
Lect
05
CST8152
Compilers

Algonquin College

Computer Engineering Technology

CST8152 Compilers

Fall, 2023

Lect 05

Based on resources developed by prof. **Svillen Ranev.**

Prof. Paulo Sousa

Algonquin College

Computer Engineering Technology

CST8152 Compilers

Fall, 2023

Lect 05

Formal Languages

Based on resources developed by prof. **Svillen Ranev.**

Prof. Paulo Sousa

Art 5: Formal Languages

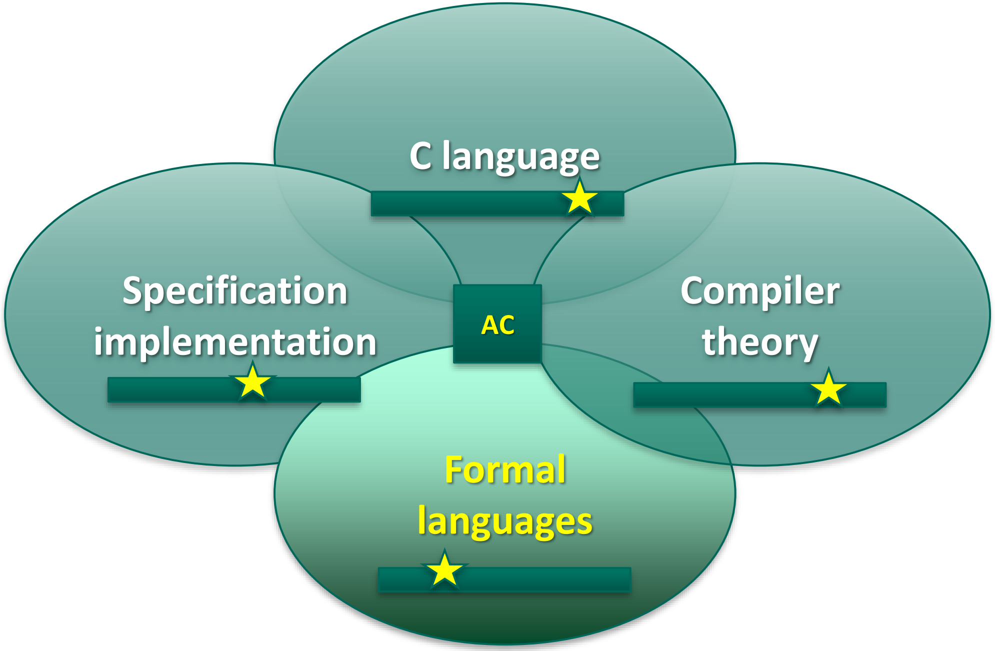
- *Formal Representations*
- *Regular Expressions*
- *Grammar*
- *Automata*

