Theory of Computation (CS C351)

BITS Pilani
Hyderabad Campus

Dr.R.Gururaj
CS&IS Dept.

Push-down Automaton (PDA)

Not all CFLs are recognized by FA.

L = { wcw^R / $w \in \{a,b\}^*$ }

Push-down Automaton (PDA)

M = (K, ∑ , *Γ*, Δ, s, F)

K	set of states
∑	alphabet (input)
Γ	stack symbols (alphabet)
Δ	is transition relation
s	start state ϵ *K*
F	set of final states

Transitions of PDA

((p, a, β), (q, γ)) ϵ Δ

β is a string ϵ *Γ** *(removed from top of the stack)*

γ is a string ϵ *Γ** *(pushed on to the stack)*

When a= e (null) the input is not consulted.

((p, u, e), (q, a))   // pushing a on to the stack
((p, u, a), (q, e))   // popping a from the stack

Like in FA, the portion of the string already read has no effect on the process.

Configuration of PDA

The Configuration of a PDA is a n element of $(K \times \Sigma^* \times \Gamma^*)$

The language accepted by M is denoted by L(M)

The correspondence between FA and PDA

FA= M= $(K, \Sigma, \Delta, s, F)$

PDA= M'= $(K, \Sigma, \Gamma, \Delta', s, F)$

$\Delta' = \{((p, u, e), (q, e)) : (p, u, q) \in \Delta\}$

Ex: For the L generated by RE - (a*ba)

Rules for constructing PDA from CFG

$\Delta =$ (1) ((p,e,e), (q,S))

(2) ((q,e,A), (q, x))

for each rule A→X in R

(3) ((q,a,a), (q,e))

for each a $\in \Sigma$

Top-down Parsing

For the Language L= $a^n b^n$

$M_1 = (\ \{p, q\},\ \{a,b\},\ \{a,b,S\},\ \Delta_1,\ R,\ \{q\})$

$S \rightarrow aSb;\ S \rightarrow e$

$\Delta_1 = \{\ ((p,e,e), (q,S)),$

$((q,e,S), (q,aSb)),$

$((q,e,S), (q,e)),$

$((q,a,a), (q,e)),$

$((q,b,b), (q,e))\ \}$

This is non-deterministic because

2 and 3 are compatible.

PDA and CFL

- If a language is accepted by a PDA, it is a CFL.
- CFL are closed under union, concatenation and Kleene star.
- The intersection of a CFL and RL is always CFL.
- CFLs are not closed under intersection and complementation.

Determinism and Parsing (Section 3.7)

CFGs are extensively used in modeling the syntax of programming languages.

A compiler for such a programming language must then embody a parser that is an algorithm to determine whether a given string is in the language generated by a given CFG, if so, to construct a parse tree of the string.

The compiler then translate this parse tree into a program in a more basic language , such as assembly language.

Many approaches to the parsing problem have been developed by compiler designers over the past four decades.

The most successful ones among them are rooted in the idea of a PDA.

The equivalence of CFG and PDA should be put to work.

A PDA is not of immediate practical use because it is a nondeterministic device.

The question is can we always make PDA to work deterministically, as we were able to do in the case of FA.

We shall see that there are some CFLs that cannot be accepted by DPDA.
This is bit disappointing to us.
Nevertheless it turns out that for most programming languages, one can construct DPDA.
Here we look at some heuristic rules for constructing DPDA.
The class of DCFL is closed under complement.
The class of DCFL is contained in CFL.

Deterministic CFLs

A PDA M is deterministic if for each configuration there is at most one configuration succeeding it in a computation.

We call two strings consistent if the first is the prefix of the second or vice versa.

We call two transitions

$((p, a, \beta), (q, \gamma))$ and $((p, a', \beta'), (q', \gamma'))$ compatible if a and a' are consistent and β and β' are consistent, in other words if there is a situation in which both transitions are applicable.

Then M is compatible if it has no two distinct compatible transitions. Ex.

Deterministic PDAs

Deterministic CFLs are essentially those that are accepted by deterministic PDA.

But we have to modify he acceptance criteria slightly.

A language is said to be deterministic CF if it is recognized by a DPDA and that also has a capability to of sensing the end of the input string.

