TD2: LL(1) Parsing

Top-down Parsing

CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

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: $S \rightarrow A\ B$ Input String: ccbca

$A \rightarrow c \mid \varepsilon$

$B \rightarrow cbB \mid ca$

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

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

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

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow \mathbf{id} + E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$

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

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
- Cannot have left-recursion
- Must be left-factored (no left-factors)
- Not all grammars can be made LL(1)

LL(1) Parser

- In recursive-descent
  - for each non-terminal and input token, many choices of production to use
  - Backtracking to remove bad choices
- In LL(1)
  - for each non-terminal and each token, only one production

$S \rightarrow^* \omega A \beta$ and next input token: t

$A \rightarrow \alpha$ is the only production

$\omega \alpha \beta$

Left Factoring

- Consider this grammar
  - $E \rightarrow T + E \mid T$
  - $T \rightarrow int \mid int * T \mid (E)$
- Hard to predict because
  - For T two productions start with int
  - For E it is not clear how to predict
- The grammar should be left-factored
  - Remove common prefixes from multiple productions for each non-terminal

Left Factoring

- In general, for rules

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$$

- Left factoring is achieved by the following grammar transformation:

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Left Factoring

- Recall the grammar
  - $E \rightarrow T + E \mid T$
  - $T \rightarrow \text{int} \mid \text{int} * T \mid (E)$
- Factor out common prefixes for productions
  - $E \rightarrow T X$
  - $X \rightarrow + E \mid \varepsilon$
  - $T \rightarrow \text{int } Y \mid (E)$
  - $Y \rightarrow * T \mid \varepsilon$

Predictive Parsing Table

- Can be specified via 2D tables
  - One dimension for current (leftmost) non-terminal to expand
  - One dimension for next token
  - Each table entry contains one production

Productions	
1	$E \rightarrow T\ X$
2	$X \rightarrow \varepsilon$
3	$X \rightarrow +\ E$
4	$T \rightarrow (\ E\ )$
5	$T \rightarrow \text{int}\ Y$
6	$Y \rightarrow *\ T$
7	$Y \rightarrow \varepsilon$

	+	*	(	)	int	$
E			T X		T X	
X	+ E			ε		ε
T			(E)		int Y	
Y	ε	* T		ε		ε

Predictive Parsing Table

- Consider [E, int] entry
  - When current non-terminal is E and the next input is int, use production $E \rightarrow T\ X$

	Productions
1	$E \rightarrow T\ X$
2	$X \rightarrow \varepsilon$
3	$X \rightarrow +\ E$
4	$T \rightarrow (\ E\ )$
5	$T \rightarrow$ int Y
6	$Y \rightarrow *\ T$
7	$Y \rightarrow \varepsilon$

	+	*	(	)	int	$
E			T X		$\boxed{\text{T X}}$	
X	+ E			ε		ε
T			(E)		int Y	
Y	ε	* T		ε		ε

Predictive Parsing Table

- Consider [Y, +] entry
  - When current non-terminal is Y and the next input is + , get rid of Y
  - Y can be followed by + only if $Y \rightarrow \varepsilon$

Productions	
1	$E \rightarrow T\ X$
2	$X \rightarrow \varepsilon$
3	$X \rightarrow +\ E$
4	$T \rightarrow (\ E\ )$
5	$T \rightarrow \text{int}\ Y$
6	$Y \rightarrow *\ T$
7	$Y \rightarrow \varepsilon$

	+	*	(	)	int	$
E			T X		T X	
X	+ E			ε		ε
T			(E)		int Y	
Y	ε	* T		ε		ε

Predictive Parsing Table

- Blank entries indicate error situations
- Consider [E, *] entry
  - There is no way to derive a string starting with * from non-terminal E

Productions	
1	**E → T X**
2	**X → ε**
3	**X → + E**
4	**T → (E)**
5	**T → int Y**
6	**Y → * T**
7	**Y → ε**

