CMPT 379
Compilers

Anoop Sarkar
http://www.cs.sfu.ca/~anoop

Parsing - Roadmap

- Parser:
  - decision procedure: builds a parse tree
- Top-down vs. bottom-up
- LL(1) – Deterministic Parsing
  - recursive-descent
  - table-driven
- LR(k) – Deterministic Parsing
  - LR(0), SLR(1), LR(1), LALR(1)
- Parsing arbitrary CFGs – Polynomial time parsing

Top-Down vs. Bottom Up

Grammar: $S \rightarrow A\ B$
$A \rightarrow c \mid \varepsilon$
$B \rightarrow cbB \mid ca$

Input String: ccbca

Top-Down/leftmost		Bottom-Up/rightmost	
$S \Rightarrow AB$	$S \rightarrow AB$	$ccbca \Leftarrow Acbca$	$A \rightarrow c$
$\Rightarrow cB$	$A \rightarrow c$	$\Leftarrow AcbB$	$B \rightarrow ca$
$\Rightarrow ccbB$	$B \rightarrow cbB$	$\Leftarrow AB$	$B \rightarrow cbB$
$\Rightarrow ccbca$	$B \rightarrow ca$	$\Leftarrow S$	$S \rightarrow AB$

Top-Down: Backtracking

$S \rightarrow A\ B$
$A \rightarrow c \mid \varepsilon$
$B \rightarrow cbB \mid ca$

True/False
$S \Rightarrow^* cbca$?

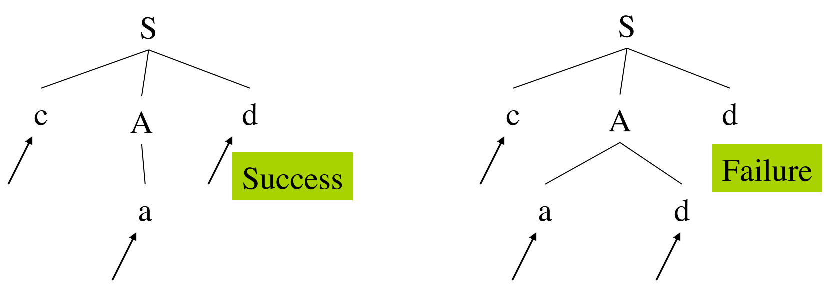
S	cbca	try $S \rightarrow AB$
AB	cbca	try $A \rightarrow c$
cB	cbca	match c
B	bca	dead-end, try $A \rightarrow \varepsilon$
εB	cbca	try $B \rightarrow cbB$
cbB	cbca	match c
bB	bca	match b
B	ca	try $B \rightarrow cbB$
cbB	ca	match c
bB	a	dead-end, try $B \rightarrow ca$
ca	ca	match c
a	a	match a, Done!

Backtracking

S → cAd | c
A → a | ad

Input: cad

S → cAd | c
A → ad | a



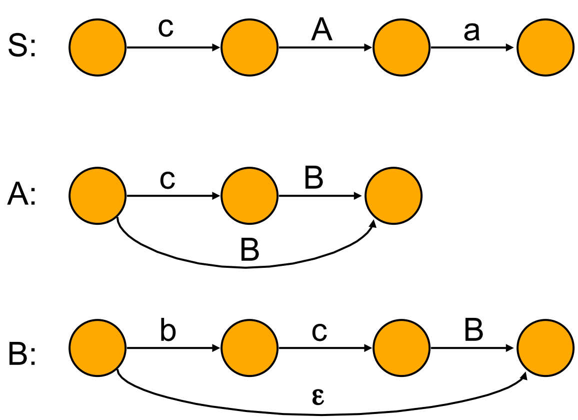
For some grammars, rule ordering is crucial for backtracking parsers, e.g S → aSa, S → aa

Transition Diagram

S → cAa

A → cB | B

B → bcB | ε



Predictive Top-Down Parser

- Knows which production to choose based on single lookahead symbol
- Need LL(1) grammars
  - First L: reads input Left to right
  - Second L: produce Leftmost derivation
  - 1 : one symbol of lookahead
- Can't have left-recursion
- Must be left-factored (no left-factors)
- Not all grammars can be made LL(1)

Leftmost derivation for **id + id * id**

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow - E$
$E \rightarrow \textbf{id}$

$E \Rightarrow E + E$
$\Rightarrow \textbf{id} + E$
$\Rightarrow \textbf{id} + E * E$
$\Rightarrow \textbf{id} + \textbf{id} * E$
$\Rightarrow \textbf{id} + \textbf{id} * \textbf{id}$

$E \Rightarrow^{*}_{lm} \text{id} + E \backslash * E$

Predictive Parsing Table

Productions	
1	T → F T’
2	T’ → ε
3	T’ → * F T’
4	F → id
5	F → (T)

	*	(	)	id	$
T		T → F T’		T → F T’	
T’	T’ → * F T’		T’ → ε		T’ → ε
F		F → (T)		F → id	

Trace “(id)*id”

	*	(	)	id	$
T		T → FT’		T → FT’	
T’	T’ → *FT’		T’ → ε		T’ → ε
F		F → (T)		F → id	

Stack	Input	Output
$T	(id)*id$	
$T’F	(id)*id$	T → F T’
$T’)T(	(id)*id$	F → (T)
$T’)T	id)*id$	
$T’)T’F	id)*id$	T → F T’
$T’)T’id	id)*id$	F → id
$T’)T’	)*id$	
$T’)	)*id$	T’ → ε

Trace "(id)*id"

	*	(	)	id	$
T		T → FT'		T → FT'	
T'	T' → *FT'		T' → ε		T' → ε
F		F → (T)		F → id	

Stack	Input	Output
$T'	*id$	
$T'F*	*id$	**T' → * F T'**
$T'F	id$	
$T'id	id$	**F → id**
$T'	$	
$	$	**T' → ε**

Table-Driven Parsing

stack.push($); stack.push(S);
a = input.read();
forever do begin
  X = stack.peek();
  if X = a **and** a = $ **then** return SUCCESS;
  elsif X = a **and** a != $ **then**
    pop X; a = input.read();
  elsif X != a **and** X ∈ **N and** M[X,a] **then**
    pop X; push right-hand side of M[X,a];
  else ERROR!
end

Predictive Parsing table

- Given a grammar produce the predictive parsing table
- We need to to know for all rules $A \rightarrow \alpha \mid \beta$ the lookahead symbol
- Based on the lookahead symbol the table can be used to pick which rule to push onto the stack
- This can be done using two sets: FIRST and FOLLOW

FIRST and FOLLOW

$$a \in \text{FIRST}(\alpha) \text{ if } \alpha \Rightarrow^* a\beta$$

$$\text{if } \alpha \Rightarrow^* \epsilon \text{ then } \epsilon \in \text{FIRST}(\alpha)$$

$$a \in \text{FOLLOW}(A) \text{ if } S \Rightarrow^* \alpha A a \beta$$

$$a \in \text{FOLLOW}(A) \text{ if } S \Rightarrow^* \alpha A \gamma a \beta$$

$$\text{and } \gamma \Rightarrow^* \epsilon$$

Conditions for LL(1)

- Necessary conditions:
  - no ambiguity
  - no left recursion
  - Left factored grammar
- A grammar G is LL(1) if - whenever
  $A \rightarrow \alpha \mid \beta$
  1. $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$
  2. $\alpha \Rightarrow^* \varepsilon$ implies **!(**$\beta \Rightarrow^* \varepsilon$**)**
  3. $\alpha \Rightarrow^* \varepsilon$ implies $\text{First}(\beta) \cap \text{Follow}(A) = \emptyset$

ComputeFirst(α: string of symbols)

```
// assume α = X1 X2 X3 … Xn
if X1 ∈ T then First[α] := {X1}
else begin
  i:=1; First[α] := ComputeFirst(X1)\{ε};
  while Xi ⇒* ε do begin
    if i < n then
      First[α] := First[α] ∪ ComputeFirst(Xi+1)\{ε};
    else
      First[α] := First[α] ∪ {ε};
    i := i + 1;
  end
end
```

ComputeFirst(α: string of symbols)

```
// assume α = X1 X2 X3 … Xn
if X1 ∈ T then First[α] := {X1}
else begin
  i:=1; First[α] := ComputeFirst(X1)\{ε};
  while Xi ⇒* ε do begin
    if i < n then
      First[α] := First[α] ∪ ComputeFirst(Xi+1)\{ε};
    else
      First[α] := First[α] ∪ {ε};
    i := i + 1;
  end
end
```

Recursion in computing FIRST causes problems when faced with recursive grammar rules

ComputeFirst; modified

```
foreach X ∈ T do First[X] := {X};
foreach p ∈ P : X → ε do First[X]  := {ε};
repeat foreach X ∈ N, p : X → Y1 Y2 Y3 … Yn do
   begin i:=1;
    while Yi ⇒* ε and i <= n do begin
      First[X] := First[X] ∪ First[Yi]\{ε};
      i := i+1;
    end
   if i = n+1 then First[X] := First[X] ∪ {ε};
until no change in First[X] for any X;
```

ComputeFirst; modified

foreach X ∈ **T do** First[X] := X;
foreach p ∈ **P** : X → **ε do** First[X] := {ε};
repeat foreach X ∈ **N,** p : X → $Y_1\ Y_2\ Y_3 \ldots\ Y_n$ **do**
  begin i:=1;
   while Y_i ⇒*
    First[X] := F
    i := i+1;
   end
  if i = n+1 **then** First[X] := First[X] ∪ {ε};
until no change in First[X] for any X;

Non-recursive FIRST computation works with left-recursive grammars. Computes a fixed point for FIRST[X] for all non-terminals X in the grammar. But this algorithm is very inefficient.

ComputeFollow

Follow(S) := {\$};
repeat
 foreach p ∈ **P do**
  case p = A → αBβ **begin**
   Follow[B] := Follow[B] ∪ ComputeFirst(β)\{ε};
   if ε ∈ First(β) **then**
    Follow[B] := Follow[B] ∪ Follow[A];
  end
  case p = A → αB
   Follow[B] := Follow[B] ∪ Follow[A];
until no change in any Follow[N]

Example First/Follow

$S \rightarrow AB$

$A \rightarrow c \mid \varepsilon$

$B \rightarrow cbB \mid ca$

Not an LL(1) grammar

First(A) = {c, ε}

First(B) = {c}

First(cbB) = First(ca) = {c}

First(S) = {c}

Follow(A) = {c}

Follow(A) ∩ First(c) = {c}

Follow(B) = {$}

Follow(S) = {$}

ComputeFirst on Left-recursive Grammars

- ComputeFirst as defined earlier loops on left-recursive grammars
- Here is an alternative algorithm for ComputeFirst
    1. Compute non left-recursive cases of FIRST
    2. Create a graph of recursive cases where FIRST of a non-terminal depends on another non-terminal
    3. Compute Strongly Connected Components (SCC)
    4. Compute FIRST starting from root of SCC to avoid cycles

ComputeFirst on Left-recursive Grammars

- Each Strongly Connected Component can have recursion
- But the connections between SCC means that (by defn) what we have now is a directed acyclic graph – hence without left recursion
- Unlike top-down LL parsing, bottom-up LR parsing allows left-recursive grammars, so this algorithm is useful for LR parsing

ComputeFirst on Left-recursive Grammars

- S → BD | D
- D → d | Sd
- A → CB | a
- C → Bb | ε
- B → Ab | b

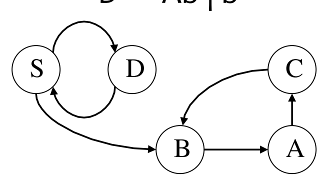
$FIRST_0[A] := \{a, b\}$

$FIRST_0[C] := \{\}$

$FIRST_0[B] := \{b\}$

$FIRST_0[S] := \{b, d\}$

$FIRST_0[D] := \{d\}$



Compute Strongly Connected Components

2 SCCs: e.g. consider B-A-C

$FIRST[B] := FIRST_0[B] + FIRST[A]$

$FIRST[A] := FIRST_0[A] + FIRST[C]$

$FIRST[C] := FIRST_0[C] + FIRST_0[B]$

$FIRST[C] := FIRST[C] + \{\varepsilon\}$

How to compute: Does $X \Rightarrow^* \varepsilon$?