Formally, we call a language L subset of Σ^* deterministic Context-Free if $L\$=L(M)$ for some deterministic PDA M.
Here \$ is a new symbol not in Σ, which is appended to the input string for the purpose of marking its end.

Every deterministic CFL is a CFL.

If we do not adopt this new convention then many CFLs which are actually deterministic turn out to be non-deterministic. Ex. $a^* \cup (a^n b^n : n \geq 1)$

The class of deterministic CFLs is closed under complement.

 The class of deterministic CFLs are properly contained in the class of CFLs.

Rules for constructing PDA from CFG

$\Delta =$ (1) ((p,e,e), (q,S))

(2) ((q,e,A), (q, x))

for each rule A→X in R

(3) ((q,a,a), (q,e))

for each a $\epsilon \sum$

Top-down Parsing

For the Language L= $a^n b^n$

$M_1 = (\ \{p, q\},\ \{a,b\}, \{a,b,S\}, \Delta_1, R, \{q\})$

$S \rightarrow aSb;\ S \rightarrow e$

$\Delta_1 = \{$ ((p,e,e), (q,S)),

((q,e,S), (q,aSb)),

((q,e,S), (q,e)),

((q,a,a), (q,e)),

((q,b,b), (q,e)) $\}$

This is non-deterministic because

2 and 3 are compatible.

For the Language L= $a^n b^n$

$\Delta = \{$ ((p,e,e), (q,S)),
((q,a,e), (q_a, e)),
((q_a,e,a), (q,e)),
((q,b,e), (q_b, e)),
((q_b,e,b), (q,e)),
((q,\$,e), ($q_\$$, e)),
((q_a,e,S), (q_a,aSb)),
((q_b,e,S), (q_b,e)) $\}$

This is deterministic.

This is achieved by *lookahead*. by consuming an input symbol ahead of time and incorporating that information into its state.

The devices like ones we have seen in the previous example Which correctly decide whether a string belongs in a CFL and in the case of positive answer, produce the corresponding *parse tree* , are called as *parsers*

The PDA we have seen is a *top-down parser* because tracing its operation at the steps where non-terminals are replaced on the stack, constructs a parse-tree in top-down order or left-to-right order.

Naturally not all CFLs have deterministic acceptors that can be derived from the standard nondeterministic one via the lookahead idea.

For certain deterministic CFLs, lookahead of just one symbol may not be sufficient to resolve all the uncertainties.

Some languages are not directly amenable to parsing by lookahead.

$E \rightarrow T$

$E \rightarrow E+T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

$F \rightarrow id(E)$

Construct a NDPDA using standard rules.

Δ = (1) ((p,e,e), (q,S))

(2) ((q,e,A), (q, x))

for each rule A→x in R

(3) ((q,a,a), (q,e))

for each a ϵ ∑

Δ = { ((p,e,e), (q,E)),	1
((q,e,E), (q, E+T)),	2
((q,e,E), (q,T)),	3
((q,e,T), (q, T*F)),	4
((q,e,T), (q,F)),	5
((q,e,F), (q, (E))),	6
((q,e,F), (q,id)),	7
((q,e,F), (q,id(E)))	8

and finally ((q,a,a), (q,e)) for all a $\epsilon \sum$

}

$F \rightarrow id$

$F \rightarrow id(E)$ creates problems. Lookahead may not work here.

Whenever we have production of the form-

$A \rightarrow \alpha\beta_{1,}\ A \rightarrow \alpha\beta_{2,}\ A \rightarrow \alpha\beta_{n,}$

Replace them by

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1$

$A' \rightarrow \beta_2$ and $A' \rightarrow \beta_n$

$F \rightarrow id$
$F \rightarrow id(E)$

Replace the above with

$F \rightarrow idA$
$A \rightarrow e$
$A \rightarrow (E)$

Now we can apply lookahead without any problem.