L1	L2
A2W01	A2W09
A2W02	A2W10
A2W03	A2W11
A2W04	A2W12
A2W05	A2W13
A2W06	A2W14
A2W07	A2W15

A2W08

ALGONQUIN COLLEGE

Let's start…

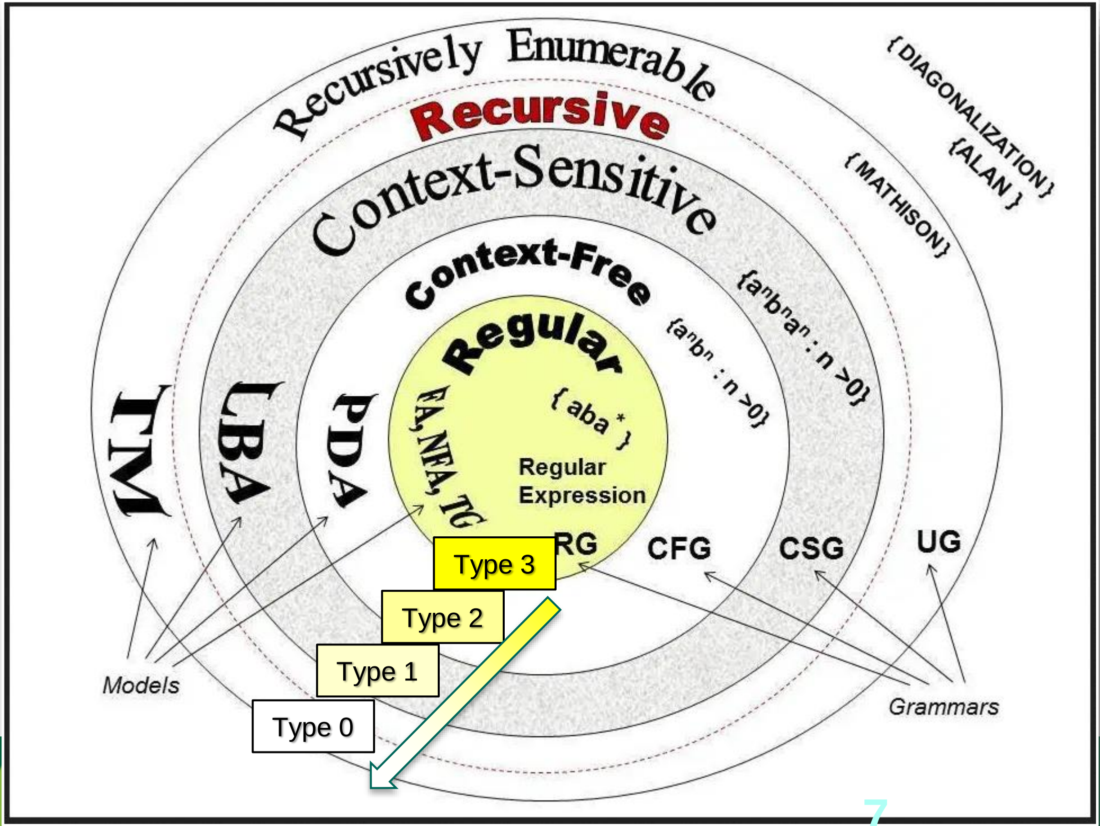

C language
Specification implementation
AC
Compiler theory
Formal languages

Compilers – Art. 5

Chomsky Models

ALGONQUIN COLLEGE

General Models (take a breath…)



Think about this [1]:

- What is the best model for PL?

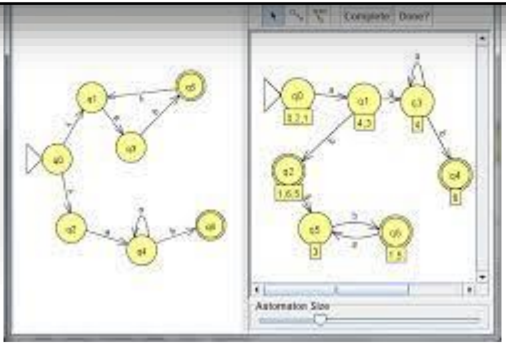
Source: https://i2.wp.com/www.theoryofcomputation.co/wp-content/uploads/2018/09/Chomsky_Hierarchy.jpg

General Models (take a breath…)

Examples

Grammar	Languages	Model	Constrains	Example
Type-3	**Recursively enumerable**	Turing machine	$\gamma \to \alpha$	$L = \{w \mid w \in TM\}$
Type-2	**Context-sensitive**	Linear-bounded machine	$\alpha A \beta \to \alpha \gamma \beta$	$L = \{a^n b^n c^n \mid n > 0\}$
Type-1	**Context-free**	Push-down automata	$A \to \alpha$	$L = \{a^n b^n \mid n > 0\}$
Type-0	**Regular**	Finite state automata	$A \to a \mid aB$	$L = \{a^n \mid n \geq 0\}$

JFLAP



Compilers – Art. 5

Formal Representations

Check the Examples

Language: Julius:

1. You can see different elements such as comments, keywords, identifiers, methods, constants and separators.

2. Most languages must define rules to recognize these elements (ex: <id,1>, etc.)

3. Different strategies can be used.

```
# Julius Example (Volume of a sphere) #
main& {
  data {
    real PI%, r%, Vol%;
  }
  code {
    PI% = 3.14;
    input&(r%);
    Vol% = 4.0 / 3.0 * PI% * (r% * r% * r%);
    print&(Vol%);
  }
}
```

Understanding the Kleene Theorem

- **Main Idea:**

Theorem

The language that can be defined by any of these three methods

1. Regular Expressions (or Regular Grammar)

   or

2. Transition graph (transition or state diagrams)

   or

3. Finite Automaton (Finite State Machine)

Prof. Kleene

Source: Wikipedia

The language that can be defined by:

1. Regular Expressions (compact language);
2. Regular Grammar (syntax production rules);
3. Finite Automaton (DFA / NFA);
4. Transition graph (transition / state diagrams);
5. Lambda calculus (math definition)

Understanding the Theorem

Model 1: Regular Expressions:

1. Regular expressions are a **convenient notation** (or means or tools) for specifying certain simple (though possibly infinite) set of strings over some alphabet.

2. A regular expression is a shorthand equivalent to a regular grammar.

L(RE) = L(G)

Code

```
var str = "EduCBA";
var regEx = /^[A-Za-z]/;
var res = "false";
if(str.match(regEx)){
res= "true";
}
alert(res);
```

Output:

true

Understanding the Kleene Theorem

Model 2: Grammar:

1. A finite set of terminal symbols (constants)
2. A finite set of non-terminals (notations to define rules).
3. A finite set of productions.
4. A symbol to start a language.

<arithmetic variable identifier> → <letter> <opt_letters or digits>

- Example: {a, b, c, …,abc, abcdf, abc123…}

<opt_letters or digits> → <letters or digits> | ε

- Example: {ε, a, b, c, …, abc, abcdf, abc123…}

<letters or digits> → <letter or digit> | <letters or digits> <letter or digit>

- Example: {a, b, c, … ,1, 2, 3, 1s, e2 …1111, aaaaa,…}

<letter or digit> → <letter> | <digit>

- Example: {a, b, c, … ,1, 2, 3 }

<letter> → a | b |…| z | A | B | …| Z

- Example: {a, b, c, … , z, A,…, Z }

<digit> → 0 | … | 9

- Example: {0, 1, 2, 3, …, 9 }

Understanding the Kleene Theorem

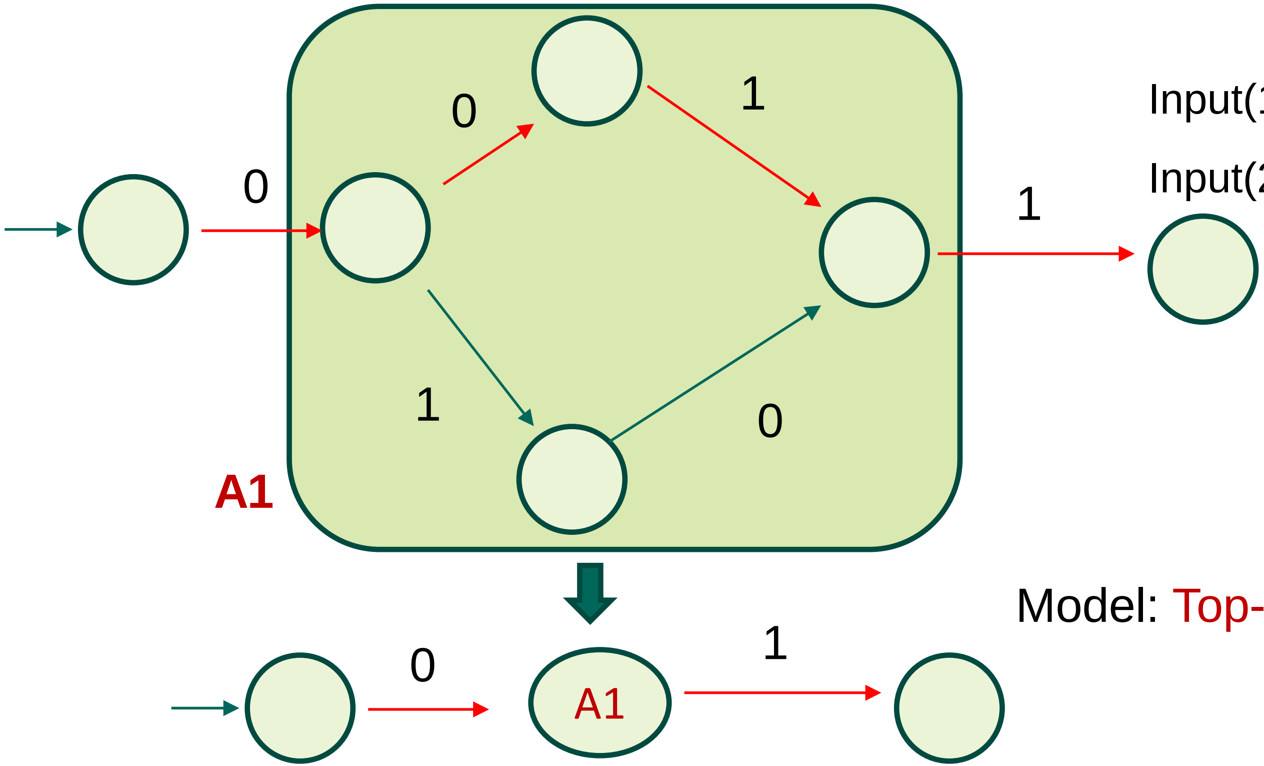
Model 3: Automata:

- A regular expression can be used to construct a **Deterministic Finite Automaton (DFA)** which therefore can recognize strings (words) of the grammar, which is the purpose of the Scanner.
- The sets of strings defined by regular expression are termed **regular sets**.

To define the RE (as any expression notation) use operands and operations.

- The **operands** are alphabet symbols or strings defined by regular expressions (regular definitions).
- The standard operations are **catenation** (concatenation) , union or **alternation** (|), and **recursion** or Kleene closure (*)
- Regular expressions use the **metasymbols** |, (,), {, } , [,], * , + (and others ?, ^) to define its operations.

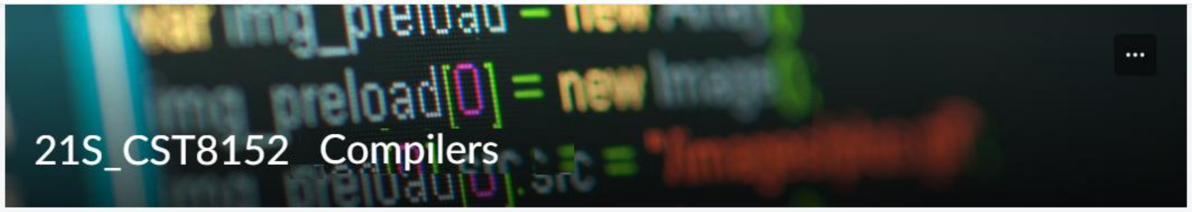
Automata



Input(1): 0011 (recognized)

Input(2): 0001 (NOT recognized)

Model: Top-down view can be done.



Compilers – Art. 5

Lambda Calculus

Initial Concepts

Alonzo Church Idea

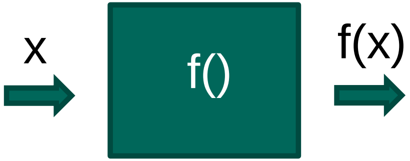
The lambda calculus (also written as λ-calculus, where lambda is the name of the Greek letter λ) was created by Alonzo Church in the early 1930s to study which functions are computable.

- In addition to being a concise yet powerful model in **computability theory**, the lambda calculus is also the simplest functional programming language.
- So much so that the lambda calculus looks like a “toy” language, even though it is (provably!) as powerful as any of the programming languages being used today, such as JavaScript, Java, C++, etc.

https://opendsa-server.cs.vt.edu/OpenDSA/Books/PL/html/Syntax.html

Lambda Calculus (1)

Model:



Abstraction for functions (no internal state is important).

We just have:

- Variables
- Functions (how to define/apply)

We do not have:

- Datatypes
- Controls

Several definitions are functions:

- Constants
- Operations
- Expressions.

Lambda Calculus [1]

Grammar (Lambda Calculus)

```
<expression> -> constant
              | variable
              | (<expression> <expression>)
              | (variable.<expression>)
```

The basic operation of the **lambda calculus** is the application of expressions such as the lambda abstractions.

Example: $(\lambda x.\ (x+1))$ or $(\lambda x.\ +\ 1\ x)$

- This is a definition of a function that adds 1 to an arbitrary number x.
- The expression $(\lambda x.\ x+1)$ represents the application of the function that adds 1 to x to the constant 2.
- Lambda calculus provides a reduction rules that permits 2 to be substituted for x in the lambda abstraction and removing the lambda producing the value:

$$(\lambda x.\ x+1)\ 2 \Rightarrow (2+1) \Rightarrow 3$$

Lambda Calculus (2)

Evaluating Lambda Calculus

Pure lambda calculus has no built-in functions. Let us evaluate the following expression –

```
(+ (* 5 6) (* 8 3))
```

Here, we can't start with '+' because it only operates on numbers. There are two reducible expressions: (* 5 6) and (* 8 3).

We can reduce either one first. For example –

```
(+ (* 5 6) (* 8 3))
(+ 30 (* 8 3))
(+ 30 24)
= 54
```

https://www.tutorialspoint.com/functional_programming/functional_programming_lambda_calculus.htm

Lambda Calculus (3)

β-reduction Rule

We need a reduction rule to handle λs

```
(λx . * 2 x) 4
(* 2 4)
= 8
```

This is called β-reduction.

The formal parameter may be used several times −

```
(λx . + x x) 4
(+ 4 4)
= 8
```

When there are multiple terms, we can handle them as follows −

```
(λx . (λx . + (− x 1)) x 3) 9
```

The inner **x** belongs to the inner **λ** and the outer x belongs to the outer one.

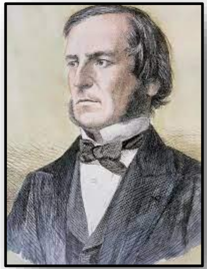
```
(λx . + (− x 1)) 9 3
+ (− 9 1) 3
+ 8 3
= 11
```

https://www.tutorialspoint.com/functional_programming/functional_programming_lambda_calculus.htm

Practical Example: Boolean Logic [(1)]

Logical Interpretation:

- Basic values are used.
  - EX: TRUE / FALSE
- Functions:
  - **TRUE** = λx. λy . x
  - **FALSE** = λx. λy . y
  - **NOT** = λx.x FALSE TRUE



- Example 1:
  - NOT TRUE =
  - λx.x FALSE TRUE TRUE =
  - (λx.λy.x) FALSE TRUE =
  - FALSE

- Example 2:
  - NOT FALSE=
  - λx.x FALSE TRUE FALSE =
  - (λx.λy.y) FALSE TRUE =
  - TRUE

More Logical functions:

- **AND** = λx.λy. x y FALSE
- **OR** = λx.λy. x TRUE y
- **XOR** = λx.λy. x (y FALSE TRUE) y
- **IMPLIES** = λx.λy. x y TRUE

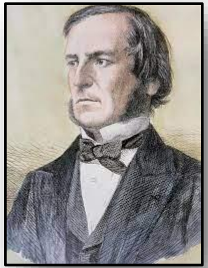
Complex expressions (ex: recursion – Haskell):

y = λf.(λx f(x x)) (λx.f(x x))

In Python (logical functions):

```
def true(x, y):
    return x

def false(x, y):
    return y

def logicalNot(x):
    return x(false, true)

def logicalAnd(x, y):
    return x(y, false)

def logicalOr(x, y):
    return x(true, y)
```

```
exp1 = logicalNot(false)
print(exp1)
>> <function true>

exp2 = logicalOr(true, false)
print(exp2.__name__)
>> <function true>
```



https://www.youtube.com/watch?v=eis11j_iGMs

In Go (logical functions):

```go
package main

import "fmt"

const True = 1
const False = 0

type funcType func(x, y int) int

func TRUE(x, y int) int {
    return x
}

func FALSE(x, y int) int {
    return y
}

func callFunc(f funcType, x, y int) int {
    return f(x, y)
}

func callDefault(f funcType) int {
    return callFunc(f, True, False)
}
```

```go
func NOT(x funcType) int {
    return callFunc(x, False, True)
}

func AND(x, y funcType) int {
    return callFunc(x, callDefault(y), callDefault(FALSE))
}

func OR(x, y funcType) int {
    return callFunc(x, callDefault(TRUE), callDefault(y))
}

func XOR(x, y funcType) int {
    return callFunc(x, callFunc(y, callDefault(FALSE),
callDefault(TRUE)), callDefault(y))
}

func IMP(x, y funcType) int {
    return callFunc(x, callDefault(y), callDefault(TRUE))
}
```

In Go:

```go
func boolean() {
    var t, f funcType
    var T, F, n, a, o, x, i int
    fmt.Println("Constants .................")
    t = TRUE
    T = callDefault(t)
    fmt.Printf("TRUE: %d\n", T)
    f = FALSE
    F = callDefault(f)
    fmt.Printf("FALSE: %d\n", F)
    fmt.Println("Not .......................")
    n = NOT(t)
    fmt.Printf("NOT TRUE: %d\n", n)
    n = NOT(f)
    fmt.Printf("NOT FALSE: %d\n", n)
    fmt.Println("And .......................")
    a = AND(t, t)
    fmt.Printf("TRUE AND TRUE: %d\n", a)
    a = AND(t, f)
    fmt.Printf("TRUE AND FALSE: %d\n", a)
    a = AND(f, t)
    fmt.Printf("FALSE AND TRUE: %d\n", a)
    a = AND(f, f)
    fmt.Printf("FALSE AND FALSE: %d\n", a)
    fmt.Println("Or ........................")
    o = OR(t, t)
    fmt.Printf("TRUE OR TRUE: %d\n", o)
    o = OR(t, f)
    fmt.Printf("TRUE OR FALSE: %d\n", o)
    o = OR(f, t)
    fmt.Printf("FALSE OR TRUE: %d\n", o)
    o = OR(f, f)
    fmt.Printf("FALSE OR FALSE: %d\n", o)
    fmt.Println("Xor .......................")
    x = XOR(t, t)
    fmt.Printf("TRUE XOR TRUE: %d\n", x)
    x = XOR(t, f)
    fmt.Printf("TRUE XOR FALSE: %d\n", x)
    x = XOR(f, t)
    fmt.Printf("FALSE XOR TRUE: %d\n", x)
    x = XOR(f, f)
    fmt.Printf("FALSE XOR FALSE: %d\n", x)
    fmt.Println("Imp .......................")
    i = IMP(t, t)
    fmt.Printf("TRUE IMP TRUE: %d\n", i)
    i = IMP(t, f)
    fmt.Printf("TRUE IMP FALSE: %d\n", i)
    i = IMP(f, t)
    fmt.Printf("FALSE IMP TRUE: %d\n", i)
    i = IMP(f, f)
    fmt.Printf("FALSE IMP FALSE: %d\n", i)
}
```

Compilers – Art. 5

Thank you for your attention