1

ID Name	ID Type
z	int

Note that variables in *global* scope and *main* scope are also visible to *Block1* scope (according to C specification)

* ممكن أسوي table وأحط في IDName , type , لأي Scope تابع



* الـchild بقدر يشوف *
الـparents

foo & main
ما بقدرو يشوفو بعض

* لقد اسوي symbol table لكل من

classes / methods / Identifier - - -

A C Code Example

```
#include "stdio.h"

int x, y;

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

void main (){

  int x, w ;
  ...
  {
    int z;
    ...
  }
  ...
}
```

Special symbol table for storing function information

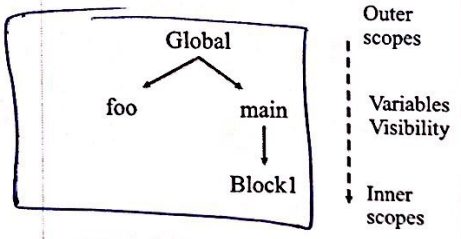
Function Name	Function Return Type	Function Arguments
foo	float	int, string
main	void	--

للـ func. التي عندي

A C Code Example

```
#include "stdio.h"

int x, y;

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

void main (){

  int x, w ;
  ...
  {
    int z;
    ...
  }
  ...
}
```

Relationship between scopes



Symbol Table Organization

- An individual table for each scope
  - Advantage: simpler design and more memory-efficient
  - Disadvantage: may need to search in multiple tables
  - ~~We will use this organization for our course project~~
- Or alternatively, one symbol table for all scopes
  - Advantage: faster search time
  - Disadvantage: more complex to design and manage

رح نستخدم هاد الديزاين بالمشروع انه كل scope لحالها جدولها

* كيف أبني symbol table إلحالها في أكثر من طريقة :- 1- array
2- hash map

Symbol Table Implementation

1. Linked List
   - Straightforward implementation
   - Impractical for big programs (due to O(n) search time) → worst case
2. Binary Search Tree
   - More practical for large programs: O(log n) search and insertion time for the average case
   - Can still be slow with the worse case scenario: O(n) time
3. Hash Table
   - Best solution
   - Most common implementation

Antlr → Visitor.java بيعطيني

أنا بعمل myvisitor.java ← بعمل عليه overwrite

و ST.java ← بيجي عليه ال interface

Symbol Table Interface

OpenScope ( )	Opens a new scope in the symbol table. New symbols are entered in the resulting scope.
CloseScope ( )	Closes the most recently opened scope in the symbol table. Symbol references subsequently revert to outer scopes.
EnterSymbol (Name *s*, Type *t*)	Insert variable *s* whose data type is *t* in the symbol table's current scope.
RetrieveSymbol (Name s)	Returns the symbol table's currently valid information for variable *s*. If no such entry exists, then a null pointer is returned.
DeclaredLocally (Name *s*)	Tests whether variable *s* is present in the symbol table's current scope.

- This is one way of designing the interface of symbol tables
- Other ways to design symbol tables are also possible

Implementation Example

- Use the visitor pattern to write semantic routines that insert declared variables in the symbol table while traversing the parse tree of the following grammar

1	*var_decl*	→	*var_type id_list ;*
2	*var_type*	→	INT
3		|	FLOAT
4	*id_list*	→	*id* id_tail
5	*id_tail*	→	, *id* id_tail
6			ε

* every rule has a visit

* مش كلهم بعالجهم visite

الا ما فيهم declarations

ST=new ST();

ST.enterSymbol(var, p);

ST.Java

ST.Java

x	int
y	int
z	int

2/21/19

bottom up sd.

Implementation Example

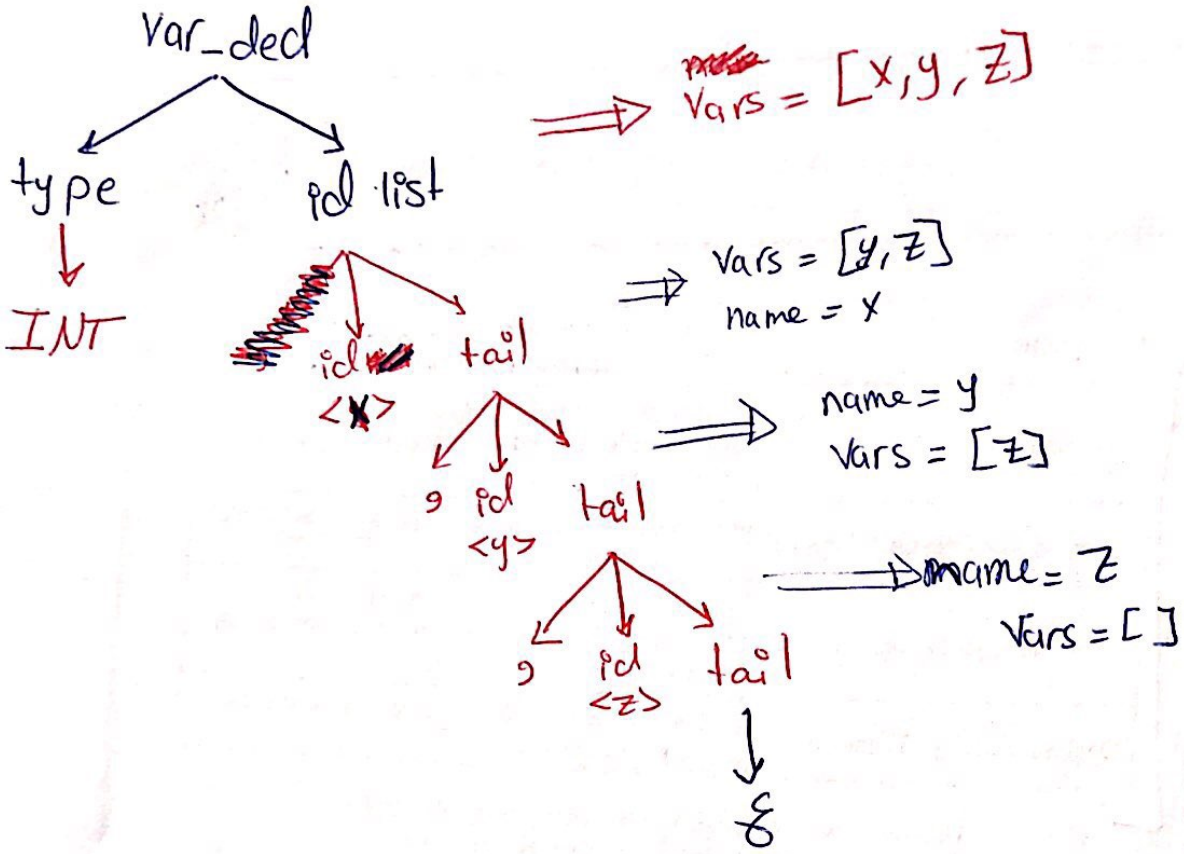
```
Visit Rule1 ( ) {
  Type p = visit ( var_type ) ;
  String [] vars = visit ( id_list ) ;
  foreach ( String var : vars )
      EnterSymbol ( var , p );
}

Visit Rule2 ( ) {
   return Type.INT ;
}

Visit Rule3 ( ) {
   return Type.FLOAT ;
}

Visit Rule4 ( ) {
  String var = getTokenName(id);
  String [] vars = visit ( id_tail ) ;
  vars.add(var) ;
  return vars ;
}

Visit Rule5 ( ) {
   // same code as Rule4
}

Visit Rule6 ( ) {
   return [ ];      // empty array
}
```


Second Pass: Performing Type Checking

- The purpose of this pass is to check for semantic errors
- Type checking implementation depends on the programming language type:
  - *Strong typed* programming languages: languages that require the arguments of an operation to be of a type that is consistent with what the operation is defined for
    - E.g., Java
  - *Weakly typed* programming languages: languages that requires no guarantees that operations are performed with arguments that make sense for the operation
    - E.g., Assembly

* مثلاً

int x, y, z;



---

* بالمشروع بركز على ال grammers
اللي مسؤولين عن declarations

* بال string

str	string value

Symantec Routines For Type Checking

id	Symbol s = RetrieveSymbol(*id*) if *s* == *null* then ***semantic_error***
Expr + Expr	Type t1 = visit (Expr[0]) Type t2 = visit (Expr[1]) if t1 ≠ t2 then ***semantic_error***
id = Expr	Type t1 = visit (id) Type t2 = visit (Expr) if t1 ≠ t2 then ***semantic_error***

يشوف نوع الـ id

اذا both expr لا يوجد نفس type

Symantec Routines For Type Checking

IF cond THEN stmt_list ELSE stmt_list	Type t = visit (cond) if t ≠ Type.*BOOLEAN* then ***semantic_error*** visit (stmt_list[0]) visit (stmt_list[1])

WHILE (cond)
stmt_list ;
ENDWHILE
	Type t = visit (cond)
	if t ≠ Type.*BOOLEAN* then ***semantic_error***
	visit (stmt_list)

FOR (id = Expr ; cond ; id = Expr)
stmt_list ;
ENDFOR
	?

cond ↓ لازم يكون boolean

Type Checking is Not Trivial

- Due to the fundamental limitation of compilers, some type checking requires complex compiler analyses
- In come cases, *dynamic* type checking is more efficient
- In general, type checking can be classified as
  - *Static*: done at compile-time
    - No runtime overhead but limited to compile-time knowledge
  - *Dynamic*: done at runtime
    - No knowledge limitation but runtime overhead occurs
  - *Hybrid*: only use dynamic analysis for the portion that requires runtime knowledge, otherwise, use static analysis

no full run time knowledge

random desicion

Java

Implementation Example

- Use the visitor pattern to write semantic routines that check if operands have consistent types while traversing the parse tree of the following expression grammar

subset of grammar expr

1	*Expr*	→	*Term Ê*
2	*Ê*	→	+ *Term Ê*
3		|	ε
4	*Term*	→	INTLITERAL
5		|	FLOATLITERAL
6		|	id

رقم صحيح

رقم كسري

x, y . .

if (t2 == null) → ε
return true
if (t1 != t2)
return false
else return true

Implementation Example

```
Visit Rule1 ( ) {
    Type t1 = visit ( Term ) ;
    Type t2 = visit (Ê) ;
    if (t2 != null & t1 != t2)
        return fail;
    else
        return success ;
}
Visit Rule2 ( ) {
    Type t1 = visit ( Term ) ;
    Type t2 = visit (Ê) ;
    if (t2 != null & t1 != t2)
        return fail;
    else
        return t1 ;
}
```

semantic error

```
Visit Rule3 ( ) {
    return null ;
}
Visit Rule4 ( ) {
    return Type.INT ;
}
Visit Rule5 ( ) {
    return Type.FLOAT ;
}
Visit Rule6 ( ) {
    return RetrieveSymbol(id).type() ;
}
```

ε

visit (+)
visit (term)
visit (Ê)
if (t2 == null)
return t1
if (t1 == t2)
return t1
else
return error;

Third Pass: Generating The IR

- *The Intermediate Representation* (*IR*): a data structure that encodes the compiler's knowledge about the input program
- The IR is expected to be
  - Expressive, i.e., contains sufficient information about the input program
  - Amenable to performing optimization → تكون سهلة اسويلها optim.
  - Easy to translate to machine code

*preserve meaning !!

it has lots of info. about input program

* Some compilers take the parse tree as a IR → if the tree representation is fine & easy

كل ما كانت الـ IR بتشبه الـ assembly machine وسهلة بقدر احولها

*high level → human lang.
*low level → assembly lang.