Δ = { ((p,e,e), (q,E)),

((q,e,E), (q, E+T)),

((q,e,E), (q,T)),

((q,e,T), (q, T*F)),

((q,e,T), (q,F)),

((q,e,F), (q, (E))),

$$\left.\begin{array}{l}((q,e,F),\ (q,idA)),\\ ((q,e,A),\ (q,e))\\ ((q,e,A),\ (q,(E)))\end{array}\right\} \quad \begin{array}{l}((q,e,F),\ (q,id)),\\ ((q,e,F),\ (q,id(E)))\end{array}$$

and finally ((q,a,a), (q,e)) for all a ϵ ∑

}

E→E+T

E→T

T→T*F

T→F

F→(E)

F→idA

A→ e

A→ (E)

$E \rightarrow E+T$

Creates problems. This is known as *Left Recursion*.

Whenever we have production of the form-

$A \rightarrow A\alpha_{1,}\ A \rightarrow A\alpha_{2,}\ A \rightarrow A\alpha_{n,}$ and

$A \rightarrow \beta_{1,}\ A \rightarrow \beta_2$ and $A \rightarrow \beta_m$

Replace these by

$A \rightarrow \beta_1 A',\ A \rightarrow \beta_2 A',\ \ldots,\ A \rightarrow \beta_m A',$ and

$A' \rightarrow \alpha_1 A',\ A' \rightarrow \alpha_2 A',\ \ldots,\ A' \rightarrow \alpha_n A',$ and

$A' \rightarrow e$

E→E+T

E→T

Can be replaced by

E→ TE'

E'→ +TE'

E'→e

T→T*F

T→F

Can be replaced by

T→ FT'

T'→ *FT'

T'→e

Also understand eliminating indirect LeftRecursion.

E→T
E→E+T
T→T*F
T→F
F→(E)
F→id
F→id(E)

$\Rightarrow$

E→ TE'
E'→ +TE'
E'→e
T→ FT'
T'→ *FT'
T'→e
F→(E)
F→idA
A→ e
A→ (E)

$\Delta = \{$ ((p,e,e), (q,E)), (1)

((q,a,e), (q_a, e)), (2) for each $a \in \sum \cup \{\$\}$

((q_a ,e,a), (q,e)), (3) for each $a \in \sum$

((q_a,e,E), (q_a , TE')), (4) for each $a \in \sum \cup \{\$\}$

((q_+,e,E'), (q_+,+TE')), (5)

((q_a,e,E'), (q_a, e)), (6) for each $a \in \{), \$\}$

((q_a,e,T), (q_a, FT')), (7) for each $a \in \sum \cup \{\$\}$

((q_*,e,T'), (q_*, FT')), (8)

((q_a,e,T'), (q_a, e)), (9) for each $a \in \{+,), \$\}$

(($q_($,e,F), ($q_($, (E))), (10)