- The question `Does $X \Rightarrow^* \varepsilon$?' can be written as the predicate: nullable(X)

```
Nullable = {} (set containing nullable non-terminals)
Changed = True
While (changed):
   changed = False
   if X is not in Nullable:
       if
         1. X → ε is in the grammar, or
         2. X → Y_1 … Y_n is in the grammar and Y_i is in Nullable for all i
       then
         add X to Nullable; changed = True
```


Converting to LL(1)

$S \rightarrow AB$

$A \rightarrow c \mid \varepsilon$

$B \rightarrow cbB \mid ca$

Note that grammar is regular: c? (cb)* ca

c (c b c b … c b) c a
(c b c b … c b) c a

→

c c (b c b … c b c) a
c (b c b … c b c) a

same as:

c c? (bc)* a

$S \rightarrow cAa$

$A \rightarrow cB \mid B$

$B \rightarrow bcB \mid \varepsilon$

Verifying LL(1) using F/F sets

$S \rightarrow cAa$

$A \rightarrow cB \mid B$

$B \rightarrow bcB \mid \varepsilon$

First(A) = {b, c, ε} Follow(A) = {a}

First(B) = {b, ε} Follow(B) = {a}

First(S) = {c} Follow(S) = {\$}

Building the Parse Table

- Compute First and Follow sets
- For each production $A \rightarrow \alpha$
  - foreach $a \in \text{First}(\alpha)$ add $A \rightarrow \alpha$ to M[A,a]
  - If $\varepsilon \in \text{First}(\alpha)$ add $A \rightarrow \alpha$ to M[A,b] for each b in Follow(A)
  - If $\varepsilon \in \text{First}(\alpha)$ add $A \rightarrow \alpha$ to M[A,\$] if $\$ \in \text{Follow}(\alpha)$
  - All undefined entries are errors

Predictive Parsing Table

Productions	
1	T → F T’
2	T’ → ε
3	T’ → * F T’
4	F → id
5	F → (T)

FIRST(T) = {id, (}
FIRST(T’) = {*, ε}
FIRST(F) = {id, (}

FOLLOW(T) = {$,)}
FOLLOW(T’) = {$,)}
FOLLOW(F) = {*,$,)}

	*	(	)	id	$
T		T → F T’		T → F T’	
T’	T’ → * F T’		T’ → ε		T’ → ε
F		F → (T)		F → id	

Revisit conditions for LL(1)

- A grammar G is LL(1) iff - whenever $A \rightarrow \alpha \mid \beta$
  1. First(α) ∩ First(β) = ∅
  2. $\alpha \Rightarrow^* \varepsilon$ implies **!**($\beta \Rightarrow^* \varepsilon$)
  3. $\alpha \Rightarrow^* \varepsilon$ implies First(β) ∩ Follow(A) = ∅
- No more than one entry per table field

Error Handling

- Reporting & Recovery
  - Report as soon as possible
  - Suitable error messages
  - Resume after error
  - Avoid cascading errors
- Phrase-level vs. Panic-mode recovery

Panic-Mode Recovery

- Skip tokens until *synchronizing set* is seen
  - Follow(A)
    - garbage or missing things after
  - Higher-level start symbols
  - First(A)
    - garbage before
  - Epsilon
    - if nullable
  - Pop/Insert terminal
    - "auto-insert"
- Add "synch" actions to table

Summary so far

- LL(1) grammars, necessary conditions
    - No left recursion
    - Left-factored
- Not all languages can be generated by LL(1) grammar
- LL(1) – Parsing: $O(n)$ time complexity
    - recursive-descent and table-driven predictive parsing
- LL(1) grammars can be parsed by simple predictive recursive-descent parser
    - Alternative: table-driven top-down parser