IR Properties

- An IR has three basic properties:

1. Structure: can be a graph, linked list, etc
   - Affects ease of generation → low level
   - Affects ease of manipulation → high level
2. Level of abstraction: vary from near-source representation to low-level representation
   - Affects optimization applicability
   - Affects ease of translation into machine code
3. Naming scheme: how values are represented in the IR
   - Affects optimization applicability

IR Classification

attributes of IR

- Graphical IRs (graph) — high level
  - Graphical representation ex:- tree
  - Usually near-source level of abstraction
  - Common example: parse trees, control flow graphs
- Linear IRs (list) — low level
  - Pseudo-code for an abstract machine
  - Usually low-level abstraction
  - Common examples: 3-address code, stack machine code
- Hybrid IRs
  - Combination of graphs and linear code both attributes

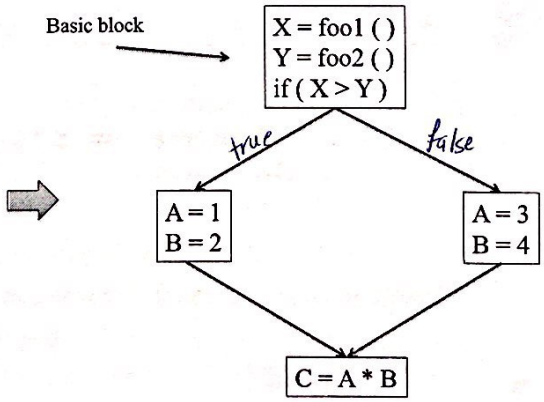
directed graph

Graphical IR Example: Control Flow Graph (CFG)

- A graph that models the transfer of control in the program
- Each node in the graph corresponds to a *basic block*
  - A basic block is a maximal-length of sequential code that is free of labels (except at the beginning) and branches (except at the end)
- Each edge in the graph corresponds to a possible control transfer between basic blocks in the program
- See the example in the next slide

CFG Example

```
X = foo1 ()
Y = foo2 ()
if ( X > Y )
    A = 1
    B = 2
else
    A = 3
    B = 4
endif
C = A * B
```



هدول بتنفذو ورا بعض

هون بصير تفرع

مر/ يعطيني كود ويطلب الـ graph

minimal number of block graph بكل

block جوا sequential statements بحط اكبر عدد من

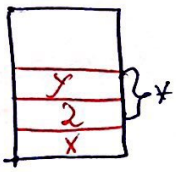
* كتير اوصل لجملة في الها خيارين بصير بدي اروح لبلوك جديد

assembly like

Linear IR Example: Stack-Machine Code

- Designed for an abstract stack-based machine
  - A stack is used for implementing registers
  - Operations pop their operands from the top stack entries, and then push their result back to the stack top entry
- Example

```
                push X
                push 2
X – 2 * Y  ⇒    push Y
                multiply
                subtract
```



اول ٢ بالستاك

- Stack-machine code greatest merit is its compactness
- Bytecode is a common stack-machine code example

EX

stack machine code (push, pop) there is No registers

*one address
↳ one operand

*two address
↳ 2 operands

stack machin is popular?
small & simple to code

Linear IR Example: Three-Address Code

- In general, 3-address code has the form:

  opcode, *operand1*, *operand2*, *result*

- Example

```
                muliply 2, Y, $T1
Z = X – 2 * Y ⇒ subtract X, $T1, $T2
                store   $T2, Z
```

- Properties:
  - Resembles assembly code
  - Introduces a new set of names (i.e., temporaries)
  - Weakly typed
  - Reasonably compact

EX

Project

parse tree → step 3 → ST, parse tree → step 4 → 3 address code

لازم يكون عندي خطوة تشيك على semantic errors

IR Code Generation

- While traversing the parse tree, **semantic actions** process each node and generate the IR data structure
- In the course project, we will use the three-address code as the intermediate representation
- Therefore, I will discuss code generation for three-address code in the next couple of slides
- However, the same concepts can be applied if a different IR is used

Three-Address Code Expressions

→ IR

Micro Code	Three-Address Code
a = 1 ;	STORE 1 $T1 STORE $T1 a
b = 0 ;	STORE 0 $T2 STORE $T2 b
a = a + b ;	ADD a b $T3 STORE $T3 a
b := 1 / a - b * 2 ;	DIV 1 a $T4 MULT b 2 $T5 SUB $T4 $T5 $T6 STORE $T6 b

store 1, a

ما في مشكلة - دايركت

sort

better

استخدم temp

Three-Address Code
IF-ELSE Control

greater or equal

Micro Code	Three-Address Code

IF (a < b)
c := b – a ;
ENDIF
	GE a b L1
	SUBI b a $T1
	STOREI $T1 c
	LABEL L1

IF (a < b)
c := b – a ;
ELSE
c := a – b ;
ENDIF
	GE a b L1
	SUBI b a $T1
	STOREI $T1 c
	JUMP L2
	LABEL L1
	SUBI a b $T2
	STOREI $T2 c
	LABEL L2

بفحص على الـ operation

إذا تحقق بنفذ

else

else

Three-Address Code
WHILE Loops

Micro Code	Three-Address Code

WHILE (i != 10)
READ (p) ;
i := i + p ;
ENDWHILE
	LABEL L1
	EQ i 10 L2
	READI p
	ADDI i p $T1
	STOREI $T1 i
	JUMP L1
	LABEL L2

cond. على

* بمشوها هنا for

Three-Address Code FOR Loops

Micro Code	Three-Address Code
	STOREI 0 i
	LABEL L1
	GE i n L2
FOR (i := 0; i < n ; i := i + 1)	MUL t i $T1
t = t * i;	STORE $T1 t
ENDFOR	~~MUL~~ i 1 $T2
	STORE $T2 i
	JUMP L1
	LABEL L2

lable for entry

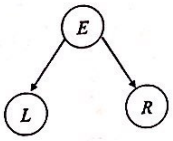
ADD

lable for exit

*STEP 4

Three-Address Code Semantic Action Pass

- Do a post-order walk of the parse tree
- A node generates code for its children before generating code for itself




```
Visit E ( ) {
   data_object lcode = visit ( L ) ;
   data_object rcode = visit ( R ) ;
   emit_code (lcode , rcode ) ;
}
```

بتضيف كود على object تاعنا

- Data objects can contain code or any other information

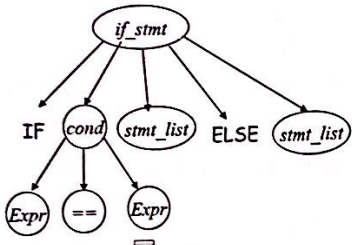
step4.visitor.Java
3
بدي اضاف فيها
ST

IR.stmt.Java
opcode
operands
result
2

IR.Java
List <IR stmt.>
add (IRstmt)
emit
1

١- بعدل على grammers
٢- نعمل rerun antlr
٣- بنجيب new visitor
٤- بعمل كلاس IR.Java

Example: IF-ELSE Control



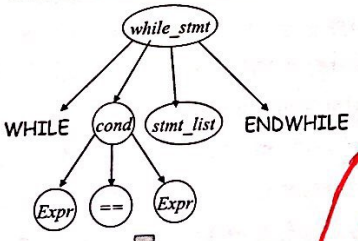

```
neq Expr1, Expr2, else_path
... stmt_list1 code ...
jump exit_path
label else_path
... stmt_list2 code ...
label exit_path
```

```
Visit if_stmt ( ) {
    exit_path = new Label ( ) ;
    else_path = new Label ( ) ;
    visit ( cond ) ;
    visit ( stmt_list[0] ) ;
    emit_code ( jump, exit_path ) ;
    emit_code ( label, else_path ) ;
    visit ( stmt_list[1] ) ;
    emit_code ( label, exit_path ) ;
}

Visit cond ( ) {
    Expression e1 = visit ( Expr1 ) ;
    Expression e2 = visit ( Expr2) ;
    emit_code (neq, e1, e2, else_path);
}
```


condition

اول واحد بيول visit

Example: WHILE Loop




```
label loop_entry
neq Expr1, Expr2, loop_exit
... stmt_list code ...
jump loop_entry
label loop_exit
```

```
Visit while_stmt ( ) {
    loop_entry = new Label ( ) ;
    loop_exit = new Label ( ) ;
    emit_code ( label, loop_entry ) ;
    visit ( cond ) ;
    visit ( stmt_list ) ;
    emit_code ( jump, loop_entry ) ;
    emit_code ( label, loop_exit ) ;
}

Visit cond ( ) {
    Expression op1 = visit ( Expr1 ) ;
    Expression op2 = visit ( Expr2) ;
    emit_code ( neq, op1, op2, loop_exit );
}
```


label entery

exit

Additional Code Structures

- Refer to the text books to learn more about IR code generation for
  - Switch and Case statements
  - Arrays
  - Functions
  - Classes
  - etc

optional read

Summary

- Semantic actions are routines that the compiler invokes while traversing the parse tree nodes to:
  - Generate the symbol table
  - Perform type checks
  - Generate the intermediate representation
- Semantic actions discover the semantic errors in the program being compiled and report them to the user
- However, semantic actions cannot always discover all semantic errors in the program
  - It is the user responsibility to find any "uncaught" errors by the compiler

Exercise 1

- The below grammar describe strings of integers that are added to each other. Use the visitor pattern to write semantic routines that determine the sum while traversing the parse tree of the below grammar.
  - For example, the sum computed by traversing the parse tree of the string 11+4+40+0 is 55
  - Assume *GetValue(Token t)* is a function that returns the integer value of *t*, where *t* is a token with the type *INT*
  - You may declare any global variables you want, if necessary.

1	*Expr*	→	*INT Expr_tail*
2	*Expr_tail*	→	+ *INT Expr_tail*
3		|	ε

* دائماً نرسم أول شي parse tree مجان كذا bottom up or top down

Exercise 1 Solution

```
// global variables declaration
int sum ;

Visit Rule1 ( ) {
   sum = GetValue (INT) ;
   return visit (Expr_tail) ;
}

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

Visit Rule3 ( ) {
   return 0;
}
```

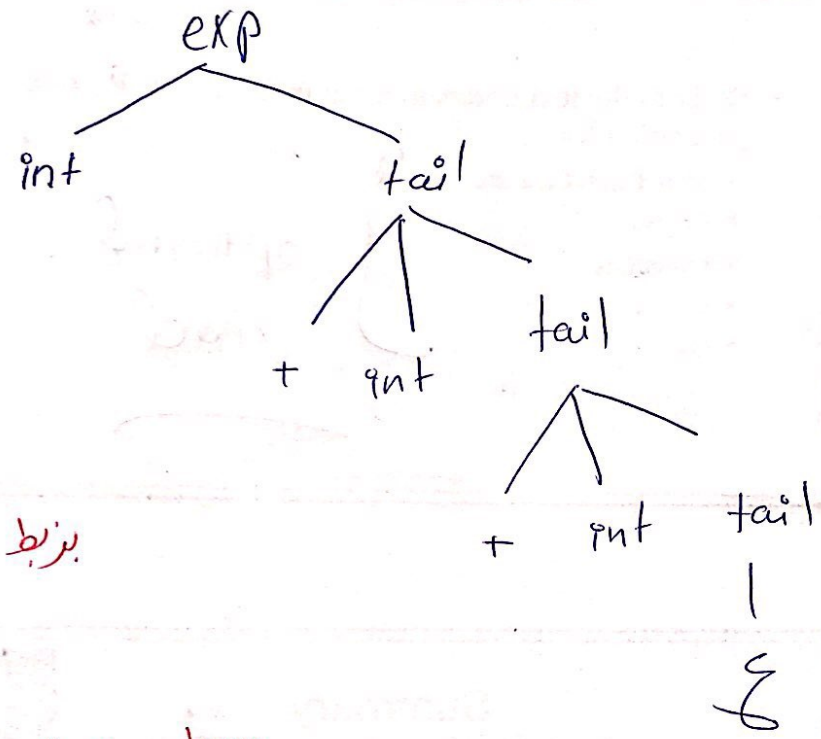
recursively

بتكتبوها وحدها
sum = int

visit (int)
visit (tail)

parse tree

11 + 4 + 40 + 0



بزبط ، طلع التيتن

top down

Exercise 2

- The below grammar describes all strings in the regular expression (aa | bab)$^+$. Use the visitor pattern to write semantic routines that counts how many time *a* has occurred while traversing the parse tree of the below grammar.
  - For example, *a* occurred 5 times in the string *aababbabbab*
  - You may declare any global variables you want, if necessary.

1	*E*	→	*aa E*
2		|	bab E
3		|	aa
4		|	bab

Top down

Exercise 3

- Use the visitor pattern to write semantic routines that counts how many floats have been declared while traversing the parse tree of the below grammar.
  - For example, 3 floats were declared for the string *INT a,b; FLOAT b; FLOAT c,d; INT e,f,g;*
  - You may declare any global variables you want, if necessary.

1	*V*	→	INT *id_list* ; *V*
2		|	FLOAT *id_list* ; *V*
3		|	ε
4	*id_list*	→	*id* id_tail
5	*id_tail*	→	, *id* id_tail
6		|	ε

declaration grammar

INT ما بزبط اعمل sum لأن رح يسير بعد

الحل انه مش بس ازود الـ id-list اضيف

بضيف لي الـ sum بس أكون ب float

* Compiler

Lec 8

Back end compiler (optimizer) ⟶ بعدل على code اللي كتبه ال human

* optimization
  ↳ من خلال مجموعة من ال passes

Ex

~~x = 1;~~
y = x + 2;

اذا ما في استخدام ل x مرة ثانية بشطبها من كل البرنامج

y = 3

~~x = 5;~~
if (5 x > 0)
else

هاد العبارة بلغي X لانه ما حاجي عليه

---

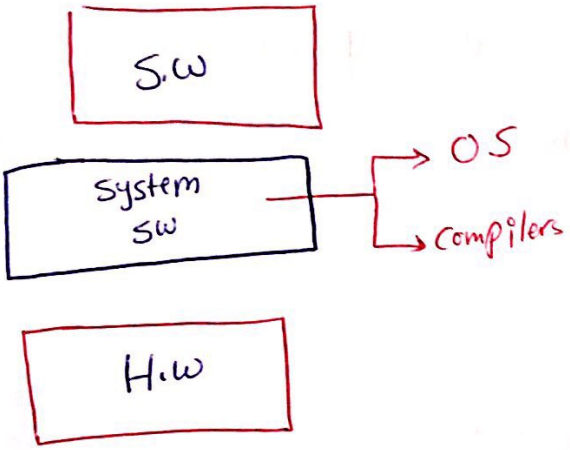
* improvements
- execution time أقل
- power / energy
- memory footprint
- high parallelism

* multiplication
more complex & energy
⟹ more parallelism

---

* Compiler make prediction (not silly predictions)
  ↳ execution time

Computers



ممكن يقرأ optimizer
كودات غبية
c = a*b
بس مش ال programer
بكتبها ← هو فعلياً
يقرأ IR اللي عشان تكون جاهزة

---

*Data flow analysis → اجمع داتا واستخدمها لقدام

[1] constant propagation

↳ ال pd اللي الهم قيم ثابتة أحوضهم بأماكن تانية

*return
c/2 +1 → بخليها c لانه صار عندي
مرة c=12 , c=10

⊛ Dead code

↳ sth I writ & never read. → dead variable

وال statement اللي بحط فيها dead var

↳ بتكون dead stat.

* بشتغل bottom up بدور من فوق لتحت ال var اللي بقرأهم
خليهم واللي ما بقرأهم تحت سميهم dead.

Lec 8

Introduction to Code Optimization Spring 2018/2019

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

Review: The Front-End Compiler

- The front-end compiler analyzes the input program to understand its meaning and generates a representation of this program, called the intermediate representation (IR)

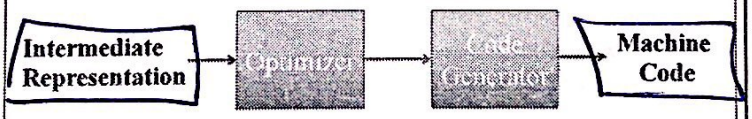
الجزء الاول من الكومبايلر سؤال عن المعنى ويطلع IR

3/12/19

Review: The Intermediate Representation

- Preserve the meaning of the input program
- Can be either graphical, linear, or hybrid of both
- Assume an abstract machine
- Assume unlimited registers
- Amenable for performing optimizations

The Back-End Compiler



- Optimizer: generates an improved IR
- Code Generator: converts the IR into machine code

This lecture will discuss code optimization

يسرع أكثر من

pass

Code Optimizer

- Analyze the IR and transform (i.e., rewrite) it into an *improved* IR version
- Structured into multiple passes
- Golden rule: new improved code must preserve correctness (i.e., applied optimizations must be *safe*)

الهم ~~ح~~ -
كلمة correct.

General Optimizer Framework

1. Analyze, at compile-time, the IR to discover information about the runtime behavior of the program
2. Use this information to apply safe transformations that improve the IR
3. Improvements can be one or many of the followings:
   - Lesser execution time
   - Lesser memory footprint
   - Lesser energy consumption
   - Higher parallelism utilization
   - etc

التحسين يكون
على هاي الاشكال

Scope of Optimization

1. *Local* optimizations are performed within a basic block
2. *Intra-procedural* or *Global* optimizations are performed across multiple basic blocks within a procedure
3. *Inter-procedural* or *whole-program* optimizations are performed across multiple procedures within a program

Code Optimization's Literature

- Code optimization has a very rich literature
- Remember Moore's law: the continuous appearance of new architectures naturally motivated researchers to make compilers "smarter"
- Given this course limited time:
  - We can only study few basic optimization techniques
  - But this is sufficient to understand the general challenges of performing automatic code optimization at compile-time

Lecture Roadmap

1. Study some traditional optimization techniques and understand how to apply them by hand
2. Study *data flow analysis*: a very common compiler framework used by optimizing compilers
3. If time permits, come back later and study *dependence analysis*: another common compiler analysis used by optimizing compilers

For simplicity, we will assume high-level IR codes in our discussion

بسوي ال exp
وبحط نتيجته بال
variable
(بعوض قيمة ال
var
direct

Constant Propagation

- Constant propagation (or folding) is a compiler technique that *recognizes* and *evaluates* expressions with constant values
- Advantages of constant propagation:
  - Simplify expressions, reducing execution time and code size in the process
  - Enable other optimizations

Constant Propagation Examples

```
X := 12                      X := 12
Y := X * 2 + Z      ⇒        Y := 24 + Z
return X + Y                 return 12 + Y

a := 3                       a := 3
b := 4                       b := 4
c := b * a          ⇒        c := 12
if ( cond )                  if ( cond )
    c := c – 2                   c := 10
endif                        endif
return c / 2 + 1             return c / 2 + 1
```

What about this *c* ??

depend on if result

ما بأثر على نتيجة البرنامج

Dead Code Elimination

- Dead code is a code that does not affect the outcome of the program
- Dead code includes:
  - Code that affects *dead variables* (written to, but never read again)
  - Code that can never be executed (aka *unreachable code*)
- Advantages of dead code elimination:
  - Reduce code size
  - In the case of dead variables, reduce execution time

dead → sth. I write & never read.

Dead Code Elimination
Examples

```
X := 12
Y := X * 2 + 1
Z := Z + X + Y
return Z
```

Constant propagation ⇒ 1

Dead variables

```
X := 12
Y := 25
Z := Z + 37
return Z
```

Dead code elimination ⇒ 2

```
Z := Z + 37
return Z
```

constant propagation

Dead Code Elimination
Examples

```
a := 3
b := 4
c := b * a
if ( c < 8 )
    c := c – 2
endif
return c / 2 + 1
```

Constant propagation ⇒ 1

```
a := 3
b := 4
c := 12
if ( 12 < 8 )
    c := c – 2
endif
return c / 2 + 1
```

Evaluates false

Unreachable code

هاد الاخر عمره ما ح يدتفذ ↓ dead

Dead code elimination ⇒ 2

```
a := 3
b := 4
c := 12
return c / 2 + 1
```

Constant propagation ⇒ 3

```
a := 3
b := 4
c := 12
return 7
```

Dead code elimination ⇒ 4

```
return 7
```

some times we have fully redundancy

Common Subexpression Elimination (CSE)

- CSE is a compiler technique that *identifies* and *removes* redundant (i.e., duplicate) expressions
- Advantages of CSE:
  - Reduce execution time
  - Reduce code size
  - Reduce number of needed temporaries or registers

CSE Example (1)

Redundant expressions

الهم نفس القيمة

```
T1 := A + B
T2 := T1 / 2
T3 := A + B
C := T3 * T2
```

⇒

```
T1 := A + B
T2 := T1 / 2
T3 := T1
C := T3 * T2
```

Incurs two loads instructions when translated to machine code

Single register-to-register move instruction

Top down

لانه جمع A,B بعدين

بطلع نتيجة T3

CSE Example (2)

Redundant expressions

احطم بـ Temp بدل ما اسوي ضرب مرتين!!

```
T1 := A * B + X        t  := A * B
T2 := A * B + Y   ⇒    T1 := t + X
                       T2 := t + Y

if ( cond )            t  := A * B
    T := A * B + X     if ( cond )
else                       T := t + X
    T := A * B + Y ⇒   else
                           T := t + Y
```


Loop Optimizations

اكثر اشي loops بياخذ وقت بالبرنامج

- *Common case scenario*: **loops** are where programs often spend most of their execution time
- Naturally, researchers have intensively studied how to optimize loops
  - Reduce their code size
  - Reduce their computation → اهم اشي
  - Optimize their cache behavior
- Next slides will introduce some loop optimization techniques

Loop-invariant Code Motion

- Loop-invariant code contains expressions whose evaluation never changes inside the loop
- The idea is to move this code outside the loop
- Advantages: remove redundant computation

قيمة x ثابتة فتربيعها ثابت

Loop-invariant Code Motion
Example 1

```
for ( i=1; i < n; i++)
    x = z + y ;
    A[ i ] = 2 * i + x * x ;
```

⇒

```
x = z + y ;
t = x * x ;
for ( i=1; i < n; i++)
    A[ i ] = 2 * i + t ;
```

```
for ( i=1; i < n; i++)
    for ( j=1; j < n; j++)
        B [i][j] = i * i + j * j;
```

Invariant with respect to j-loop

⇒

```
for ( i=1; i < n; i++)
    t = i * i ;
    for ( j=1; j < n; j++)
        B [i][j] = t + j * j;
```


كل مرة بسويها وهي كل مرة نفس النتيجة فبطلعها برة

* if X is not a func. of i ⟹ then X invarient

ما بتغير بتغير K فبطلعها فوق loop

K

Loop-invariant Code Motion
Example 2

Original loop

```
for ( i=1; i < n; i++)
    for ( j=1; j < n; j++)
        for ( k=1; k < n; k++)
            x [ k + j * n + i * n * n ] = a [ k + j * n + i * n * n ]
```

Invariant with respect to k-loop

Invariant with respect to k-loop and j-loop

Optimized loop

```
for ( i=1; i < n; i++)
    t1 = i * n * n ;
    for ( j=1; j < n; j++)
        t2 = j * n + t1 ;
        for ( k=1; k < n; k++)
            t3 = k + t2 ;
            x [ t3 ] = a [ t3 ];
```


هاي غلط لازم تطلع برة loop ؟

Strength Reduction

induction variable

اللي بزيدو بكم ثابت كل ما تلف ال loop

- **Definition**: a variable inside a loop is called an *induction variable* if it is incremented or decremented by a fixed amount in each iteration of the loop
- Strength reduction is a compiler technique that takes advantage of induction variables' property to replace expensive operations with simpler operations
  - E.g., replace division or multiplication operations by addition operations

when $i=1 \longrightarrow j=a$

$i=2 \longrightarrow j=2a$

$i=3 \longrightarrow j=3a$

بس اجمع a على j لانه الجمع اخف من ضرب

Strength Reduction Example

```
for ( i=1; i < n; i++)
    j = a * i
    a [ j ] = ....
```

⇒

```
j = 0
for ( i=1; i < n; i++)
    j = j + a
    a [ j ] = ....
```

```
for ( i=1; i < n; i++)
    t1 = i * n * n
    for ( j=1; j < n; j++)
        t2 = j * n + t1
        for ( k=1; k < n; k++)
            t3 = k + t2
            x [ t3 ] = a [ t3 ]
```

⇒

```
t1 = 0
for ( i=1; i < n; i++)
    t1 = t1 + n * n
    t2 = t1
    for ( j=1; j < n; j++)
        t2 = t2 + n
        for ( k=1; k < n; k++)
            t3 = k + t2
            x [ t3 ] = a [ t3 ]
```


عكس اللي فوق

الفائدة منه أقلل dependecy

Induction Variable Substitution

- This technique performs the opposite transformation of strength reduction, i.e, rewrite induction variables so that they are functions of the loop index

```
j = j0
for ( i=1; i < n; i++)
    j = j + a
    a [ j ] = ....
```

⇒

```
for ( i=1; i < n; i++)
    j = a * i + j0
    a [ j ] = ....
```

- This technique obviously introduces more expensive operations
- Nevertheless, this technique is substantially useful for many machines (can you give an example?)

$j = j + a \longrightarrow j = a \times i + j$

بعتمد على j اللي قبل

ما في اعتمادية

Loop Unrolling

- Replicate body of the loop by a factor of *n*
- Disadvantages:
  - Increase code size → increas memory footprint
- Advantages:
  - Reduce number of branches
  - Increase the opportunity of *instruction scheduling*

بزيد parallelism

```
for ( i=1; i < n; i++)
    a [ i ] = ....
```

Unroll by a factor of 4

```
for ( i=1; i < n; i=i+4)
    a [ i ] = ....
    a [ i + 1 ] = ....
    a [ i + 2] = ....
    a [ i + 3] = ....
```


Cache Behavior

- Many loop optimizations target cache performance by attempting to improve *temporal locality* or *spatial locality*
  - Temporal locality: reuse of recently accessed data
  - Spatial locality: access of nearby data
- Exercise: identify which array accesses have spatial or temporal locality in the following nested loop (assume row-major order)

```
for ( i=1; i < n; i++)
    for ( j=1; j < n; j++)
        x[ i ] = A[ i ][ j ] + x[ i ]
```


اجيب الداتا المجاورة

* اذا 2 seperate Loops بمرو على نفس العدد كلهم بعض. * fusion

temporal locality

* الفائدة ← ال Cashe عمالها Limit

بس يجيب a[i] ويروح 10000

راح يتغلب بس يرجع يجيب a[i]

Loop Fusion

- A compiler technique that combines two or more loops into a single loop
- Example:

```
for ( i=1; i < 10000; i++)
    a [ i ] = x
for ( i=1; i < 10000; i++)
    b [ i ] = a [ i ] * 5
```

⇒

```
for ( i=1; i < 10000; i++)
    a [ i ] = x
    b [ i ] = a [ i ] * 5
```

- The above transformation improves the cache behavior (how?)
- The above transformation also reduces number of branches

عكس اللي قبلها

Loop Fission

- Also known as loop distribution
- A compiler technique that breaks a single loop into multiple loops
  - Opposite technique of loop fusion
- Example:

```
for ( i=1; i < 10000; i++)
    a [ i ] = a [ i ] * 5
    b [ i ] = b [ i ] * 6
```

⇒

```
for ( i=1; i < 10000; i++)
    a [ i ] = a [ i ] * 5
for ( i=1; i < 10000; i++)
    b [ i ] = b [ i ] * 6
```

- The above transformation improve the cache behavior (how?)
- However, it increases the number of branches

بكل loop بيكون عندي one array

إيه أثر

*disadv. of Loop fission

بزيد عدد arrays بالكاش

وبزيد cashe miss

3/12/19

*adv. of fission

b[i] = a[i+1]

a[i] = c[i]

a[2] = a[3]

بس يوصل

a[2]

بكون عندي قبل

بسا بكل نفذتها

بس اذا كل وحده

بلوب بدي

لستى 10000

لحتى ارجع

انفذها

miss

Which One is Better: Loop Fusion or Loop Fission

- Both techniques affect loops' cache behavior
- Therefore, profitability depends on the *data access pattern* of loops and on the ability of compilers to predict cache behavior
- Compilers usually rely on heuristics and hardware information (if available) to make profitability decisions
- Exercise: argue if applying loop fusion for the following code is profitable or not (assume row-majored order)

```
for ( i=1; i < 10000; i++)
    b [ i ] = a [ i + 1]
for ( i=1; i < 10000; i++)
    a [ i ] = c [ i ]
```

⇒

```
for ( i=1; i < 10000; i++)
    b [ i ] = a [ i + 1]
    a [ i ] = c [ i ]
```


Loop Interchange

- A compiler technique that exchanges inner loops with outer loops in loop nests
- Loop interchange does not reduce computation but changes the order of accessing array elements
- Example:

```
for ( j=1; j < n; j++)
    for ( i=1; i < n; i++)
        y[ i ][ j ] = x[ i ][ j ] * 2
```

⇒

```
for ( i=1; i < n; i++)
    for ( j=1; j < n; j++)
        y[ i ][ j ] = x[ i ][ j ] * 2
```

- Explain how would the above transformation be useful?

لازم احافظ على correctness
خصوصاً

data flow ⟹ RAW

Do Not Forget
The Golden Rule

- Optimizations change the code and therefore their changes must be **safe**
- To apply an optimization by a compiler, **a static analysis is needed to ensure safety**
- Such compiler analyses are not trivial (remember the fundamental limitation of compilers)

Correctness Examples

S1: x = 1

S2: x = y * y

S3: y = x * x ⟶ x ≠ 1 because most recent definition of x is obtained from S2

Applying constant propagation is NOT safe

S1: t = x * y * z

S2: z = w * 2

S3: s = x * y * z ⟶ s ≠ t because z was re-defined by S2

Applying common subexpression elimination is NOT safe

Terminology

- A statement has a *definition* of variable X if it writes X
- A statement has a *use* of variable X if it reads X
- Let S1 and S2 be two statements where S2 follows S1 in execution, we say there is a *flow dependency* S1→S2 if there is a *definition* in S1 that produces a value that is consumed by a *use* in S2

```
S1:   X = ...          // S1 has a defintion of X
      \ S1 and S2 have a dataflow dependency
S2:   ... = X + ...    // S2 has a use of X
```


نوع ① لل optim.

Data Flow Analysis

- Ensuring safety of optimizations such as constant propagation and common subexpression elimination requires a compiler analysis that gathers information about the runtime flow of values from statements with variable definitions to statements with variable uses
- This analysis is referred to as the data flow analysis (DFA)
- DFA has been intensively used in the literature for enabling and applying many compiler optimizations
- We will study dataflow analysis in the next lecture

3/12/19

More Correctness Examples

Is applying loop fusion safe?

```
for ( i=1; i < 100; i++)
    a [ i ] = b [ i ]
for ( i=1; i < 100; i++)
    b [ i ] = a [ i + 1] + 2
```

⇒

```
for ( i=1; i < 100; i++)
    a [ i ] = b [ i ]
    b [ i ] = a [ i + 1] + 2
```

Original code	Optimized code
a[1] = b[1] a[2] = b[2] ... a[99] = b[99] ... b[1] = a[2] + 2 b[2] = a[3] + 2 ... b[99] = a[100] + 2	a[1] = b[1] b[1] = a[2] + 2 a[2] = b[2] b[2] = a[3] + 2 ... a[99] = b[99] b[99] = a[100] + 2

Optimized code violates the **dependencies** exist in the original code

⇓

Applying loop fusion is NOT safe

نوع ② optimi لل

Dependence Analysis

- Ensuring safety of optimizations such as loop fission and loop fusion requires a compiler analysis that gathers runtime information about the execution-order constraints between statements
- This analysis is referred to as the dependence analysis
- Similar to DFA, the dependence analysis has also been a hot topic among researchers for enabling and applying many compiler optimizations
- If the course time permits, we will come back later and study dependence analysis in more details

* ما بيعكس dependency ⇓ اذا اجا سؤال بالامتحان بنفذ الكود وبشوف legal or not اذا

* الكود الاصلي ← بكتب على a[2] قبل قراءتها ← RAW

* opt. code ← بقرأ a[2] القديمة ← illegal

Summary

- We studied some of the widely-known compiler optimization techniques and how they can be performed ***by hand***
- The next question is how to make these techniques **automatic**, i.e., what are the compiler analyses needed to perform these optimizations?
- **In the next lecture**, we will study **data flow analysis**: a widely-used compiler analysis for performing compiler optimization

Exercise

```
for ( i=1; i < n; i++)
    b [ i ] = a [ i + 1] * 5

for ( i=1; i < n; i++)
    a [ i ] = b [ i ]
```

Assume row-majored order

Assume n is in the order of millions

- Write the code obtained by applying loop fusion to the above code
- Explain why applying loop fusion is safe
- Discuss weather applying loop fusion for the above code is likely to be profitable or not

Lec 9

Data Flow Analysis
Spring 2018/2019

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

What is Data Flow Analysis?

- A compiler analysis that collects, for each program statement, a set of valid runtime data properties
- This is done by reasoning about the runtime flow of data properties across program statements
- To investigate the flow of data across different execution paths, DFA requires the *control flow graph* (*CFG*) representation of the IR

 2

1

The Data Flow Problem of Constant Propagation

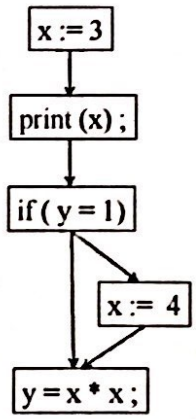
- To perform constant propagation, the compiler needs to collect information about which variables have constant values at each statement in the program
- Let S be a statement and x is a variable that is visible to this statement in a program, the compiler needs to determine one of the following properties about S:
  - S has the property $\{x = c\}$, where c is some constant
  - S has the property $\{x = T\}$, which means that x is not a constant

 5

Example 1

```
x := 3
print ( x ) ;
if ( y = 1 )
    x := 4
endif
y = x * x ;
```

CFG ⇒



For simplicity, let us assume single-statement basic blocks

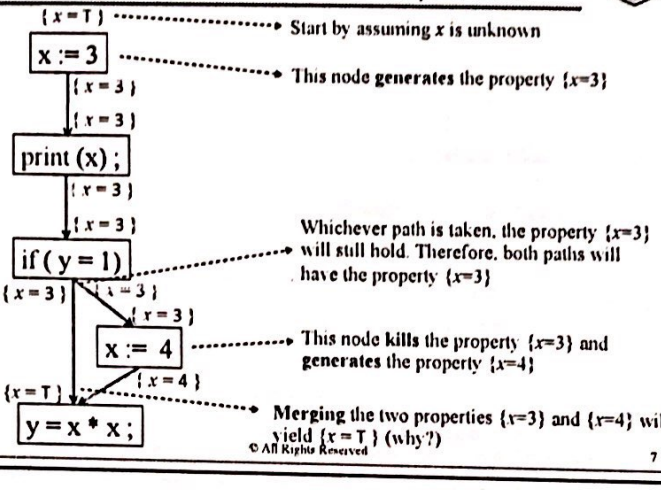
 6

* الجملة ال basic block بتكون في عبارة static

3

static نسبة للمحولة مع كون كل block

Example 1: Constant Values Properties



Start by assuming x is unknown

This node **generates** the property $\{x=3\}$

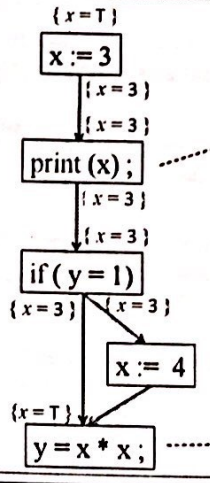
Whichever path is taken, the property $\{x=3\}$ will still hold. Therefore, both paths will have the property $\{x=3\}$

This node **kills** the property $\{x=3\}$ and **generates** the property $\{x=4\}$

Merging the two properties $\{x=3\}$ and $\{x=4\}$ will yield $\{x = T\}$ (why?)

7

Example 1: Applying Constant Propagation



The compiler knows that a=3 is true for all possible execution path to this statement

```
x := 3
print ( 3 ) ;
if ( y = 1 )
    x := 4
endif
y = x * x ; // no changes are made here
```

The compiler **cannot** prove statically that x=c for this statement

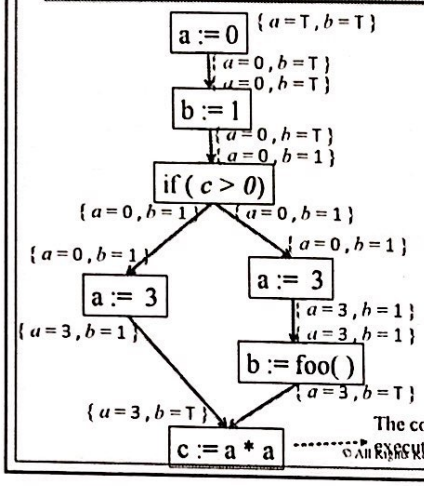
8

Kill → sth I write

Gen → sth I read

* we have 4 sets:-
Kill, gen, in, out

Example 2: Applying Constant Propagation




```
a := 0
b := 1
if ( c > 0)
    a := 3
    b := foo( )
else
    a := 3
endif
c := 9
```

The compiler knows that a=3 is true for all possible execution paths leading to this statement

Observations

We can observe the following concepts from the constant propagation data flow problem:

- When **transferring** knowledge through a node in the CFG, some properties are maybe **killed** and some new properties are maybe **generated**
- A **merge** operation is needed when *multiple paths are joined*
  - In the case of constant propagation, this merge operation is an intersection operation (i.e., at node *e*, x=c if and only if x=c holds for all paths leading to *e* in the CFG)
- The constant propagation data flow problem is **forward**, i.e., data properties flow from top to bottom through the CFG edges
  - **Backward** data flow problems also exit

1- transfer ⟶ out (in, gen, kill)
2- data flow ولا
3- merge func.

Data Flow Analysis Algorithm

- The compiler framework for DFA comprises the following three main components:
  1. Control Flow Graph (CFG): models the control flow in the program
  2. Data Flow Sets: collect the properties of interest (gen, kill, in, out)
  3. Data Flow Equations: compute the dataflow sets iteratively by traversing the CFG → merge, transfer

To understand how the algorithm works, we will study *Liveness* analysis
However, the same concepts still apply with other DFA problems

 13

sth I write & read.

Live Variables

- Definition: a variable *x* is ***live*** at a program statement *S* if there is a use of *x* in a future statement
  - This is the opposite definition of dead variables

```
S1:   x := 3
S2:   y := 3
S3:   z = y * x ;
```

⇒ S2 has the property {*x is live*} because a future statement (S3) uses *x*

```
S1:   x := 3
S2:   y := 3
S3:   z = y * y ;
```

⇒ S2 has the property {*x is dead*} because there is no use of *x* in future statements

 14

* Lpveness analysis → يرجعلي كل كلمة إذا dead or Live

7

* bottom up بكون → we care about future

Data Flow Equations

$$Live_{out}(e) = \bigcup_{x \in succ(e)} IN(x)$$

$$Live_{in}(e) = GEN(e) \cup \left(Live_{out}(e) - KILL(e)\right)$$

```
// assume CFG has N blocks
// numbered 0 to N-1
for i ← 0 to N-1
   LiveOut(i) ← ∅
changed ← true
while (changed)
   changed ← false
   for i ← 0 to N-1
      recompute LiveOut(i)
      if LiveOut(i) changed then
         changed ← true
```

The compiler uses this iterative algorithm for solving the equations

Solving The Equations

1. Build the CFG
2. For each node *e* in the CFG, determine GEN and KILL sets
3. Perform the data flow equations using the iterative algorithm in the previous slide
4. Terminate when the data flow sets reach a *fixpoint, i.e,* data flow sets do not change even if re-computed

Kill → اي شي بكتبه

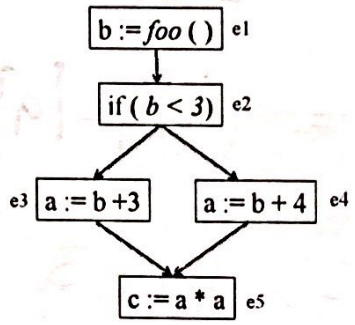
gen → اي اشي بقرأه

* حتى بال if انا بقرأ Var

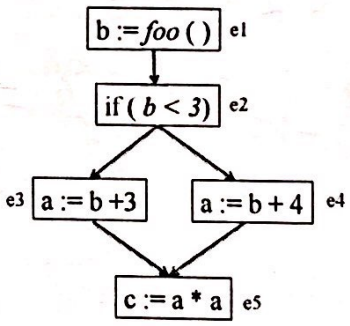
Example 1

① CFG

```
b := foo ( )
if ( b < 3)
    a := b + 3
else
    a := b + 4
endif
c := a * a
```



Example 1
GEN and KILL Sets



② Kill, Gen

	GEN	KILL
e1	Φ	b
e2	b	Φ
e3	b	a
e4	b	a
e5	a	c

write

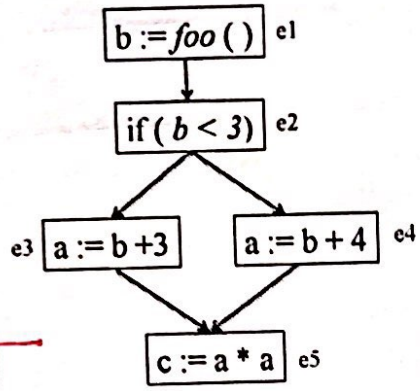
read

حتى بال if

يقرأ

vars.

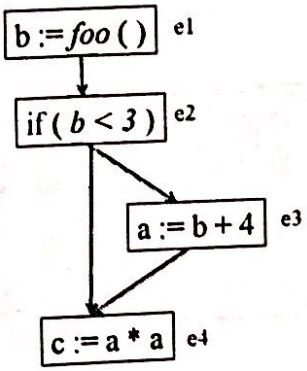
Example 1
$Live_{in}$ and $Live_{out}$ Sets



	GEN	KILL	$LIVE_{out}$	$LIVE_{in}$
e1	Φ	b	b	Φ
e2	b	Φ	b	b
e3	b	a	a	b
e4	b	a	a	b
e5	a	c	Φ	a

Example 2

```
b := foo ( )
if ( b < 3)
    a := b + 4
endif
c := a * a
```



Ex1

out e5 = $\emptyset$

In e5 = $(\emptyset - \{c\}) \cup \{a\} = \{a\}$

out e4 = In(e5) = $\{a\}$

In e4 = $(\{a\} - \{a\}) \cup b = \{b\}$

out e3, In e3 ⟶ نفس 4

out e2 = In(e3) ∪ In(e4) = $\{b\}$

In(e2) = $(\{b\} - \emptyset) \cup \{b\} = \{b\}$

out e1 = In e2 = $\{b\}$

In e1 = $(b - b) \cup \emptyset = \emptyset$

* بضل اطبق معادلات حتى اوصل stable

⟶ Live in = Live out

* بما انه بمشي buttomup بفضل آخذ من اخر node

Ex2

out $e4 = \emptyset$

In $e4 = (\emptyset - c) \cup a = \{a\}$

out $e3 =$ In $e4 = \{a\}$

In $e3 = (a - a) \cup b = \{b\}$

out $e2 =$ In$(e3) \cup$ In$(e4)$

$= b \cup a = \{a, b\}$ → correction باخذ a مكان

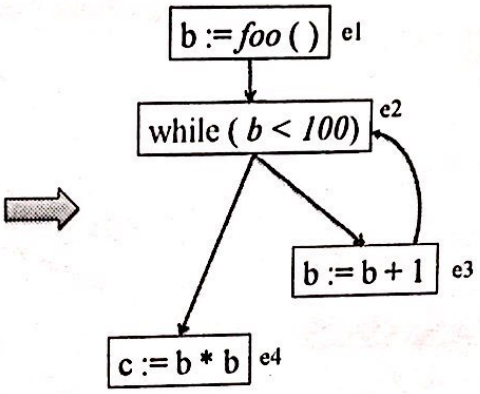
In $e2 = (\{a,b\} - \emptyset) \cup b = \{a, b\}$

out $e1 = \{a, b\}$

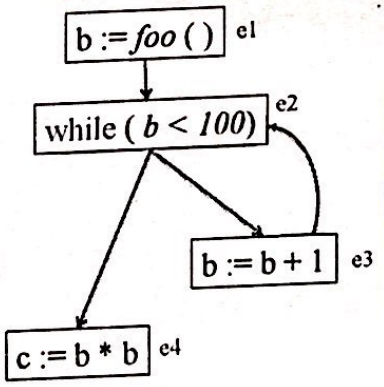
In $e1 = (\{a,b\} - b) \cup \emptyset = \{a\}$

Example 3

```
b := foo ( )
while ( b < 100 )
    b := b + 1
endwhile
c := b * b
```



