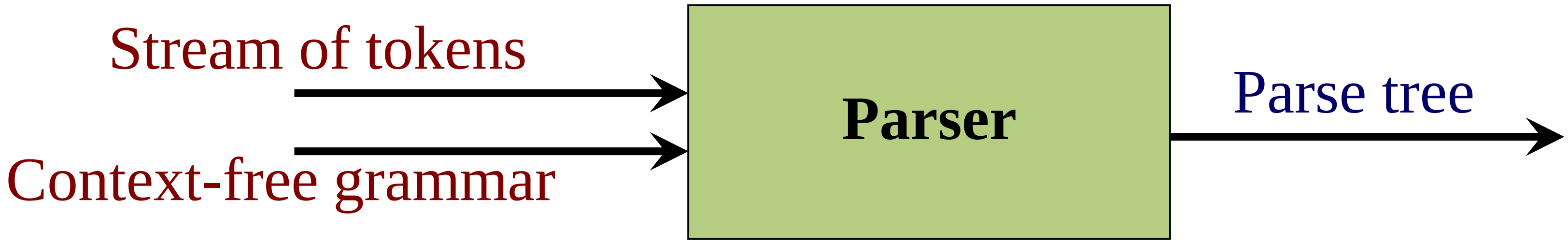
Parsing

Outline

- Top-down v.s. Bottom-up
- Top-down parsing
  - Recursive-descent parsing
  - LL(1) parsing
    - LL(1) parsing algorithm
    - First and follow sets
    - Constructing LL(1) parsing table
    - Error recovery
- Bottom-up parsing
  - Shift-reduce parsers
  - LR(0) parsing
    - LR(0) items
    - Finite automata of items
    - LR(0) parsing algorithm
    - LR(0) grammar
  - SLR(1) parsing
    - SLR(1) parsing algorithm
    - SLR(1) grammar
    - Parsing conflict

Introduction

- Parsing is a process that constructs a syntactic structure (i.e. parse tree) from the stream of tokens.
- We already learn how to describe the syntactic structure of a language using (context-free) grammar.
- So, a parser only need to do this?



Top–Down Parsing

- A parse tree is created from root to leaves
- The traversal of parse trees is a preorder traversal
- Tracing leftmost derivation
- Two types:
  - Backtracking parser
  - Predictive parser

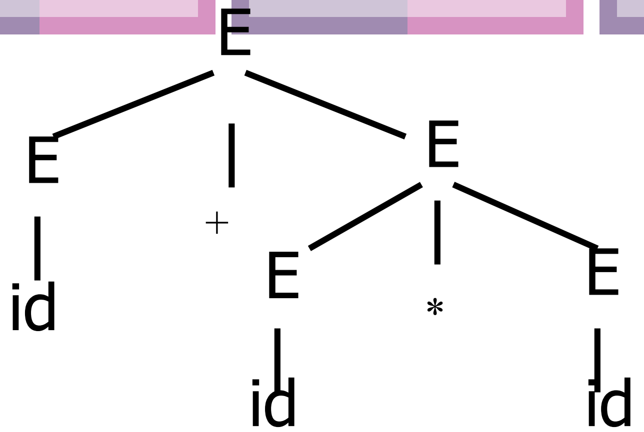
Bottom–Up Parsing

- A parse tree is created from leaves to root
- The traversal of parse trees is a reversal of postorder traversal
- Tracing rightmost derivation

Try different structures and backtrack if it does not matched the input

Guess the structure of the parse tree from the next input

Parse Trees and Derivations



Top-down parsing

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

Top-down Parsing

- What does a parser need to decide?
  - Which production rule is to be used at each point of time ?
- How to guess?
- What is the guess based on?
  - What is the next token?
    - Reserved word if, open parentheses, etc.
  - What is the structure to be built?
    - If statement, expression, etc.

Top-down Parsing

- Why is it difficult?
  - Cannot decide until later
    - Next token: **if**   Structure to be built: St
    - St $\rightarrow$ MatchedSt | UnmatchedSt
    - UnmatchedSt $\rightarrow$

      if (E) St| **if** (E) MatchedSt **else** UnmatchedSt
    - MatchedSt $\rightarrow$ **if** (E) MatchedSt **else** MatchedSt |...
  - Production with empty string
    - Next token: **id**   Structure to be built: par
    - par $\rightarrow$ parList | λ
    - parList $\rightarrow$ exp , parList | exp

Recursive-Descent

- Write one procedure for each set of productions with the same nonterminal in the LHS
- Each procedure recognizes a structure described by a nonterminal.
- A procedure calls other procedures if it need to recognize other structures.
- A procedure calls *match* procedure if it need to recognize a terminal.

