# Recursive Descent and Pushdown Automata

CSCI 3136: Principles of Programming Languages

# Agenda

- Recursive Descent
- Pushdown Automata (PDA)
- Deterministic Pushdown Automata

# LL(1) Parser Implementation

- Two efficient approaches:
  - Recursive Descent
  - Deterministic Pushdown Automata (DPDA)

- Recursive Descent is easier to understand and implement.

# Recursive Descent

- Idea: For each variable X, write a procedure: parse_X()
- Where is the stack?
- Where do we start?
  - parse_S()
- How do we know if the syntax is correct?
  - No syntax error

```
parse_X:
    t = peek_next_token()
    select X            based on t
    for each    i          :
        if    i == Y1    V:
            parse_Y1()
        elseif    i == Y2    V:
            parse_Y2()
        ...
```

# Example

Grammar

- S → Add | Sub | Mul
- S → Div | Neg | Val
- Add → + S S
- Sub → − S S
- Mul → X S S
- Div → / S S
- Neg → neg S

```
parse_S:
  t = peek_at_token()

  select S          based
on t
  for each   i        :
    if   i ==Add     V:
      parse_Add()
    elseif   i ==Sub
   V:
      parse_Sub()

    ...
    elseif   i ==Val
```

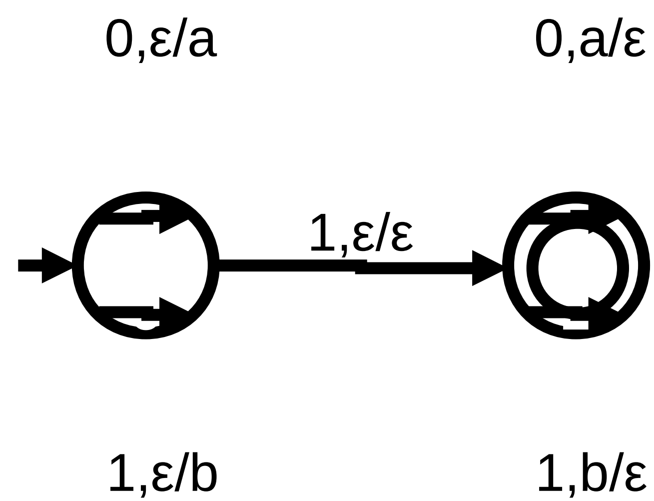
# Push Down Automata

- We proved that
  - L can be parsed by a DFA if and only if it is regular
  - Some context free languages, including most programming languages, are not regular.
  - DFAs are not powerful enough to parse context free languages.
- We need a more powerful automata!
- A *push-down automaton* (PDA) is an NFA with a stack.
  - We can use this model to reason and derive properties of context free languages.

# Example: PDA for L = {σ1σr| σE{0,1}X}

- States: Q ={q0,q1}
- Input alphabet: Σ ={0,1}
- Stack alphabet: Γ ={a,b}

| State | Input | Pop | New State | Push |
|---|---|---|---|---|
| q0 | 0 | ε | q0 | a |
| q0 | 1 | ε | q0 | b |
| q0 | 1 | ε | q0 | ε |
| q1 | 0 | a | q1 | ε |
| q1 | 1 | b | q1 | ε |




*Problem*: In this language we do not know when to transition to q1.

# Formal Definition of a PDA

- A pushdown automata (PDA) M is a 7-Tuple

  $M = (Q,\Sigma,\Gamma,\delta,q0,S,F)$

  - $Q$ is the set of states
  - $\Sigma$ is the input alphabet
  - $\Gamma$ is the stack alphabet
  - $\delta$ is the transition function: $\delta: Q \times \Sigma \times \Gamma \rightarrow 2Q \times \Gamma$
  - $q0$ is the start state
  - $S$ the initial symbol on the stack
  - $F$ is the set of final states
- There are two different modes of acceptance

# Modes of Acceptance for PDAs

- **Empty Stack** : Accept if and only if it is possible to reach a configuration where
  - The input has been consumed completely
  - The stack is empty
  - State does not matter
- **Final state :** Accept if and only if it is possible to reach a configuration where
  - The input has been consumed completely
  - The current state is an accepting state
  - Stack contents do not matter

# Facts about PDAs

- A language is a CFL if and only if it can be recognized by a PDA.
- A *deterministic PDA* (DPDA) is a PDA that has only one possible transition in any configuration
- L can be recognized by a DPDA if and only if it is LL(k) or LR(k)
- Not all L are LL(k) or LR(k),

  e.g. Languages of palindromes.