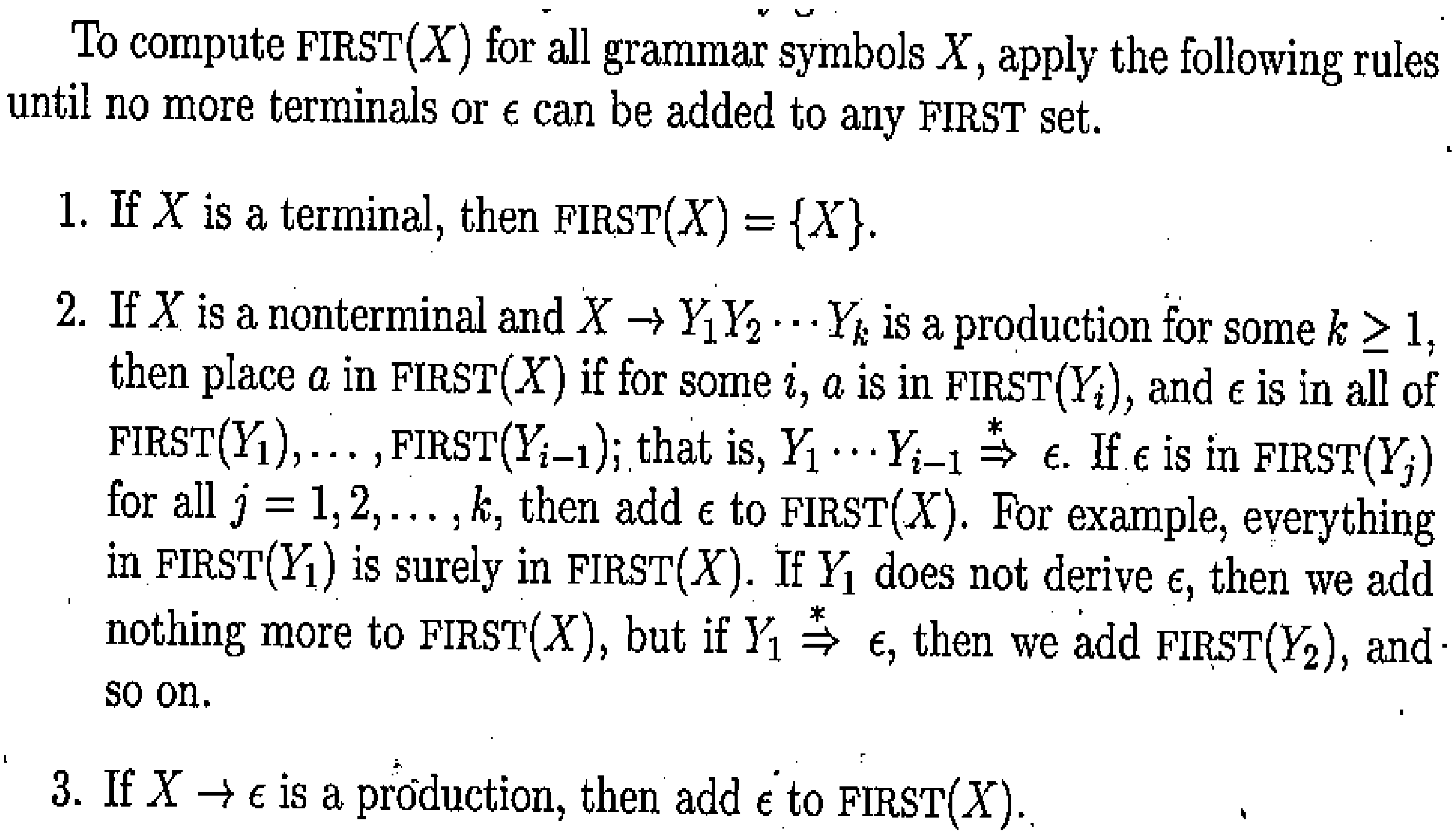
((q_{id},e,F), (q_{id},idA)), (11)

(($q_($,e,A), ($q_($,(E))) (12)

((q_a,e,A), (q_a, e)), (13) for each $a \in \{+, *,), \$\}$

$\}$

To Compute FIRST



To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then FIRST$(X) = \{X\}$.
2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST$(Y_1), \ldots,$ FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \overset{*}{\Rightarrow} \epsilon$, then we add FIRST$(Y_2)$, and so on.
3. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

Now, we can compute FIRST for any string $X_1X_2\cdots X_n$ as follows. Add to FIRST($X_1X_2\cdots X_n$) all non-ϵ symbols of FIRST(X_1). Also add the non-ϵ symbols of FIRST(X_2), if ϵ is in FIRST(X_1); the non-ϵ symbols of FIRST(X_3), if ϵ is in FIRST(X_1) and FIRST(X_2); and so on. Finally, add ϵ to FIRST($X_1X_2\cdots X_n$) if, for all i, ϵ is in FIRST(X_i).

To Compute FOLLOWS

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.

2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).

3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Other Properties of CFL

The Context-Free languages are closed under *union*, *concatenation*, and *Kleene* star.

The intersection of a Context-Free language with Regular Language is a Context Free Language.

CFLs are not closed under *intersection* or *complementation*.

There is a polynomial algorithm which, given a CFG, construct an equivalent PDA, and vice versa.

Let *G= (V, ∑, R, S)* be a Context-free Language.
The *fanout* of *G*, denoted by *Ø(G)*, is the largest number of symbols on the right-hand side of any rule in R.
A *path* in a parse-tree is a sequence of distinct nodes, each connected to the previous node by a line segment. The first node is the Root of the tree, and last node is the leaf.
The *length of the path* is the number of line segments in it.
The *height of the parse tree* is the length of the longest path in it.

The yield of any parse tree of *G* of height *h* has length at most $\emptyset(G)^h$

Pumping theorem for CFL (page-145 of T1)

Let $G= (V, \sum, R, S)$ be a Context-free Grammar.

Then $w \in L(G)$ of length greater than $Ø(G)^{|V-\Sigma|}$ can be written as $w=uvxyz$ in such a way that v and y are non empty and uv^nxy^nz is in $L(G)$ for every $n \geq 0$

Then it is CFL otherwise not.

EX: $a^nb^nc^n$

Is not CFL

Top-Down Parsing

The parser discussed so far is a top-down parser.

Because during its operations it constructs the parse tree in top-down and left-to-right fashion.

We start with the starting NT on the stack.

Every time we replace the leftmost NT in the intermediate string.

Top-down parsing mimics LMD

$E \rightarrow T$

$E \rightarrow E+T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

$F \rightarrow id(E)$

Δ = { ((p,e,e), (q,E)), 1

((q,e,E), (q, E+T)), 2

((q,e,E), (q,T)), 3

((q,e,T), (q, T*F)), 4

((q,e,T), (q,F)), 5

((q,e,F), (q, (E))), 6

((q,e,F), (q,id)), 7

((q,e,F), (q,id(E))) 8

and finally ((q,a,a), (q,e)) for all a $\epsilon \sum$

}

Bottom-up Parsing

There exist many approaches to parse CFLs.
Different methods are preferable for different languages/grammars.
An alternative to top-down parsing is Bottom-up parsing. Popular in construction of PDA.
In top-down parsing we construct tree from root to leaves. (LL)
In Bottom-up approach we start with leaves and end-up with root(starting NT). (LR)

The advantage of LR parsers :

1.LR parsers can handle a large class of context-free grammars.

2.The LR parsing method is a most general non-back tracking shift-reduce parsing method.

3.An LR parser can detect the syntax errors as soon as they can occur.

4.LR grammars can describe more languages than LL grammars.

Constructing a (Bottom-up) PDA for a given Grammar

$\Delta =$ (1) ((p,a,e), (p,a))

for each a $\epsilon \sum$

(2) ((p,e,α^R), (p, A))

for each rule A$\rightarrow\alpha$ in R

(3) ((p,e,S), (q,e))

$\Delta = \{$ ((p,a,e), (p,a)), a $\epsilon \sum$ 1

((p,e,T+E), (p, E)), 2

((p,e,T), (p, E)), 3

((p,e,F*T), (p, T)), 4

((p,e,F), (p,T)), 5

((p,e,)E( ), (p, F)), 6

((p,e,id), (p, F)), 7

((p,e,E), (q,e)) 8

$\}$

Transitions of type 1 move input symbols onto the stack. (Shift)

Transitions of type 3 pop terminal symbols off the stack when they match input symbols.

Transitions of type 2 replace the right-hand side of a rule on the stack by the corresponding left-hand side .(Reduce)
Transitions of type 3 here end a computation by moving to final state when only the start symbol remains on the stack.

Shift/Reduce conflict

Resolving Shift/Reduce conflict using precedence relation P.
Top of the stack=a
Next input symbol=b
If (a,b)ϵ P then reduce else shift.
For this precedence relation tables are constructed for a given Grammar.
It depends on the Grammar.

Reduce/Reduce conflict

Resolving Reduce/Reduce conflict using longest prefix.
This may not be applicable to all grammars.
For the Grammar we have seen in the example, the longest prefix works.

Weak Precedence Grammars

Heuristics applied for constructing deterministic top-down parsers are-
(1)Resolving Shift/Reduce conflict using precedence relation P.
(2) Resolving Reduce/Reduce conflict using longest prefix.

There are grammars for which the above will not work. The grammars for which the heuristics work are known as *weak precedence grammars*.

Most of the grammars of Computer languages can be converted to weak precedence grammars.

Summary

1. CFG
2. Regular Grammar
3. FA and RG
4. Ambiguity
5. Precedence
6. PDA
7. Top-down parsing
8. Look-ahead parsing
9. Properties of CFL pumping theorem
10. Bottom-up parsing
11. Shift/reduce ; reduce/reduce conflict
12. Weak precedence grammars.