Recursive-Descent: Example

E → E O F | F
O → + | -
F → (E) | id

E ::= F {O F}
O ::= + | -
F ::= (E) | id

```
procedure F
{   switch token
    {     case (:  match('(');
                   E;
                   match(')');
          case id: match(id);
          default:  error;
    }
}
```

```
procedure E
{        E; O; F; }
```

- For this grammar:
  - We cannot decide which rule to use for E, and
  - If we choose E → E O F, it leads to infinitely recursive loops.
- Rewrite the grammar into EBNF

```
procedure E
{   F;
    while (token=+ or token=-)
    {     O; F;   }
}
```

Match procedure

```
procedure match(expTok)
{   if (token==expTok)
    then    getToken
    else    error
}
```

- The token is not consumed until `getToken` is executed.

Problems in Recursive-Descent

- Difficult to convert grammars into EBNF
- Cannot decide which production to use at each point
- Cannot decide when to use λ-production $A \rightarrow \lambda$

LL(1) Parsing

- LL(1)
  - Read input from (**L**) left to right
  - Simulate (**L**) leftmost derivation
  - **1** lookahead symbol
- Use stack to simulate leftmost derivation
  - Part of sentential form produced in the leftmost derivation is stored in the stack.
  - Top of stack is the leftmost nonterminal symbol in the fragment of sentential form.

Concept of LL(1) Parsing

- Simulate leftmost derivation of the input.
- Keep part of sentential form in the stack.
- If the symbol on the top of stack is a terminal, try to match it with the next input token and pop it out of stack.
- If the symbol on the top of stack is a nonterminal X, replace it with Y if we have a production rule $X \rightarrow Y$.
  - Which production will be chosen, if there are both $X \rightarrow Y$ and $X \rightarrow Z$?

Example of LL(1) Parsing

E ⇒TX
⇒FNX
⇒(E)NX
⇒(TX)NX
⇒(FNX)NX
⇒(nNX)NX
⇒(nX)NX
⇒(nATX)NX
⇒(n+TX)NX
⇒(n+FNX)NX
⇒(n+(E)NX)NX
⇒(n+(TX)NX)NX
⇒(n+(FNX)NX)NX
⇒(n+(nNX)NX)NX
⇒(n+(nX)NX)NX
⇒(n+(n)NX)NX
⇒(n+(n)X)NX
⇒(n+(n))NX
⇒(n+(n))MFNX
⇒(n+(n))*FNX
⇒(n+(n))*nNX
⇒(n+(n))*nX
⇒(n+(n))*n

Stack: n, N, X,), N, *, n, N, X, $

Finished

Input: (n + (n)) * n $

E → T X
X → A T X | λ
A → + | -
T → F N
N → M F N | λ
M → *
F → (E) | n

LL(1) Parsing Algorithm

Push the start symbol into the stack

WHILE stack is not empty ($ is not on top of stack) and the stream of tokens is not empty (the next input token is not $)

SWITCH (Top of stack, next token)

CASE (terminal a, a):

Pop stack;   Get next token

CASE (nonterminal A, terminal a):

IF the parsing table entry M[A, a] is not empty THEN

Get $A \rightarrow X_1\ X_2\ \ldots\ X_n$ from the parsing table entry M[A, a] Pop stack;

Push $X_n\ \ldots\ X_2\ X_1$ into stack in that order

ELSE Error

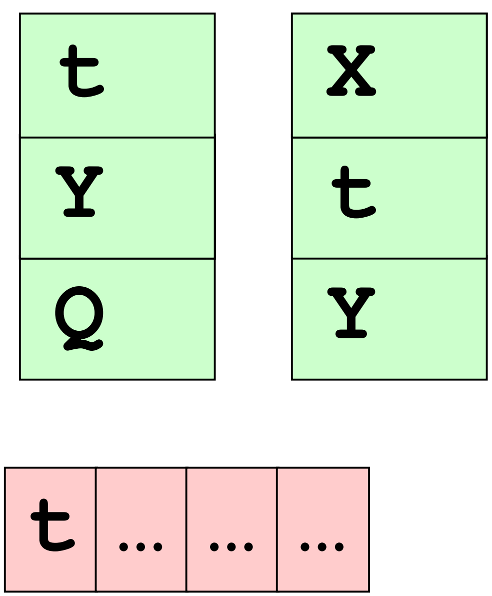
CASE ($,$):   Accept

OTHER: Error

LL(1) Parsing Table

If the nonterminal N is on the top of stack and the next token is t, which production rule to use?