	+	*	(	)	int	$
E			**T X**		**T X**	
X	**+ E**			**ε**		**ε**
T			**(E)**		**int Y**	
Y	**ε**	*** T**		**ε**		**ε**

Predictive Parsing

- Method similar to recursive descent, except
  - For each non-terminal S
  - We look at the next token a
  - And chose the production shown at entry [S,a]
- We use a stack to keep track of pending non-terminals (frontier of parse tree)
- We reject when we encounter an error state
- We accept when we encounter end-of-input and empty stack

Table-Driven Parsing

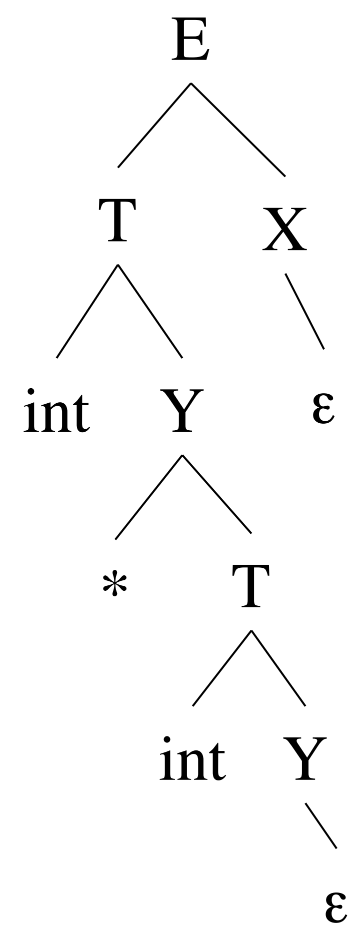
Stack: to keep track of pending non-terminals in the derivation (leaves in parse tree)

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**
    stack.pop(X); a = input.read();
  elsif X != a **and** X $\in$ **N and** M[X,a] not empty **then**
    stack.pop(X);
    stack.push(M[X,a]); /* M[X, a] = $Y_1 \ldots Y_n$ */
  else ERROR! $X \longrightarrow Y_1 \ldots Y_n$
end

Trace “int*int”

	+	*	(	)	int	$
E			**T X**		**T X**	
X	**+ E**			**ε**		**ε**
T			**(E)**		**int Y**	
Y	**ε**	*** T**		**ε**		**ε**

Stack	Input	Action
E $	int*int$	**T X**
T X $	int*int$	**int Y**
int Y X $	int*int$	**terminal**
Y X $	*int$	*** T**
* T X $	*int$	**terminal**
T X $	int$	**int Y**
int Y X $	int$	**terminal**
Y X $	$	**ε**
X $	$	**ε**
$	$	**Accept!**



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

Predictive Parsing Table

- For Nonterminal A, rule $A \rightarrow \alpha$, and the token t, $M[A, t] = \alpha$ in two cases:
- If $\alpha \Rightarrow^* t\,\beta$
  - α can derive a t in the first position
  - We say that $t \in First(\alpha)$
- $A \rightarrow \alpha$ and $\alpha \Rightarrow^* \varepsilon$ and $S \Rightarrow^* \beta\, A\, t\, \delta$
  - Useful if stack has A, input is t and A cannot derive t
  - In this case only option is to get rid of A (by $\alpha \Rightarrow^* \varepsilon$)
    - Can work only if t can follow A in at least on derivation
  - We say $t \in Follow(A)$

