Recursive Descent and Pushdown Automata

CSCI 3136: Principles of Programming Languages

Agenda

- Announcements
  - Assignment 4 is out and due June 19.
  - Midterm next week: June 19, 10 – 11:30 in **CHEB 170.**
- Readings:
  - Today: 2.3.0, 2.3.1
  - Note: I recommend using alternative texts for this part of the course:
  - E.g., Hopcorft et al, "Introduction to Automata Theory"
- Lecture Contents
  - 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.

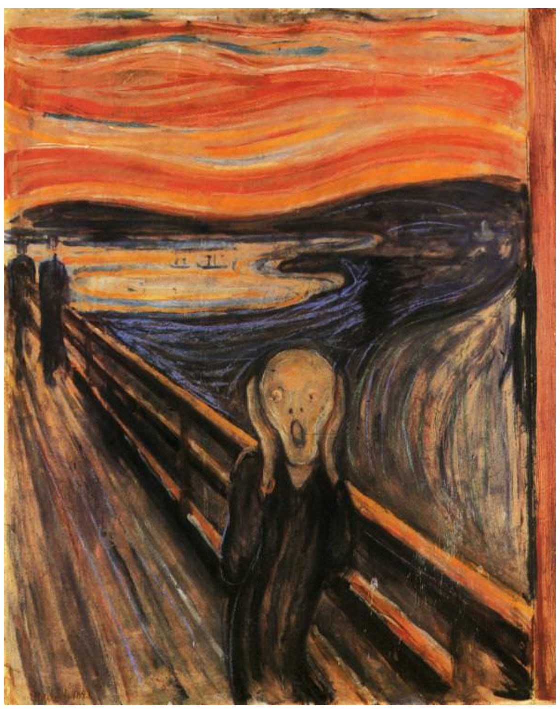
LL(1) Parser Implementation

- 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()

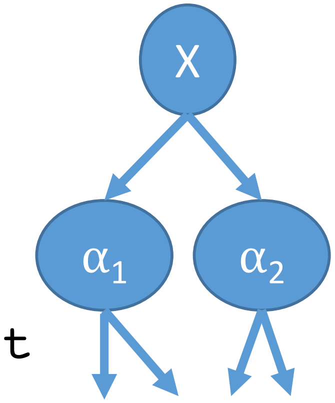


Use PREDICT table

```
parse_X:
  n = TreeNode('X')
  t = peek_next_token()
  select X → α based on t
  for each α_i ∈ α :
    if α_i == Y_1 ∈ V:
      n.addChild(parse_Y_1())
    elseif α_i == Y_2 ∈ V:
      n.addChild(parse_Y_2())

    ...

    elseif α_i == Y_k ∈ V:
      n.addChild(parse_Y_k())
    elseif α_i == t:
      n.addChild(remove_token())
      t = peek_next_token()
    else:
      syntax error
  return n
```



Example

Grammar

- S → Add
- S → Sub
- S → Mul
- S → Div
- S → Val
- Add → + S S
- Sub → – S S
- Mul → * S S
- Div → / S S
- Val → integer

```
parse_S:
  t = peek_at_token()
  select S → α based on t
  for each αi ∈ α :
    if αi == Add ∈ V:
      parse_Add()
    elseif αi == Sub ∈ V:
      parse_Sub()
    elseif αi == Mul ∈ V:
      parse_Mul()
    …
    elseif αi == Val ∈ V:
      parse_Val()
    else:
      syntax error
```

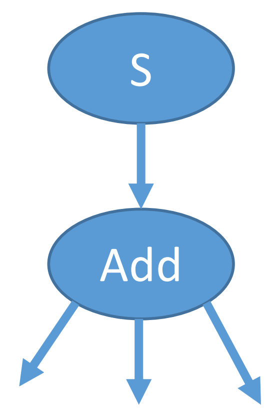
Example
More Concrete

Grammar

- **S → Add**
- **S → Sub**
- **S → Mul**
- **S → Div**
- **S → Val**
- Add → + S S
- Sub → – S S
- Mul → ∗ S S
- Div → / S S
- Val → integer

```
parse_S:
  t = peek_at_token()
  if t == '+' :
    n = parse_Add()
  elseif t == '-':
    n = parse_Sub()
  elseif t == '*':
    n = parse_Mul()
  elseif t == '/':
    n = parse_Div()
  elseif t == 'neg':
    n = parse_Neg()
  elseif t is integer:
    n = parse_Val()
  else:
    syntax error

  return TreeNode("S", n)
```