# Deterministic Pushdown Automata

- Definition: A deterministic pushdown automata (DPDA) M is a 7-Tuple:

  $M = (Q,\Sigma,\Gamma,\delta,q0,S,F$

  - $Q$ is the set of states
  - $\Sigma$ is the input alphabet
  - $\Gamma$ is the stack alphabet
  - $\delta$ is the transition function: $\delta: Q\times\Sigma\times\Gamma \rightarrow Q\times\Gamma$
  - $q0$ is the start state
  - $S$ the initial symbol on the stack
  - $F$ is the set of final states

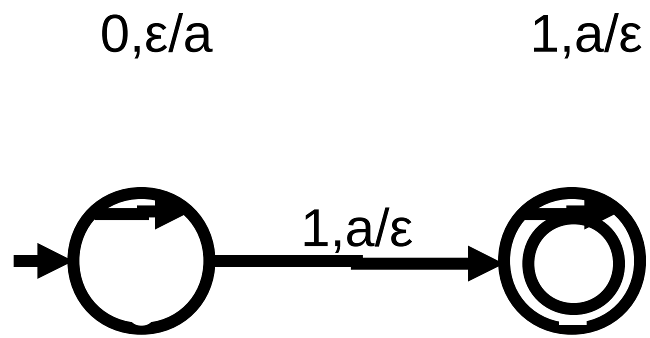
# Example: DPDA for L = {0n1n| n>0}

- States: Q ={q0,q1}
- Input alphabet: Σ ={0,1}
- Stack alphabet: Γ ={a}

| State | Input | Pop | New State | Push |
|---|---|---|---|---|
| q0 | 0 | ε | q0 | a |
| q0 | 1 | a | q1 | ε |
| q1 | 1 | a | q1 | ε |

={q1}∈Q

- Transition function: δ



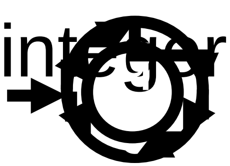
# Parsing with a DPDA

- How do we build a DPDA to implement LL(1) grammar?
- Idea:
  - Input: token stream
  - Σ is the alphabet of tokens.
  - Transitions are based on:
    - Tokens read, matching predictor sets for given productions
    - Symbols on the stack
  - The stack contains partial sentential forms
  - Rewriting involves popping off a nonterminal and

# Example: Our Favourite Grammar

Grammar

- S → + S S
- S → − S S
- S → X S S
- S → / S S
- S → neg S
- S → integer




- Q = {q0}
- Σ = {+,−,X,/,neg,int}
- Γ = {S}
- q0: q0EQ
- Stack = SEΓ
- F ={q0}öQ
- δ:

| State | Input | Pop | Next | Push |
|---|---|---|---|---|
| q0 | + | S | q0 | SS |
| q0 | - | S | q0 | SS |
| q0 | * | S | q0 | SS |
| q0 | / | S | q0 | SS |
| q0 | neg | S | q0 | S |
| q0 | int | S | q0 | ε |

# Implementing DPDAs

Implementation Options

- **Using nested case statements**
  - Level 1: Branch on current state
  - Level 2: Branch on current input symbol
  - Level 3: Branch on current stack symbol
- **Similar to recursive-descent parsing**
  - Instead of recursion, maintain the stack explicitly
- **Table-driven**
  - 3-D table mapping (state, input symbol, stack symbol) triples to strings to be pushed onto the stack.