- Choose a rule $N \rightarrow X$ such that
  - $X \Rightarrow^* tY$ or
  - $X \Rightarrow^* \lambda$ and $S \Rightarrow^* WNtY$



First Set

- Let X be λ or be in V or T.
- First(X) is the set of the first terminal in any sentential form derived from X.
  - If X is a terminal or λ, then First(X) ={X}.
  - If X is a nonterminal and $X \rightarrow X_1 X_2 \ldots X_n$ is a rule, then
    - First(X_1) -{λ} is a subset of First(X)
    - First(X_i)-{λ} is a subset of First(X) if for all $j<i$ First(X_j) contains {λ}
    - λ is in First(X) if for all $j \leq n$ First(X_j)contains λ

Examples of First Set

exp → exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → *
factor → (exp) | num

First(addop) = {+, -}
First(mulop) = {*}
First(factor) = {(, num}
First(term) = {(, num}
First(exp) = {(, num}

st → ifst | other
ifst → if (exp) st elsepart
elsepart → else st | λ
exp → 0 | 1

First(exp) = {0,1}
First(elsepart) = {else, λ}
First(ifst) = {if}
First(st) = {if, other}

Algorithm for finding First(A)

For all terminals a, First(a) = {a}

For all nonterminals A, First(A) := {}

While there are changes to any First(A)

For each rule $A \rightarrow X_1 X_2 \dots X_n$

For each X_i in $\{X_1, X_2, \dots, X_n\}$

If for all j<i $First(X_j)$ contains λ,

Then

add $First(X_i)$-{λ} to First(A)

If λ is in $First(X_1)$, $First(X_2)$, ..., and $First(X_n)$

Then add λ to First(A)

If A is a terminal or λ, then First(A) = {A}.

If A is a nonterminal, then for each rule $A \rightarrow X_1 X_2 \dots X_n$, First(A) contains $First(X_1)$ - {λ}.

If also for some i<n, $First(X_1)$, $First(X_2)$, ..., and $First(X_i)$ contain λ, then First(A) contains $First(X_{i+1})$-{λ}.

If $First(X_1)$, $First(X_2)$, ..., and $First(X_n)$ contain λ, then First(A) also contains λ.

Finding First Set: An Example

exp → term exp'

exp' → addop term exp' | λ

addop → + | -

term → factor term'

term' → mulop factor term' | λ

mulop → *

factor → (exp) | num

	First
exp	
exp'	λ
addop	+ -
term	(num
term'	λ
mulop	*
factor	(num

Follow Set

- Let \$ denote the end of input tokens
- If A is the start symbol, then \$ is in Follow(A).
- If there is a rule $B \rightarrow X\ A\ Y$, then First(Y) - $\{\lambda\}$ is in Follow(A).
- If there is production $B \rightarrow X\ A\ Y$ and λ is in First(Y), then Follow(A) contains Follow(B).

Algorithm for Finding Follow(A)

Follow(S) = {\$}

FOR each A in V-{S}

Follow(A)={}

WHILE change is made to some Follow sets

FOR each production $A \rightarrow X_1\ X_2\ \ldots\ X_n$,

FOR each nonterminal X_i

Add First($X_{i+1}\ X_{i+2} \ldots X_n$)-{$\lambda$} into Follow($X_i$).

(NOTE: If i=n, $X_{i+1}\ X_{i+2} \ldots X_n = \lambda$)

IF λ is in First($X_{i+1}\ X_{i+2} \ldots X_n$) THEN

Add Follow(A) to Follow(X_i)

If A is the start symbol, then \$ is in Follow(A).

If there is a rule $A \rightarrow Y\ X\ Z$, then First(Z) - {λ} is in Follow(X).

If there is production $B \rightarrow X\ A\ Y$ and λ is in First(Y), then Follow(A) contains Follow(B).