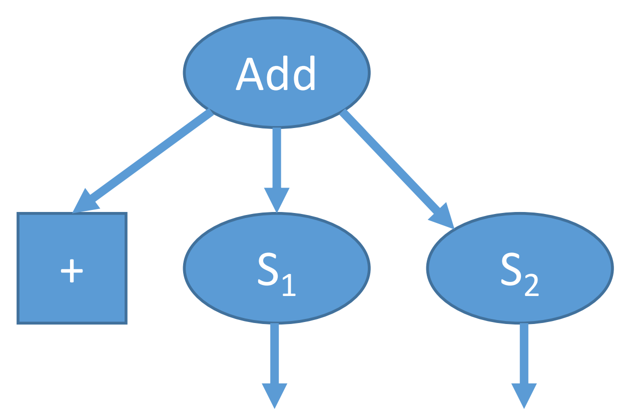


Example
More Concrete

Grammar

- S → Add
- S → Sub
- S → Mul
- S → Div
- S → Val
- **Add → + S S**
- Sub → – S S
- Mul → ∗ S S
- Div → / S S
- Val → integer

```
parse_Add:
  t = peek_at_token()
  if t != '+':
    # never happens
    syntax error
  remove_token()
  s1 = parse_S()
  s2 = parse_S()
  return TreeNode("Add",t,s1,s2)
```



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.

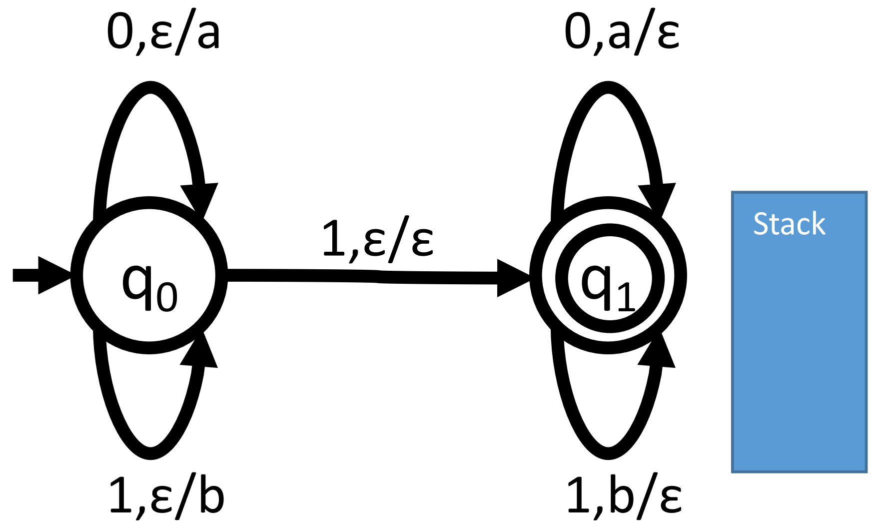
Example: PDA for $L = \{\sigma 1 \sigma^r \mid \sigma \in \{0,1\}^*\}$

- States: $Q = \{q_0, q_1\}$
- Input alphabet: $\Sigma = \{0,1\}$
- Stack alphabet: $\Gamma = \{a,b\}$
- Start state: $q_0 \in Q$
- Initial stack: $S = \varepsilon \in \Gamma$
- Final states: $F = \{q_1\} \subseteq Q$
- Transition function: δ

State	Input	Pop	New State	Push
q_0	0	ε	q_0	a
q_0	1	ε	q_0	b
q_0	1	ε	q_1	ε
q_1	0	a	q_1	ε
q_1	1	b	q_1	ε

Transition: **a**,*x*/*y* means
- Read input symbol **a**
- Pop *x* of the stack
- Push *y* on the stack

Transition to next state
ε means empty (nothing)



Problem: In this language we do not know when to transition to q_1.

Formal Definition of a PDA

- A pushdown automata (PDA) M is a 7-Tuple
  $M = (Q,\Sigma,\Gamma,\delta,q_0,S,F)$
  - Q is the set of states
  - Σ is the input alphabet
  - Γ is the stack alphabet
  - δ is the transition function: $\delta: Q\times\Sigma\times\Gamma \rightarrow 2^Q\times\Gamma$
  - q_0 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 we can adopt.