Example 3
GEN and KILL Sets



	GEN	KILL
e1	Φ	b
e2	b	Φ
e3	b	b
e4	b	c

$out(e_4) = \phi$

$In(e_4) = (\phi - \{c\}) \cup \{b\} = \{b\}$

$out(e_3) = In(e_2) = \phi$? لسا ما لقدنا حتى

$In(e_3) = (\phi - \{b\}) \cup \{b\} = \{b\}$

$out(e_2) = In(e_3) \cup In(e_4) = \{b\}$

$In(e_2) = (b - \phi) \cup \{b\} = \{b\}$

$out(e_1) = \{b\}$

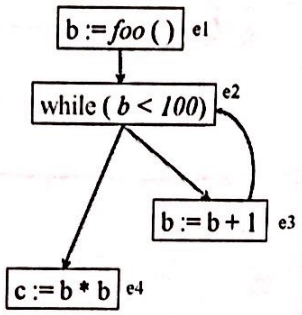
$In(e_1) = (\{b\} - \{b\}) \cup \phi = \phi$

* هون ما وصلنا stable point

* لازم اعيد مشان اوصل لها

* بالعادة اذا عندي loop بدي على الاقل 3,2 pass

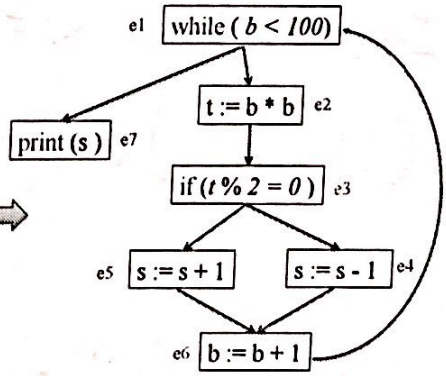
Example 3
$Live_{in}$ and $Live_{out}$ Sets



	GEN	KILL	$LIVE_{out}$	$LIVE_{in}$
e1	Φ	b	b	Φ
e2	b	Φ	b	b
e3	b	b	b	b
e4	b	c	Φ	b

Example 4

```
while ( b < 100 )
    t := b * b
    if ( t % 2 = 0 )
        s := s + 1
    else
        s := s - 1
    endif
    b := b + 1
endwhile
print (s )
```



Example 4

second pass

$out(e_7) = \phi$

$In(e_7) = (\phi - \phi) \cup \{s\} = \{s\}$

$out(e_6) = In(e_1) = \phi$ → $\{b, s\}$

$In(e_6) = (\phi - b) \cup \{b\} = \{b\}$ → $\{b, s\}$

$out(e_5) = In(e_6) = \{b\}$ → $\{b, s\}$

$In(e_5) = (\{b\} - \{s\}) \cup \{s\} = \{b, s\}$ → $\{b, s\}$

$out(e_4) = In(e_6) = \{b\}$ → $\{b, s\}$

$In(e_4) = \{b, s\}$

$out(e_3) = In(e_5) \cup In(e_4) = \{b, s\}$

$In(e_3) = (\{b, s\} - \phi) \cup \{t\} = \{b, s, t\}$

$out(e_2) = \{b, s, t\}$

$In(e_2) = (\{b, s, t\} - t\} \cup \{b\} = \{b, s\}$

$out(e_1) = \{b, s\}$

$In(e_1) = (\{b, s\} - \phi) \cup \{b\} = \{b, s\}$

Why The Knowledge of Live Variables is Useful

Live variable information enables many compiler uses transformation, such as:

- Dead variable stores elimination
- Uninitialized variables detection
- Register allocation (we will elaborate on this in the next lecture)

Lec 10

 33

things that are never used again

we have to write on var to read it

How To Design A Data Flow Analysis In General

- Same concepts we learned in *Liveness* analysis can be generalized to design a compiler analysis for any data flow problem
- Basically, a DFA can be designed by answering four questions:
  1. What are GEN sets?
  2. What are KILL sets?
  3. What is the analysis direction? (this question determines whether we use the forward or the backward dataflow problem)
  4. What is the merge operation? (needed when multiple paths are joined)

 34

General Data Flow Equations

Forward DFA

$$IN(e) = \bigvee_{x \in pred(e)} OUT(x)$$

$$OUT(e) = GEN(e) \cup (IN(e) - KILL(e))$$

Backward DFA

$$OUT(e) = \bigvee_{x \in succ(e)} IN(x)$$

$$IN(e) = GEN(e) \cup (OUT(e) - KILL(e))$$

the merge operation $\bigvee$ can be either a union or an intersection operation, depending on the DFA problem

35

ال In يتحسب قبل ال out وال In هي بتسوي merge

الهي

Example

امتحان

Design a DFA that determines, for each node in a CFG, the set of variables with constant values?

- As mentioned before, we need to specify four factors:
- For each node *e* in the CFG, we specify GEN and KILL sets to be the following:
  - GEN: set of variables that are assigned to a constant by *e*
  - KILL: set of variables that are defined by *e*
- DFA direction is forward
- Merge operation is an intersection operation
  - This is because, at a node *e* in the CFG, $x=c$ holds if and only if $x=c$ holds for all paths leading to e in the CFG have

data flow anal. لل design نعمل

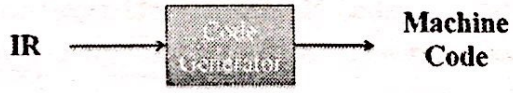
← Sol

Lec 10

Code Generation
Spring 2018/2019

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

Code Generator



- Final phase of the compilation process
- Converts the IR into machine code
- Code generation is split into the following three passes:
  - Instruction Selector
  - Instruction Scheduler
  - Register Allocator

3 passes for code generating

ترتيب كود الترتيب

Pattern matching

add $R_1, R_2, 5$ ⟹ mov $R_1, 5$
add R_2, R_1

First Pass:
* Instruction Selector

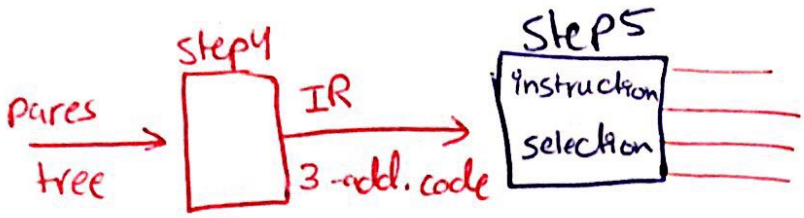
- Maps IR instructions into target machine (assembly) instructions
- Instruction selection is a *pattern-matching problem*: identify a correct and efficient sequence of machine instructions for each IR instruction
- Two commonly-used approaches for instruction selection:
  - Tree-pattern matching → visit patten → لإلها
  - String matching

يعمل map

Tree-pattern Matching

اخذنا قبل

- Typically used with tree-oriented IRs
- We partly covered this scheme in lecture 6
- Do a post-order walk of the IR tree and map each pattern to a sequence of target-machine instructions



step5 ← String Matching

- Typically used with linear IRs
- Instruction selector converts the IR code into machine code, as follows:
  1. **Expands** each instruction in the IR into an equivalent sequence of machine instructions, **in isolation**
  2. Perform **peephole optimization**

map ≡

يقلل الحجم

يمسك كل جملة و يحولها لـassembly

* من مميزات optimization

Example
Three-Address Code to Assembly

- Assume two-address assembly code
  - similar to *tiny* assembly code in the course project

```
ADD A, B, C        MOV A, T1
                   MOV B, T2
                   ADD T1, T2
                   MOV T2, C
MUL C, 4, D        MOV C, T3
                   MUL 4, T3
                   MOV T3, D
```

value in memory

tempor. reg.

Each IR instruction is translated separately

Note that T1-T3 are temporaries that will eventually be replaced by physical registers by the register allocator

MOV T2, C → كتبت على C

MOV C, T3 → نقل C على temp

* من مميزات optimization

ممكن أعمل mapping بأكثر من طريقة

وما بعرف مين احسن طريقة

The Fundamental Problem of Instruction Selection

- Most modern machines have multiple ways to express the same code
- Example: all the following instructions have the same outcome
  - MOV r1, r2
  - ADD r1, 0 , r2
  - MUL r1, 1, r2

  → same result
- Therefore, a group of IR instructions can have multiple machine code versions!
- Need to examine all machine code versions to find the best solution ⇒ can take exponential time!

guess on experiance ← decision

الحل اسوي

Heuristics Help

- Compilers approximate: use heuristics to decide which machine instructions to use

MUL A, 4, C

1
```
MOV A, T1
MUL 4, T1
MOV T2, C
```

2
```
MOV A, T1
SHL 2, T1
MOV T2, C
```

This version is more likely to be faster

→ SHL ≡ الضرب

ال shift أسهل فبختار الحل الثاني

بتيجي من الـ instruction
الزابت

The Redundancy Problem

- Translating each IR instruction in isolation often causes the generated machine code to have redundant work that may hurt performance

```
ADD A, B, C
ADD C, 4, D
```

```
MOV A, T1
MOV B, T2
ADD T1, T2
MOV T2, C      Wasteful
MOV C, T3      work
ADD 4, T3
MOV T3, D
```

The code we want to be generated

```
MOV A, T1
MOV B, T2
ADD T1, T2
ADD 4, T2
MOV T2, D
```

Solution: peephole optimization

نافذة لتمشي على مجموعة instru.
وبتسوي لكل مجموعة
opt. كال

عشان هيك
peep'hole

Peephole Optimizations

- Simple optimizations that are performed by pattern matching to remove work redundancy
- Essentially, the compiler looks "through a peephole" to a short sequence of assembly code and checks if it matches a pattern that can be replaced with better code
- Remember the Golden rule: correctness must be preserved

الهدف الأساسي الحا

Common Peephole Optimizations

1. Constant propagation

```
MOV  4, T1
ADD  T1 , T2   ⇒   ADD  4 , T2
```

- بقلل حجم برنامج
- " عدد temp

2. Strength reduction

```
MUL  4, T1   ⇒   SHL  2 , T1
DIV  4, T1   ⇒   SHR  2 , T1
```

3. Null sequence

```
MOV  0, T2
ADD  T1 , T2   ⇒   MOV T1 , T2
```


Common Peephole Optimizations

4. Combine operations

```
JEQ L1
JMP L2      ⇒   JNE L2
L1:             L1:
```

5. Removing redundant operations

```
MOV  4, T1
MOV  T1, T2   ⇒   MOV  4, T2
```

6. Address mode operations (@)

```
MOV  A, T1
ADD  T1 , T2   ⇒   ADD @(A) , T2
```


مباشرة جيب الادرس

تاع A حسب اللغة اذا

بتسمح)

Review: Superscalars

- Superscalars are processors that can issue and execute several operations per cycle
  - Accomplished by exploiting *instruction-level parallelism* and the fact that superscalars have multiple functional units
  - Most modern processors are superscalars
- Execution time is **order-dependent**: the order at which the instructions are issued dictates performance
- The compiler helps: attempt to reorder assembly instructions to maximize instruction-level parallelism, i.e., maximize the number of operations that can be issued in the same cycle

لكن الكومبايلر Just help بيعطي code فيها parallelism

* هاي شغلة هارد وير

reorder to decrease time (superscalar) → out of order execution

Second Pass: Instruction Scheduler

Assembly Code → Instruction Scheduler → Reordered Code

- Instruction scheduling is the process whereby a compiler reorders the instructions in the compiled code to decreases its running time
- Opportunities for scheduling:
  - Identify independent operations: operations that can be issued in parallel
  - Insert *non-blocking operations* between dependent operations to hide their waiting time

هاد الكود النهائي بس بعدل عليه تحسنه

flow from consumer to producer

Data Dependency Types

- Let S1 and S2 be two statements where S2 executes after S1
- There are three types of dependencies:

1. Flow dependency
   - Also known as read-after-write (RAW) dependency
   - Occurs when a use in S2 depends on a definition in S1
2. Anti-dependency
   - Also known as write-after-read (WAR) dependency
   - Occurs when a definition in S2 overrides a variable that is used by S1
3. Output dependency
   - Also known as write-after-write (WAW) dependency
   - Occurs when a definition in S2 overrides a variable that is also defined by S1

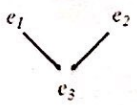
يطلع dependence graph

Dependence Analysis

- A compiler analysis that discovers data dependencies between operations and produces a *dependence graph* to models these dependencies
- The dependence graph is a directed graph that has a node for each operation, where an edge connects two nodes *e1* and *e2* if *e2* depends on *e1*

parallel

e1: X = A * B
e2: Y = D * C
e3: Z = X + Y



e1 and *e2* can execute in parallel because they have no connecting edge

* كل instru. → node
* كل edeg → dependence

* اللي ما فيه edge ← parallel
* اتجاه السهم حسب مين لازم يطلع قبل مين

Instruction Scheduling Algorithm

- The currently dominant technique for static instruction scheduling is a greedy heuristic called *list scheduling*
- List schedulers take advantage of the dependence graph to reorder a set of operations
- Each operation is given a *rank* to indicate their priority (higher-priority operation is chosen when two operations are ready for scheduling)
  - A popular priority scheme is to use operations latencies

بتعطي كل instruction
↓
rank
وبمشي حسب priority

List Scheduling

The algorithm of list scheduling comprises the following four steps:

1. Use *register renaming* to eliminate anti- and output dependencies
2. Build the dependence graph
3. Compute the priority for each node in the dependence graph
4. Schedule operations while reflecting dependencies and priorities

* ما تقدر اخلص من RAW بال regi. renaming *

أحد كل WAW , WAR بحين بسوي

rename

Register Renaming

- In contrast to flow dependency, anti- and output dependencies are only name dependency, i.e., occurred because the same temporary (or register) variable is used
- Solution: rename registers so that dependent instructions use different registers ⇒ which **eliminates** the dependency
- Remember that the instruction scheduler uses temporary (i.e., virtual registers), which are infinite!
  - Register allocator will eventually fix everything by mapping virtual registers to physical registers

Scheduling Example

- We will show how list scheduling algorithm will reorder the following **MIPS** code

e1: LD T1, A
e2: ADD T1, T1, T1
e3: LD T2, B
e4: MUL T1, T1, T2
e5: LD T2, C
e6: MUL T1, T1, T2
e7: LD T2, D
e8: MUL T1, T1, T2
e9: SW T1, A

operation	Latency
LD & SW	3 cycles
MUL	2 cycles
ADD	1 cycle

We need operations' latencies for determining their priority

معلومات
هارد ويير
عرفها
من خلال
Heuristics

Scheduling Example (cont.)

1. Perform register renaming

عيدت كل الاسماء

e1: LD T1, A		e1: LD T1, A
e2: ADD T1, T1, T1		e2: ADD T2, T1, T1
e3: LD T2, B	Register Renaming	e3: LD T3, B
e4: MUL T1, T1, T2		e4: MUL T4, T2, T3
e5: LD T2, C		e5: LD T5, C
e6: MUL T1, T1, T2		e6: MUL T6, T4, T5
e7: LD T2, D		e7: LD T7, D
e8: MUL T1, T1, T2		e8: MUL T8, T6, T7
e9: SW T1, A		e9: SW T8, A

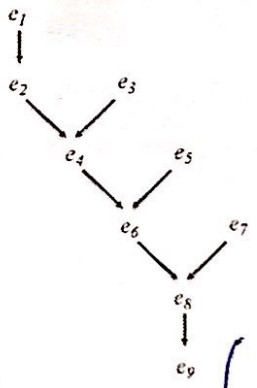
كل كل ما عدا RAW

Scheduling Example (cont.)

2. Build the dependence graph

label

e1: LD T1, A
e2: ADD T2, T1, T1
e3: LD T3, B
e4: MUL T4, T2, T3
e5: LD T5, C
e6: MUL T6, T4, T5
e7: LD T7, D
e8: MUL T8, T6, T7
e9: SW T8, A



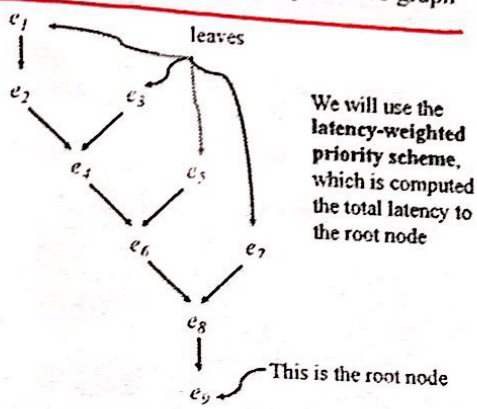
dependence graph

اعطي كل node
rank
حسب حساب معينة

Scheduling Example (cont.)

3. Compute a priority function for each node in the dependence graph

e1: LD T1, A
e2: ADD T2, T1, T1
e3: LD T3, B
e4: MUL T4, T2, T3
e5: LD T5, C
e6: MUL T6, T4, T5
e7: LD T7, D
e8: MUL T8, T6, T7
e9: SW T8, A

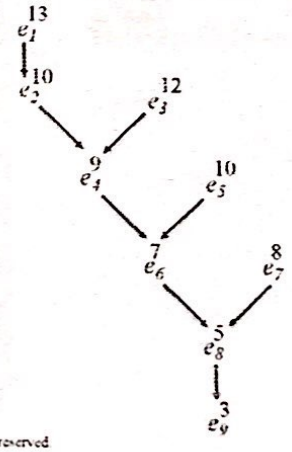


We will use the **latency-weighted priority scheme**, which is computed as the total latency to the root node

Scheduling Example (cont.)

3. Compute a priority function for each node in the dependence graph

e1: LD T1, A
e2: ADD T2, T1, T1
e3: LD T3, B
e4: MUL T4, T2, T3
e5: LD T5, C
e6: MUL T6, T4, T5
e7: LD T7, D
e8: MUL T8, T6, T7
e9: SW T8, A



Scheduling Example (cont.)

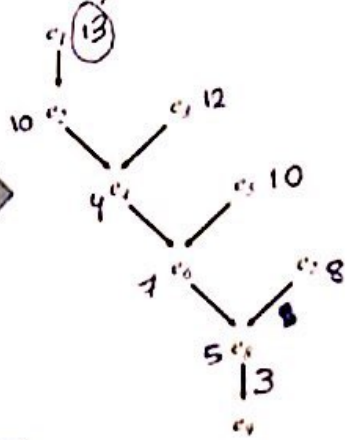
① Perform register renaming

e1: LD T1, A		e1: LD T1, A
e2: ADD T1, T1, T1		e2: ADD T2, T1, T1
e3: LD T2, B	Register Renaming →	e3: LD T3, B
e4: MUL T1, T1, T2		e4: MUL T4, T2, T3
e5: LD T2, C		e5: LD T5, C
e6: MUL T1, T1, T2		e6: MUL T6, T4, T5
e7: LD T2, D		e7: LD T7, D
e8: MUL T1, T1, T2		e8: MUL T8, T6, T7
e9: SW T1, A		e9: SW T8, A

Scheduling Example (cont.)

② Build the dependence graph (easy step)

e1: LD T1, A
e2: ADD T2, T1, T1
e3: LD T3, B
e4: MUL T4, T2, T3
e5: LD T5, C
e6: MUL T6, T4, T5
e7: LD T7, D
e8: MUL T8, T6, T7
e9: SW T8, A



latency عشان اعرف اسوي code schedual.

labels

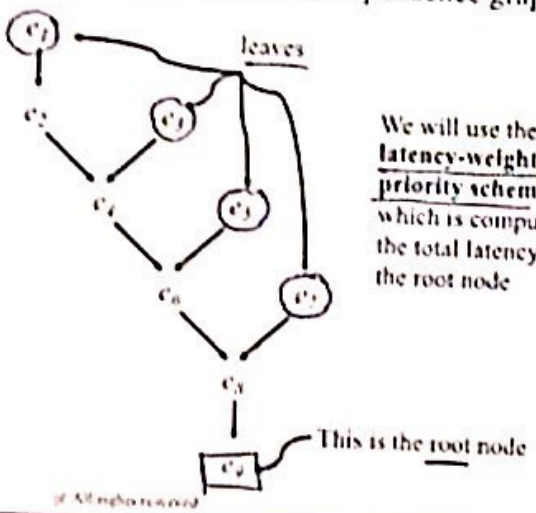
• الي فش بينهم خط ← مافي بينهم dependency

• نبلش ~~keep down~~ buttom up

Scheduling Example (cont.)

3. Compute a priority function for each node in the dependence graph

e1: LD T1, A
e2: ADD T2, T1, T1
e3: LD T3, B
e4: MUL T4, T2, T3
e5: LD T5, C
e6: MUL T6, T4, T5
e7: LD T7, D
e8: MUL T8, T6, T7
e9: SW T8, A



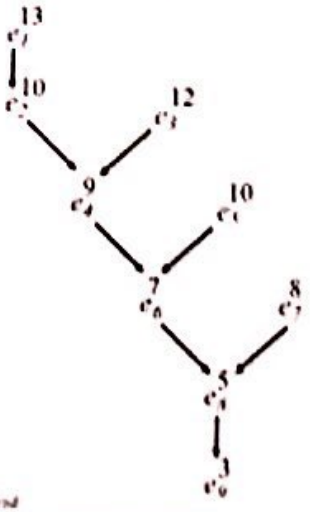
We will use the **latency-weighted priority scheme**, which is computed as the total latency to the root node

نعتمد على latency

Scheduling Example (cont.)

3. Compute a priority function for each node in the dependence graph

e1: LD T1, A
e2: ADD T2, T1, T1
e3: LD T3, B
e4: MUL T4, T2, T3
e5: LD T5, C
e6: MUL T6, T4, T5
e7: LD T7, D
e8: MUL T8, T6, T7
e9: SW T8, A



Scheduling Example (cont.)

4) Schedule instruction using the below algorithm

```
Cycle ← 1
Ready ← leaves of D
Active ← 0
while (Ready ∪ Active ≠ 0)
  for each op ∈ Active
    if (S(op) + delay(op) ≤ Cycle) then
      remove op from Active
      for each successor s of op in D
        if (s is ready) then
          Ready ← Ready ∪ s
  if (Ready ≠ 0) then
    remove an op from Ready
    S(op) ← Cycle
    Active ← Active ∪ op
  Cycle ← Cycle + 1
```

Implement Ready as a priority queue

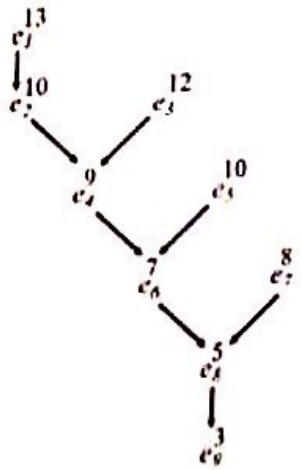
op has completed execution

If successor's operands are "ready", add it to Ready

Remove in priority order

code scheduling

Scheduling Example (cont.)