Finding Follow Set: An Example

exp → term exp’

exp’ → addop term exp’ | λ

addop → + | -

term → factor term’

term’ → mulop factor term’ |λ

mulop → *

factor → (exp) | num

	First	Follow
exp	(num	$)
exp’	λ + -	$)
addop	+ -	
term	(num	+ - $)
term’	λ *	
mulop	*	
factor	(num	

Constructing LL(1) Parsing Tables

FOR each nonterminal A and a production $A \rightarrow X$

FOR each token a in First(X)

$A \rightarrow X$ is in M(A, a)

IF λ is in First(X) THEN

FOR each element a in Follow(A)

Add $A \rightarrow X$ to M(A, a)

Example: Constructing LL(1) Parsing Table

	First	Follow
exp	{(, num}	{$,)}
exp’	{+,-, λ}	{$,)}
addop	{+,-}	{(,num}
term	{(,num}	{+,-,),$}
term’	{*, λ}	{+,-,),$}
mulop	{*}	{(,num}
factor	{(, num}	{*,+,-,),$}

1 exp → term exp’
2 exp’ → addop term exp’
3 exp’ → λ
4 addop → +
5 addop → -
6 term → factor term’
7 term’ → mulop factor term’
8 term’ → λ
9 mulop → *
10 factor → (exp)
11 factor → num

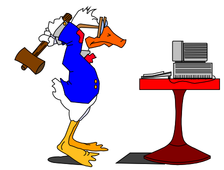
	(	)	+	-	*	n	$
exp	1					1	
exp’		3	2	2			3
addop			4	5			
term	6					6	
term’		8	8	8	7		8
mulop					9		
factor	10					11	

LL(1) Grammar

- A grammar is an LL(1) grammar if its LL(1) parsing table has at most one production in each table entry.

LL(1) Parsing Table for non-LL(1) Grammar

1 exp → exp addop term
2 exp → term
3 term → term mulop factor
4 term → factor
5 factor → (exp)
6 factor → num
7 addop → +
8 addop → -
9 mulop → *



	(	)	+	-	*	num	$
exp	1,2					1,2	
term	3,4					3,4	
factor	5					6	
addop			7	8			
mulop					9		

First(exp) = { (, num }
First(term) = { (, num }
First(factor) = { (, num }
First(addop) = { +, - }
First(mulop) = { * }

Causes of Non-LL(1) Grammar

- What causes grammar being non-LL(1)?
  - Left-recursion
  - Left factor

Left Recursion

- Immediate left recursion
  - $A \to A\ X \mid Y$ $A=Y\ X^*$
  - $A \to A\ X_1 \mid A\ X_2 \mid \ldots \mid A\ X_n \mid Y_1 \mid Y_2 \mid \ldots \mid Y_m$

$A=\{Y_1, Y_2, \ldots, Y_m\}\ \{X_1, X_2, \ldots, X_n\}^*$

- Can be removed very easily
  - $A \to Y\ A',\ A' \to X\ A' \mid \lambda$
  - $A \to Y_1\ A' \mid Y_2\ A' \mid \ldots \mid Y_m\ A',$ $A' \to X_1\ A' \mid X_2\ A' \mid \ldots \mid X_n\ A' \mid \lambda$

- General left recursion
  - $A \Rightarrow X \Rightarrow^* A\ Y$

- Can be removed when there is no empty-string production and no cycle in the grammar

Removal of Immediate Left Recursion

exp → exp + term | exp - term | term

term → term * factor | factor

factor → (exp) | num

- Remove left recursion

exp → term exp'

exp = term (± term)*

exp' → + term exp' | - term exp' | λ

term → factor term'

term = factor (* factor)*

term' → * factor term' | λ

factor → (exp) | num

General Left Recursion

- Bad News!
  - Can only be removed when there is no empty-string production and no cycle in the grammar.
- Good News!!!!
  - Never seen in grammars of any programming languages

Left Factoring

- Left factor causes non-LL(1)
  - Given $A \rightarrow X\ Y \mid X\ Z$. Both $A \rightarrow X\ Y$ and $A \rightarrow X\ Z$ can be chosen when A is on top of stack and a token in First(X) is the next token.

$A \rightarrow X\ Y \mid X\ Z$

can be left-factored as

$A \rightarrow X\ A'$ and $A' \rightarrow Y \mid Z$

Example of Left Factor

ifSt $\rightarrow$ **if (** exp **)** st **else** st | **if (** exp **)** st

can be left-factored as

ifSt $\rightarrow$ **if (** exp **)** st elsePart

elsePart $\rightarrow$ **else** st | λ

seq $\rightarrow$ st ; seq | st

can be left-factored as

seq $\rightarrow$ st seq'

seq' $\rightarrow$; seq | λ