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
- *The two modes are equivalent!*
- *We can convert one kind of PDA to the other!*

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,Σ,Γ,δ,q0,S,F

  - Q is the set of states
  - Σ is the input alphabet
  - Γ is the stack alphabet
  - δ is the transition function: $\delta: Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma$
  - q_0 is the start state
  - S the initial symbol on the stack
  - F is the set of final states

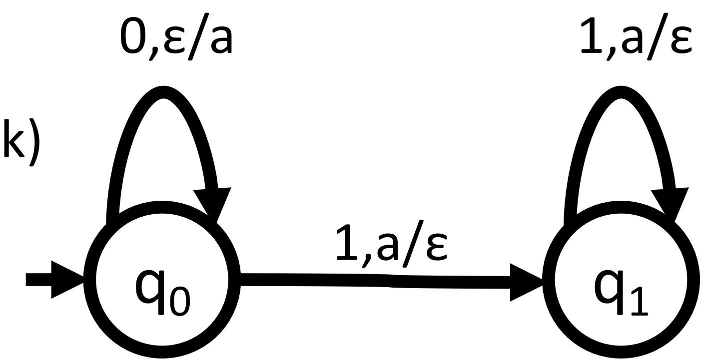
This is the only difference between PDAs and DPDAs

- DPDAs also accept by empty stack or by final state

Example: DPDA for $L = \{0^n1^n | n>0\}$

- States: $Q = \{q_0, q_1\}$
- Input alphabet: $\Sigma = \{0,1\}$
- Stack alphabet: $\Gamma = \{a\}$
- Start state: $q_0 \in Q$
- Initial stack: $S = \varepsilon \in \Gamma$
- Final states: F = ? (Accept by empty stack)
- Transition function: δ

State	Input	Pop	New State	Push
q_0	0	ε	q_0	a
q_0	1	a	q_1	ε
q_1	1	a	q_1	ε



Examples

- Build a DPDA that recognizes
  - $L = \{0^n 1 0^n \mid n \geq 0\}$
  - $L = \{0^n 1^n 0^m 1^m \mid n, m \geq 0\}$
- Build a PDA that recognizes
  - $L = \{w \in \{0,1\}^* \mid w \text{ is a palindrome}\}$

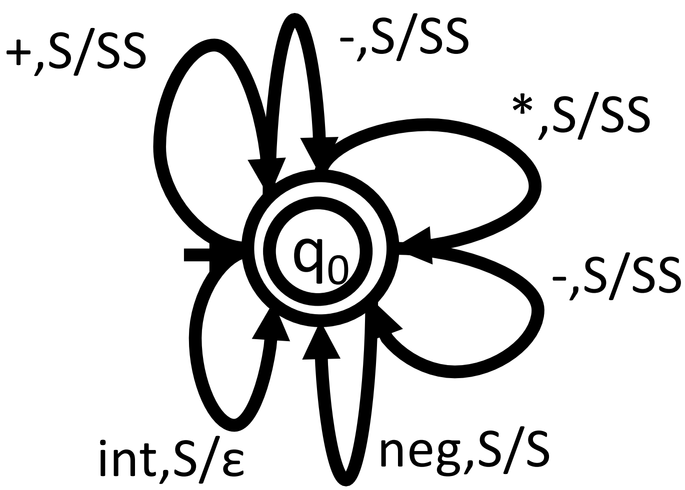
Parsing with a DPDA

- 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 pushing on the RHS of the corresponding production

Example: Our Favourite Grammar

Grammar

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



- $Q = \{q_0\}$
- $\Sigma = \{+,-,*,/,neg,int\}$
- $\Gamma = \{S\}$
- q_0: $q_0 \in Q$
- Stack = $S \in \Gamma$
- $F = \{q_0\} \subseteq Q$
- δ:

State	Input	Pop	Next	Push
q_0	+	S	q_0	SS
q_0	-	S	q_0	SS
q_0	*	S	q_0	SS
q_0	/	S	q_0	SS
q_0	neg	S	q_0	S
q_0	int	S	q_0	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.

Options for generating the parser

- Hand-coded
- Automatic generation from grammar
  - E.g.using yacc, bison, etc