FIRST and FOLLOW

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

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

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

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

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

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
```

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

ComputeFirst; modified

foreach $X \in$ **T do** First[X] := {X};

foreach $p \in$ **P** : $X \rightarrow \varepsilon$ **do** First[X] := {ε};

repeat foreach $X \in$ **N**, $p : X \rightarrow Y_1\ Y_2\ Y_3 \ldots\ Y_n$ **do**

    begin i:=1;

      while $Y_i \Rightarrow^* \varepsilon$ and i <= n **do begin**

        First[X] := First[X] $\cup$ First[Y_i]\{ε};

        i := i+1;

      end

    if i = n+1 **then** First[X] := First[X] $\cup$ {ε};

until no change in First[X] for any X;

ComputeFirst; modified

foreach $X \in$ **T** **do** First[X] := X;

foreach $p \in$ **P** : $X \to \varepsilon$ **do** First[X] := {ε};

repeat foreach $X \in$ **N**, $p : X \to Y_1\, Y_2\, Y_3 \ldots\, Y_n$ **do**

  begin i:=1;

   while $Y_i \Rightarrow^*$

    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.

First Sets

First(+) = {+}
First(*) = {*}
First(‘(‘) = {‘(’}
First(‘)’) = {‘)’}
First(int) = {int}

First(E) = ?
First(T) ⊆ First(E)
First(T) = {int, ‘(‘}
First(E) = {int, ‘(‘}
First(X) = {+, ε}
First(Y) = {*, ε}

Productions	
1	E → T X
2	X → ε
3	X → + E
4	T → (E)
5	T → int Y
6	Y → * T
7	Y → ε

Follow Sets

- Algorithm sketch
  1. Add $\$$ to Follow(S)
  2. For each production $A \longrightarrow \alpha\ X\ \beta$
     - Add First(β) – {ε} to Follow(X)
  3. For each $A \longrightarrow \alpha\ X\ \beta$ where $\varepsilon \in$ First(β)
     - Add Follow(A) to Follow(X)
  – Repeat steps 2-3 until no follow set grows

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]
```

Follow Sets. Example

Follow(E)⊆ Follow(X)

Follow(X)⊆ Follow(E)

First(X)-{ε}⊆ Follow(T)

Follow(E)⊆ Follow(T)

Follow(Y)⊆ Follow(T)

Follow(T)⊆ Follow(Y)

Follow(E) = {$,)}

Follow(X) = {$,)}

Follow(T) = {+, $,)}

Follow(Y) = {+, $,)}

Follow(‘(‘) = {(, int}

Follow(‘)‘) = {+,$,)}

Follow(+) = {(, int}

Follow(*) = {(, int}

Follow(int) = {*,+,$,)}

Productions	
1	E → T X
2	X → ε
3	X → + E
4	T → (E)
5	T → int Y
6	Y → * T
7	Y → ε

Example First/Follow

$S \rightarrow AB$

$A \rightarrow c \mid \varepsilon$

$B \rightarrow cbB \mid ca$

Not an LL(1) grammar

$\text{First}(A) = \{c, \varepsilon\}$

$\text{First}(B) = \{c\}$

$\text{First}(cbB) =$

$\quad \text{First}(ca) = \{c\}$

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

$\text{Follow}(A) = \{c\}$

$\text{Follow}(A) \cap$

$\quad \text{First}(c) = \{c\}$

$\text{Follow}(B) = \{\$\}$

$\text{Follow}(S) = \{\$\}$

Building the Parse Table

- Compute First and Follow sets
- For each production $A \rightarrow \alpha$
  - For each $t \in \text{First}(\alpha)$
    - $M[A,t] = \alpha$
  - If $\boldsymbol{\varepsilon} \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$
    - $M[A,t] = \alpha$
  - If $\boldsymbol{\varepsilon} \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(\alpha)$
    - $M[A,\$] = \alpha$
  - All undefined entries are errors

Predictive Parsing Table

First(T) = {int, ‘(‘}
First(X) = {+, ε}
First(Y) = {*, ε}
First(E) = {int, ‘(‘}

Follow(E) = {$,)}
Follow(X) = {$,)}
Follow(T) = {+, $,)}
Follow(Y) = {+, $,)}

Productions	
1	**E → T X**
2	**X → ε**
3	**X → + E**
4	**T → (E)**
5	**T → int Y**
6	**Y → * T**
7	**Y → ε**

	+	*	(	)	int	$
E			**T X**		**T X**	
X	**+ E**			**ε**		**ε**
T			**(E)**		**int Y**	
Y	**ε**	*** T**		**ε**		**ε**

