Course 11

Push-Down Automata (PDA)

Intuitive Model

Definition

- A push-down automaton (APD) is a 7-tuple $M = (Q,\boldsymbol{\Sigma},\boldsymbol{\Gamma},\boldsymbol{\delta},q_0,Z_0,F)$ where:
  - Q – finite set of states
  - $\boldsymbol{\Sigma}$ - alphabet (finite set of input symbols)
  - $\boldsymbol{\Gamma}$ – stack alphabet (finite set of stack symbols)
  - $\boldsymbol{\delta} : Q \times (\boldsymbol{\Sigma} \cup \{\boldsymbol{\varepsilon}\}) \times \boldsymbol{\Gamma} \rightarrow \mathcal{P}(Q \times \boldsymbol{\Gamma}^*)$ –transition function
  - $q_0 \in Q$ – initial state
  - $Z_0 \in \boldsymbol{\Gamma}$ – initial stack symbol
  - $F \subseteq Q$ – set of final states

Push-down automaton

Transition is determined by:

- Current state
- Current input symbol
- Head of stack

Reading head -> input band:

- Read symbol
- No action

Stack:

- Zero symbols => pop
- One symbol => push
- Several symboluri => repeat push

Configurations and transition / moves

- Configuration:

$$(\boldsymbol{q}, \boldsymbol{x}, \boldsymbol{\alpha}) \in Q \times \boldsymbol{\Sigma}^* \times \boldsymbol{\Gamma}^*$$

where:

  - PDA is in state **q**
  - Input band contains **x**
  - Head of stack is $\boldsymbol{\alpha}$
- Initial configuration (q_0, w, Z_0)

Configurations and moves(cont.)

- Moves between configurations:

p,q $\in Q$, a$\in \boldsymbol{\Sigma}$, Z $\in \boldsymbol{\Gamma}$, w $\in \boldsymbol{\Sigma}^*$,$\boldsymbol{\alpha}$,$\boldsymbol{\gamma}$ $\in \boldsymbol{\Gamma}^*$

$(q,aw,Z\boldsymbol{\alpha}) \vdash (p,w,\boldsymbol{\gamma}\boldsymbol{\alpha})$ iff $\boldsymbol{\delta}(q,a,Z) \ni (p,\boldsymbol{\gamma})$

$(q,aw,Z\boldsymbol{\alpha}) \vdash (p,aw,\boldsymbol{\gamma}\boldsymbol{\alpha})$ iff $\boldsymbol{\delta}(q,\boldsymbol{\varepsilon},Z) \ni (p,\boldsymbol{\gamma})$

($\boldsymbol{\varepsilon}$-move)

- $\overset{k}{\vdash}$, $\overset{+}{\vdash}$, $\overset{*}{\vdash}$

Language accepted by PDA

- Empty stack principle:

$L_{\varepsilon}(M) = \{w | w \in \Sigma^*, (q_0,w,Z_0) \vdash^* (q,\varepsilon,\varepsilon), q \in Q\}$

- Final state principle:

$L_f(M) = \{w | w \in \Sigma^*, (q_0,w,Z_0) \vdash^* (q_f,\varepsilon,\gamma), q_f \in F\}$

Representations

- Enumerate
- Table
- Graphic

Construct PDA

- $L = \{0^n1^n \mid n \geq 1\}$
- States, stack, moves?

1. States:
   - Initial state: q_0 – beginning and process symbols '0'
   - When first symbol '1' is found – move to new state => q_1
   - Final: final state q_2
2. Stack:
   - Z_0 – initial symbol
   - X – to count symbols:
     - When reading a symbol '0' – push X in stack
     - When reading a symbol '1' – pop X from stack

Exemple 1 (enumerate)

$M = (\{q_0,q_1,q_2\}, \{0,1\}, \{Z_0,X\},\boldsymbol{\delta},q_0,Z_0,\{q_2\})$

$\boldsymbol{\delta}(q_0,0,Z_0) = (q_0,XZ_0)$

$\boldsymbol{\delta}(q_0,0,X) = (q_0,XX)$

$\boldsymbol{\delta}(q_0,1,X) = (q_1,\boldsymbol{\varepsilon})$

$\boldsymbol{\delta}(q_1,1,X) = (q_1,\boldsymbol{\varepsilon})$

$\cancel{\boldsymbol{\delta}(q_1,\boldsymbol{\varepsilon},Z_0) = (q_2,Z_0)}$

$\boxed{\boldsymbol{\delta}(q_1,\boldsymbol{\varepsilon},Z_0) = (q_1, \boldsymbol{\varepsilon})}$

Empty stack

$\vdash (q_1, \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon})$

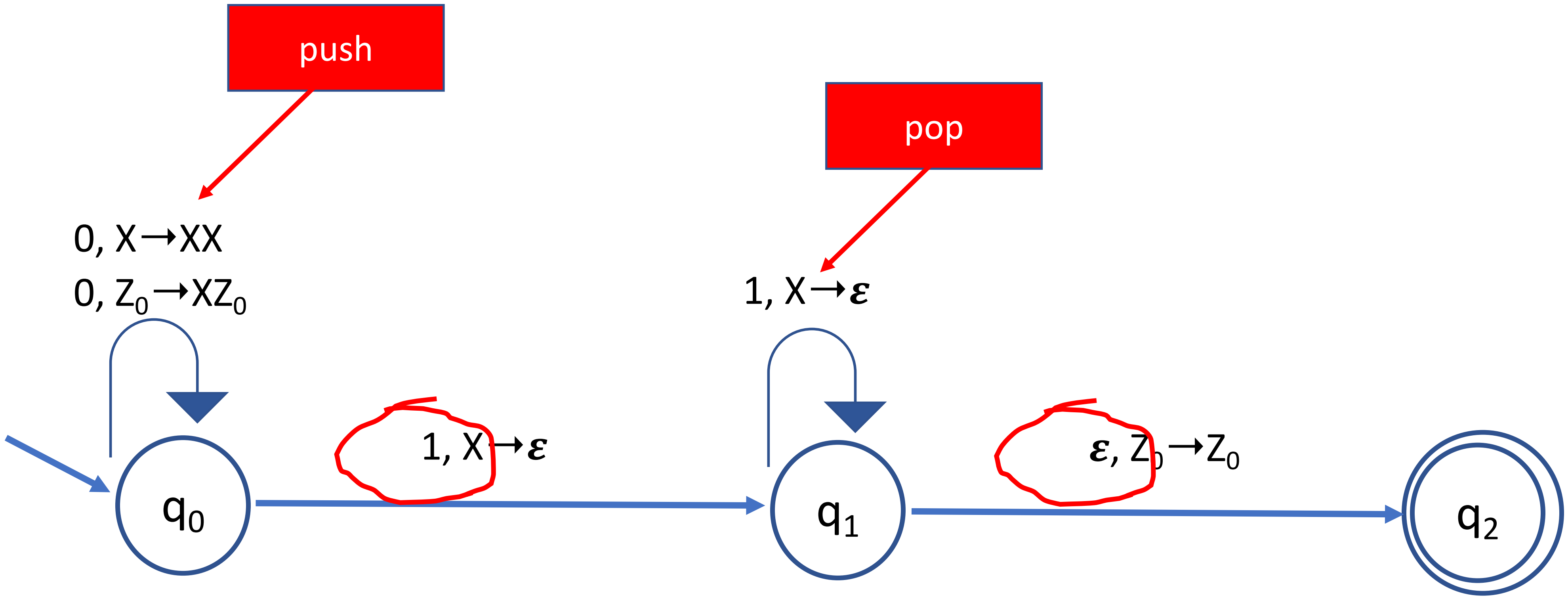
$(q_0,0011,Z_0) \vdash (q_0,011,XZ_0) \vdash (q_0,11,XXZ_0) \vdash (q_1,1,XZ_0) \vdash (q_1, \boldsymbol{\varepsilon}, Z_0) \vdash (q_2, \boldsymbol{\varepsilon}, Z_0)$

Final state

Exemple 1 (table)

		0	1	ε
q_0	Z_0	q_0,XZ_0		
	X	q_0,XX	q_1,ε	
q_1	Z_0			q_2,Z_0
	X		q_1,ε	
q_2	Z_0			
	X			

Exemple 1 (graphic)



Properties

Theorem 1: For any PDA M, there exists a PDA M' such that

$$L_{\varepsilon}(M) = L_f(M')$$

Theorem 2: For any PDA M, there exists a context free grammar such that

$$L_{\varepsilon}(M) = L(G)$$

Theorem 3: For any context free grammar there exists a PDA M such that

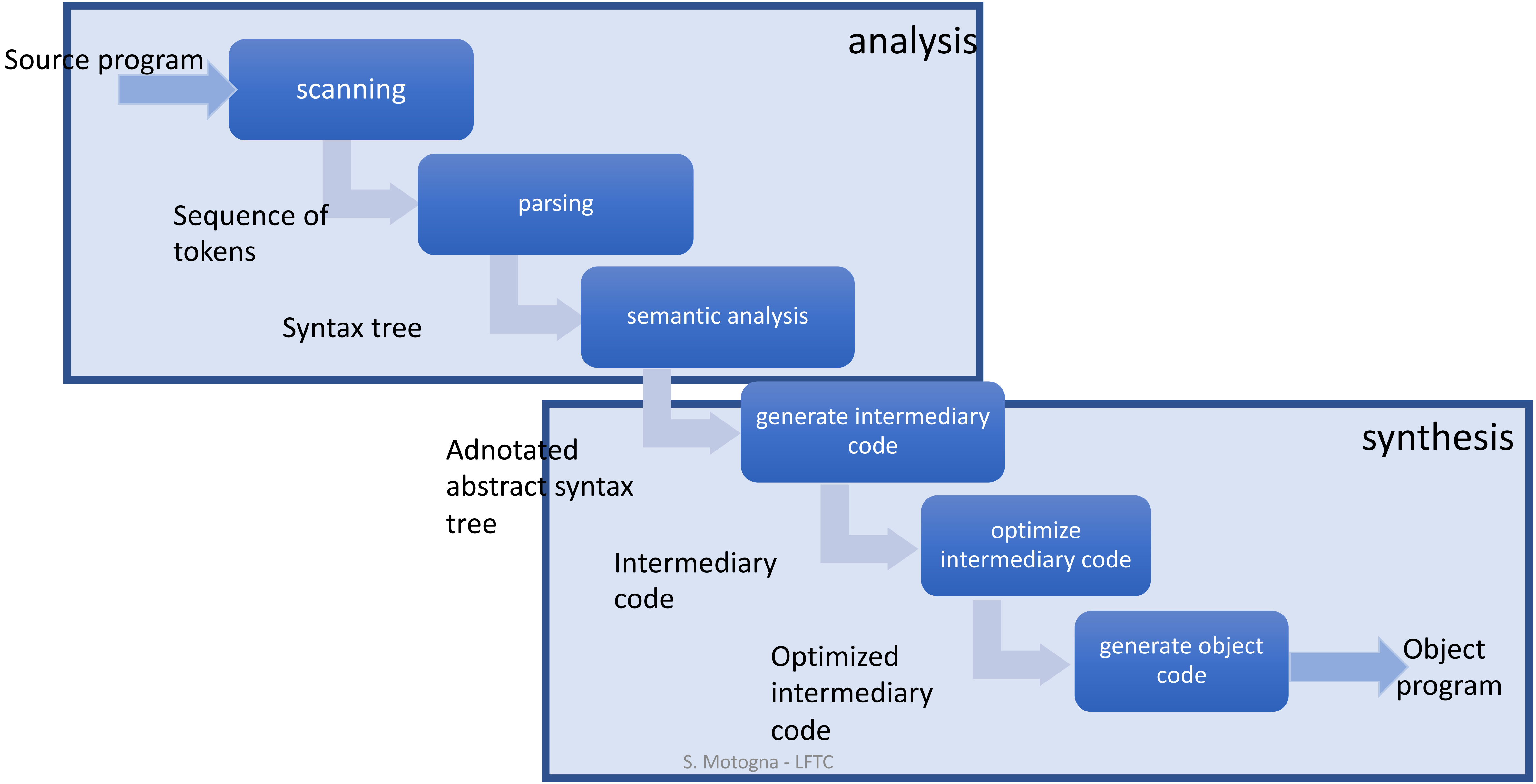
$$L(G) = L_{\varepsilon}(M)$$

HW

- Parser:
  - Descendent recursive
  - LL(1)
  - LR(0), SLR, LR(1)

Corresponding PDA

Structure of compiler



Semantic analysis

- Parsing – result: syntax tree (ST)
- Simplification: abstract syntax tree (AST)
- Adnotated abstract syntax tree (AAST)
  - Attach semantic info in tree nodes

Semantic analysis

- Attach meanings to syntactical constructions of a program
- What:
  - Identifiers -> values / how to be evaluated
  - Statements -> how to be executed
  - Declaration -> determine space to be allocated and location to be stored
- Examples:
  - Type checkings
  - Verify properties
- How:
  - **Attribute grammars**
  - Manual methods

Attribute grammar

- Syntactical constructions (nonterminals) – attributes

$$\forall\, X \;\in N \;\cup\; \Sigma : A(X)$$

- Productions – rules to compute/ evaluate attributes

$$\forall\, p \;\in P : R(p)$$

Definition

AG = (G,A,R) is called ***attribute grammar*** where:

- G = (N,$\boldsymbol{\Sigma}$,P,S) is a context free grammar
- A = {A(X) | X $\in$N U $\boldsymbol{\Sigma}$} – is a finite set of attributes
- R = {R(p) | p $\in$P} – is a finite set of rules to compute/evaluate attributes

Example 1

- G = ({N,B},{0,1}, P, N}

P: $N_1 \to N_2 B$

$N \to B$

$B \to 0$

$B \to 1$

$N_1.v = 2 * N_2.v + B.v$

$N.v = B.v$

$B.v = 0$

$B.v = 1$

Attribute – value of number = **v**

- **Synthetized attribute: A(lhp) depends on rhp**
- **Inherited attribute: A(rhp) depends on lhp**

Evaluate attributes

- Traverse the tree: can be an infinite cycle

- Special classes of AG:
  - L-attribute grammars: for any node the depending attributes are on the “*left*”;
    - can be evaluated in one left-to-right traversal of syntax tree
    - Incorporated in top-down parser (LL(1))
  - S-attribute grammars: synthetized attributes
    - Incorporated in bottom-up parser (LR)

Steps

- What? - decide what you want to compute (type, value, etc.)
- Decide attributes:
  - How many
  - Which attribute is defined for which symbol
- Attach evaluation rules:
  - For each production – which rule/rules

Example 2 (L-attribute grammar)

Decl -> DeclTip ListId

ListId -> Id

ListId -> ListId, Id

ListId.type = DeclTip.type

Id.type = ListId.type

$ListId_2$.type = $ListId_1$.type

Id.type = $ListId_1$.type

Attribute – type

int i,j

Example 3 (S-attribute grammar)

ListDecl -> ListDecl; Decl

ListDecl -> Decl

Decl -> Type ListId

Type -> int

Type -> long

ListId -> Id

ListId -> ListId, Id

$ListDecl_1.dim = ListDecl_2.dim + Decl.dim$

$ListDecl.dim = Decl.dim$

$Decl.dim = Type.dim * ListId.no$

$Type.dim = 4$

$Type.dim = 8$

$ListId.no = 1$

$ListId_1.no = ListId_2.no + 1$

Attributes – dim + no – **for which symbols**

int i,j; long k

Proposed problems (HW):

1) Define an attribute grammar for arithmetic expressions
2) Define an attribute grammar for logical expressions
3) Define an attribute grammar for if statement