Schedule	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	

Ready	<e_1, 13>, <e_2, 12>, <e_3, 10>, <e_7, 8>
Active	

Initially:
- Add all leaves to Ready queue
- Active queue is empty

2 priority queues

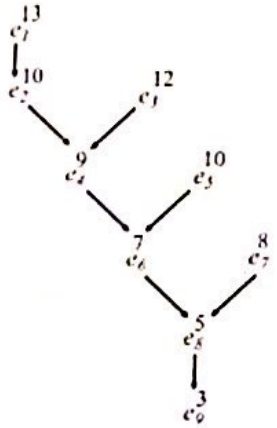
ready

اللي جاهزين يدخلو

(ما بستنوا نتيجة

من حدا)

Scheduling Example (cont.)



Schedule	
1	e_1
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	

⇐ Current cycle

$e1$ is removed first from Ready queue because it has highest priority

Ready $< e_3, 12 >, < e_5, 10 >, < e_7, 8 >$

Active $< e_1, \#4 >$

cycle where e_1 finishes execution

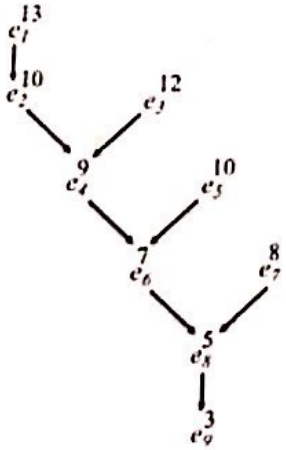
time

1 + 3 = 4

بتفاج ←

بلبش بالي موجود على top ready

Scheduling Example (cont.)



Schedule	
1	e_1
2	e_3
3	
4	
5	
6	
7	
8	
9	
10	
11	

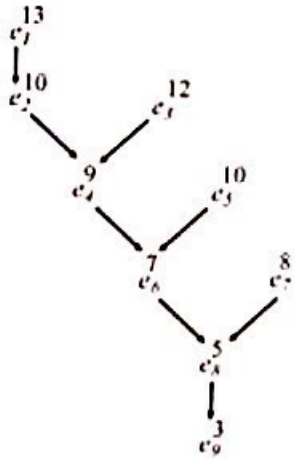
⇐

Ready $< e_5, 10 >, < e_7, 8 >$

Active $< e_1, \#4 >, < e_3, \#5 >$

دخلت بالجدول عند 2 وبدها 3 & 5

Scheduling Example (cont.)



Schedule	
1	e_1
2	e_3
3	e_5 ⇐
4	
5	
6	
7	
8	
9	
10	
11	

Ready	< e_7, 8 >
Active	< e_1, #4 >, < e_3, #5 >, < e_5, #6 >

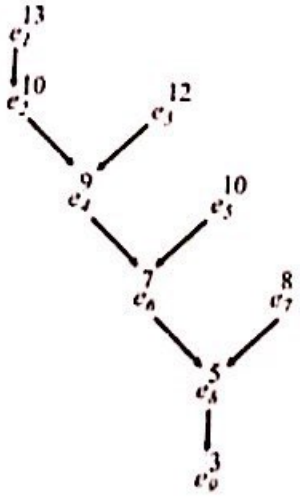
3 + 3 = 6

* يكون هون بس خلص e5
وينبي على رقم 4
بتكون خلصت e_1

* لازم دائماً استيك على active مين خلص بشيلو

* بعدين بختار من ready مين يدخل

Scheduling Example (cont.)



Schedule	
1	e_1
2	e_3
3	e_5
4	e_2 ⇐
5	
6	
7	
8	
9	
10	
11	

Ready	< e_7, 8 >, < e_2, 10 >
Active	< e_1, #4 >, < e_3, #5 >, < e_5, #6 >

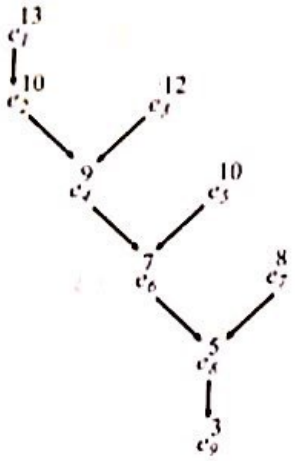
e_1 has completed

e_2 is now ready

Ready	< e_7, 8 >
Active	< e_1, #5 >, < e_3, #5 >, < e_5, #6 >

بتخلص عند 6

Scheduling Example (cont.)



Schedule	
1	e_1
2	e_3
3	e_5
4	e_2
5	e_4
6	
7	
8	
9	
10	
11	

Ready $<e_7, 8>, <e_4, 9>$

Active $<e_2, \#5>, <e_3, \#5>, <e_5, \#6>$

e_2 and e_5 have completed

e_4 is now ready

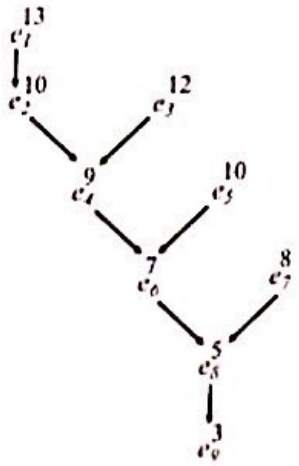
Ready $<e_7, 8>$

Active $<e_5, \#6>, <e_4, \#7>$

بتدخل e4

e3, e2 و

Scheduling Example (cont.)



Schedule	
1	e_1
2	e_3
3	e_5
4	e_2
5	e_4
6	e_7
7	
8	
9	
10	
11	

Ready $<e_7, 8>$

Active $<e_5, \#6>, <e_4, \#7>$

e_5 have completed

Ready

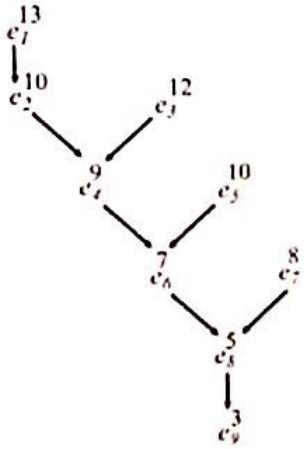
Active $<e_4, \#7>, <e_7, \#9>$

ما بدخل e6

لانه بعتمد على e4

اللي لسا ما وصلنا لها

Scheduling Example (cont.)



Schedule	
1	e_1
2	e_3
3	e_5
4	e_2
5	e_4
6	e_7
7	e_6
8	
9	
10	
11	

Ready $< e_6, 7 >$

Active $< e_4, \#7 >, < e_7, \#9 >$

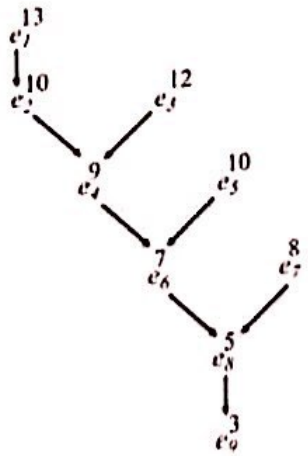
e_6 is now ready

e_4 have completed

Ready

Active $< e_6, \#9 >, < e_7, \#9 >$

Scheduling Example (cont.)



Schedule	
1	e_1
2	e_3
3	e_5
4	e_2
5	e_4
6	e_7
7	e_6
8	nop
9	
10	
11	

Ready

Active $< e_6, \#9 >, < e_7, \#9 >$

e_6 and e_7 are still active

No operation is ready at this cycle

ما عندي active حتى يدخل

Scheduling Example (cont.)

Schedule	
1	e_1
2	e_3
3	e_5
4	e_2
5	e_4
6	e_7
7	e_6
8	nop
9	e_8
10	nop
11	e_9

I ignore the nops but you can put them if you want

The reordered MIPS code produced by the instruction scheduler

```
e1: LD   T1, A
e3: LD   T3, B
e5: LD   T5, C
e2: ADD  T2, T1, T1
e4: MUL  T4, T2, T3
e7: LD   T7, D
e6: MUL  T6, T4, T5
e8: MUL  T8, T6, T7
e9: SW   T8, A
```

هي الترتيب الجديد (ما بحط في Nop)

لأنه code compiler مش كود hard-ware

Tie Breaking in List Scheduling

- When two or more ready operations have the same rank, list scheduler needs another priority scheme to break ties
- Several schemes have been proposed for breaking ties
- Examples:
  - Prioritize the node that has the more expensive operation
  - Prioritize the node with the higher number of immediate successors in the dependence graph
  - Prioritize the node with the higher number of total descendants in the dependence graph

or, randomly

* اذا لقيت تنين نفس ال priority

ح اعمل strategy معينة

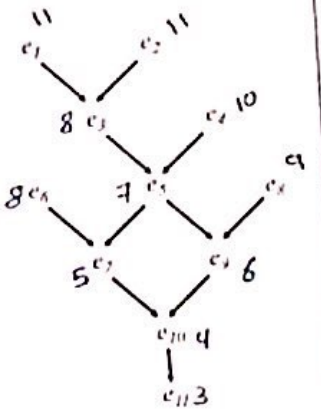
3/29/19

ممكن بالامتحان يعطينا هيك / مش كود

Exercise

Consider the shown data dependency graph to answer the following questions:

1. Using latency-weighted priority scheme, determine the priority function for each node (nodes latency are shown in the table below)
2. Using the list scheduling algorithm, write the scheduling order of the graph nodes. If there is a tie in priority, choose the node with smallest label number



operation	Latency
$e_1, e_2, e_4, e_6, e_8, e_{11}$	3 cycles
e_9	2 cycles
e_3, e_5, e_7, e_{10}	1 cycle

اكل ٥/١

بالنسبة لـ e_5 · الها طريقتين باخذ worst case (الأول)

اقترح علي اذا تساوو الـ priority اختار الي رقم الـ label الها أقل

Bonus: Regional Scheduling

X

- We studied list scheduling, which is a heuristic technique for scheduling instructions in basic blocks (maximal-length of sequential instructions)
- What about scheduling for an entire program?
- Section 12.4 in the text book introduces *regional scheduling*: techniques that can schedule instructions across multiple basic blocks by relying on the control flow graph representation of the program

الكودات الي أخذنا كان بس على local scope

+ ما كان منا Loops & branches

Recap

- Instruction selector and instruction scheduler produce an assembly code that assumes, for simplicity, virtual registers (which are infinite)
- Machines have limited registers, i.e., correct assembly codes must follow this constraint
- The next and the final step in the compilation process is to assign hardware physical registers to program values in the assembly code

Next lecture will cover the third and final pass of code generation: register allocation

تمار فبختار الاقل

ready	$e_{1,11}$	$e_{4,10}$	$e_{6,8}$	$e_{5,7}$
	$e_{2,11}$	$e_{8,9}$	$e_{3,8}$	
active	$e_{1,4}$	$e_{4,6}$	$e_{3,6}$	$e_{5,8}$
	$e_{2,5}$	$e_{8,7}$	$e_{6,9}$	

sol exercise

1	e_1
2	e_2
3	e_4
4	e_8
5	e_3
6	e_6
7	e_5
8	e_9
9	e_7
10	e_{10}
11	e_{11}

empty

ready	$e_{6,9}$	e_{10}	
active	$e_{6,9}$	$e_{9,10}$	$e_{10,11}$
	$e_{5,8}$	$e_{7,10}$	

فيها