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}

First(B) = {b, ε}

First(S) = {c}

Follow(A) = {a}

Follow(B) = {a}

Follow(S) = {$}

Building the Parse Table

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

Predictive Parsing Table

Productions	
1	$T \rightarrow F\ T'$
2	$T' \rightarrow \varepsilon$
3	$T' \rightarrow *\ F\ T'$
4	$F \rightarrow id$
5	$F \rightarrow (\ T\ )$

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

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

	*	(	)	id	$
T		$T \rightarrow F\ T'$		$T \rightarrow F\ T'$	
T'	$T' \rightarrow *\ F\ T'$		$T' \rightarrow \varepsilon$		$T' \rightarrow \varepsilon$
F		$F \rightarrow (\ T\ )$		$F \rightarrow id$	

Revisit conditions for LL(1)

- A grammar G is LL(1) iff - 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 $\text{First}(\beta) \cap \text{Follow}(A) = \emptyset$
- 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

Extra Slides

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

- $S \rightarrow BD \mid D$
- $D \rightarrow d \mid Sd$
- $A \rightarrow CB \mid a$
- $C \rightarrow Bb \mid \varepsilon$
- $B \rightarrow Ab \mid b$

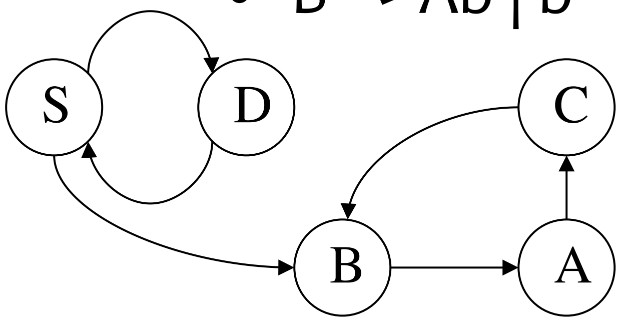
$FIRST_0[A] := \{a\}$

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

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

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

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



Compute Strongly Connected Components

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

$FIRST[B] := FIRST_0[B] + ComputeFirst(A)$

$FIRST[A] := FIRST_0[A] + ComputeFirst(C\ )$

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

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

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

Examples

$S \rightarrow A\ B\ C$

$A \rightarrow a \mid \varepsilon$

$B \rightarrow b\ B \mid \varepsilon$

$C \rightarrow c \mid \varepsilon$

Is this LL(1)?

$S \rightarrow F$

$F \rightarrow A\ (\ B\ ) \mid B\ A$

$A \rightarrow x \mid y$

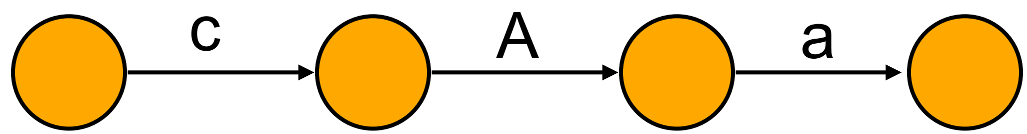
$B \rightarrow a\ B \mid b\ B \mid \varepsilon$

Is this LL(1)?

Transition Diagram

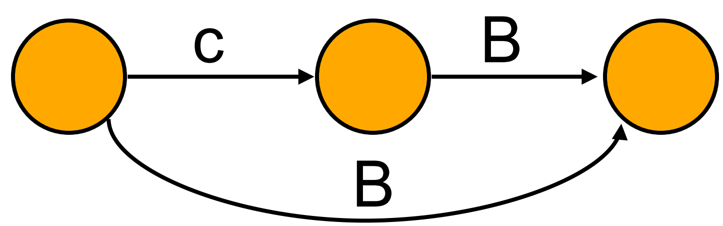
$S \rightarrow cAa$

S:



$A \rightarrow cB \mid B$

A:



$B \rightarrow bcB \mid \varepsilon$

B: