

Top-down Parsing

Outline

- Inference rules for computing grammar properties
- Top-down parsing
- SLL(1) grammars
- Recursive-descent parsing
- Generating parsing tables for SLL(1) grammars

Concepts

- Distinction between language and grammar
 - Language: set of strings over some alphabet
 - Grammar: a set of rules for generating the strings in a language
 - G : grammar, $L(G)$: language generated by grammar
- Recognition vs. parsing, given grammar G and word w
 - Recognition is decision problem - is w in $L(G)$?
 - Parsing: if w in $L(G)$, show a derivation (proof)
- Context-free grammar: 4-tuple
 - S : Start symbol
 - T : Terminals aka tokens (also written as Σ by some authors)
 - N : Non-terminals (also written as V)
 - P : Productions
- Sentential forms and sentences
 - Sentential form: string that can be obtained by starting with S and using productions as rewrite rules to rewrite nonterminals
 - Sentence: sentential form without nonterminals (word in language)

Concepts

- Derivation of string using grammar
 - Start from S and repeatedly rewrite a nonterminal using the productions of the grammar until there are no nonterminals left
 - Leftmost/rightmost-derivation: rewrite only the leftmost/rightmost nonterminal at each step
- Ambiguous grammar
 - Grammar in which there are two or more leftmost derivations for some word

Inference rules

Big picture

- Given grammar, we need to compute certain sets that will be used by parser
- These sets are usually specified recursively
 - (e.g.) if A is a member of the set, then B is also a member of that set
- Inference rules are an elegant way to specify these recursive sets without writing code
- From inference rules, it is easy to write down code

Parsing SLL(1) grammars

- Compute relations NULLABLE, FIRST and FOLLOW
- $\text{NULLABLE} \subseteq N$
 - set of non-terminals that can be rewritten to the empty string
- $\text{FIRST}(A) \subseteq T \cup \{\varepsilon\}$
 - if A can be rewritten to a string starting with terminal t , then t is in $\text{FIRST}(A)$
 - if A can be rewritten to ε , then ε is in $\text{FIRST}(A)$
- $\text{FOLLOW}(A) \subseteq T \cup \{\$\}$
 - $\$$ is a special symbol (see later)
 - set of terminals that can follow A in a sentential form
 $t \in \text{FOLLOW}(A)$ if we can derive a string $\dots At \dots$
- These relations are defined for any context-free grammar but if grammar is SLL(1), they can be used to implement a recursive-descent parser.

NULLABLE

- ϵ -production
 - A production whose righthand side is the empty string ϵ
 - (e.g.) $A \rightarrow \epsilon$
- Nullable non-terminal
 - Nonterminal that can be rewritten to ϵ
- NULLABLE: set of nullable nonterminals
 - If there is production $A \rightarrow \epsilon$, then $A \in \text{NULLABLE}$
 - If there is production $A \rightarrow Y_1..Y_n$ and $Y_i \in \text{NULLABLE}$ for all Y_i , then $A \in \text{NULLABLE}$

Example

$S \rightarrow At$

$A \rightarrow BC \mid x$

$B \rightarrow t \mid \epsilon$

$C \rightarrow v \mid \epsilon$

$\text{NULLABLE} = \{B, C, A\}$

Inference rules for NULLABLE

- This can be specified compactly using inference rules
 - If predicate in numerator is true, then predicate in denominator must be true
 - Caution: inference rule is “if-then”, not “if and only if”

$NULLABLE \subseteq N$

Smallest set for which

$$\frac{(A \rightarrow \varepsilon) \in P}{A \in NULLABLE}$$

$$\frac{(A \rightarrow Y_1..Y_n) \in P \quad Y_1..Y_n \in NULLABLE}{A \in NULLABLE}$$

- Given proposal $X \subseteq N$, we can check if proposal is consistent with inference rules.
- Examples:
 - $\{\}$: no
 - $\{A\}$: no
 - $\{A,B,C\}$: yes
 - $\{S,A,B,C\}$: yes (numerators of both rules are false)
- For our problems, we want the **smallest** set that is consistent with all inference rules
 - $\{A,B,C\}$

Grammar

$S \rightarrow At$
 $A \rightarrow BC \mid x$
 $B \rightarrow t \mid \varepsilon$
 $C \rightarrow v \mid \varepsilon$

Computing NULLABLE

- Initialize NULLABLE to $\{\}$
- Round-based computation
 - In each round, visit every production and see if you can add more elements to NULLABLE
 - Terminate when NULLABLE does not change in some round

$NULLABLE \subseteq N$

Smallest set for which

$$\frac{(A \rightarrow \varepsilon) \in P}{A \in NULLABLE}$$

$$\frac{(A \rightarrow Y_1..Y_n) \in P \quad Y_1, .. Y_n \in NULLABLE}{A \in NULLABLE}$$

Computing NULLABLE:

$NULLABLE = \{ \}$

repeat

for each production $A \rightarrow Y_1..Y_n$

if $Y_1..Y_n = \varepsilon$ or all Y_i are in NULLABLE, add A to NULLABLE

until NULLABLE set does not change

Slicker way to write inference rules

- Define predicate NULLABLE: string \rightarrow true/false
- $NULLABLE(\alpha) = \text{true}$ if α can be rewritten to ε
 - $NULLABLE(\varepsilon) = \text{true}$
 - $NULLABLE(t:T) = \text{false}$
 - $NULLABLE(Y_1..Y_n) = NULLABLE(Y_1) \& \dots \& NULLABLE(Y_n)$

$NULLABLE \subseteq N$

Smallest set for which

$$\frac{(A \rightarrow \varepsilon) \in P}{A \in NULLABLE}$$

$$\frac{(A \rightarrow Y_1..Y_n) \in P \quad Y_1..Y_n \in NULLABLE}{A \in NULLABLE}$$

$NULLABLE \subseteq N$

Smallest set for which

$$\frac{(A \rightarrow Y_1..Y_n) \in P \quad NULLABLE(Y_1..Y_n)}{A \in NULLABLE}$$

Lifting

- We generalized NULLABLE

- from non-terminals
- to strings of terminals and non-terminals

- From Google AI

“In programming languages, "lifting" refers to the process of transforming a function to operate on a different data structure, usually a more complex one like a container or a monad, while preserving its original behavior, essentially allowing you to apply a function to elements within that data structure without explicitly iterating through them; this is often achieved using higher-order functions and is particularly prevalent in functional programming languages like Haskell where functions can be treated as data.”

Another relation: FIRST

- $\text{FIRST}(A) \subseteq T \cup \{\varepsilon\}$ (A is nonterminal)
 - if A can be rewritten to a string starting with terminal t , then $t \in \text{FIRST}(A)$
 - if A can be rewritten to ε (that is, $\text{NULLABLE}(A)$), then $\varepsilon \in \text{FIRST}(A)$
- Convenient to define $F = T \cup \{\varepsilon\}$
 - f denotes element of F (so either terminal or ε)
- As with NULLABLE , it is convenient to extend this to arbitrary strings

Example

$S \rightarrow A$

$A \rightarrow BC \mid x$

$B \rightarrow t \mid \varepsilon$

$C \rightarrow v \mid \varepsilon$

NULLABLE = {A,B,C}

FIRST(A) = {x,t,v, ε }

FIRST(B) = {t, ε }

FIRST(C) = {v, ε }

FIRST(S) = {x,t,v, ε }

Note: $v \in \text{FIRST}(A)$ even though it is not in $\text{FIRST}(B)$.
This is because B is nullable and $v \in \text{FIRST}(C)$.

General FIRST relation

$$\text{FIRST}(\alpha) \subseteq F = T \cup \{\varepsilon\}$$

if α can be rewritten to a string starting with terminal t ,
then t is in $\text{FIRST}(\alpha)$

if α can be rewritten to ε , then ε is in $\text{FIRST}(\alpha)$

Helper function: $+_1$

binary operation on subsets of $T \cup \{\varepsilon\}$

concatenate each element of first set with each element of second set
and truncate to 1 symbol

$$\{\varepsilon, a, b\} +_1 \{\varepsilon, c\} = \{\varepsilon, a, b, c\}$$

$$\text{FIRST}(\varnothing) = \{\varepsilon\}$$

$$\text{FIRST}(t) = \{t\}$$

$$\text{FIRST}(Y_1 \dots Y_n) = \text{FIRST}(Y_1) +_1 \text{FIRST}(Y_2) +_1 \dots \text{FIRST}(Y_n)$$

So if we can compute FIRST for all nonterminals, we can compute it for any string

Inference rule for FIRST

$\text{FIRST}(A) \subseteq F = T \cup \{\varepsilon\}$ Smallest sets for which

$$\frac{(A \rightarrow Y_1 Y_2 \dots Y_n) \in P \quad f \in \text{FIRST}(Y_1 Y_2 \dots Y_n)}{f \in \text{FIRST}(A)}$$

FIRST sets can be computed using iterative algorithm as before

```
for each A:N do {  
    FIRST[A] = { };  
}  
repeat  
    for each production  $A \rightarrow Y_1 \dots Y_n$  do {  
        FIRST[A] = FIRST[A]  $\cup$  FIRST( $Y_1 \dots Y_n$ )  
    }  
until FIRST sets do not change
```


Example

Grammar

$S \rightarrow A$

$A \rightarrow BC \mid x$

$B \rightarrow t \mid \varepsilon$

$C \rightarrow v \mid \varepsilon$

```
for each A:N do {  
    FIRST[A] = { };  
}  
repeat  
    for each production A → Y1..Yn do {  
        FIRST[A] = FIRST[A] U FIRST(Y1..Yn)  
    }  
until FIRST sets do not change
```

$N \backslash \text{Round}$	0	1	2	3
S	{ }	{ }	{x}	{x,t,v, ε }
A	{ }	{x}	{x,t,v, ε }	{x,t,v, ε }
B	{ }	{t, ε }	{t, ε }	{t, ε }
C	{ }	{v, ε }	{v, ε }	{v, ε }


Why inference rules?

From Parsing chapter in Dragon book

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

FOLLOW

- $\text{FOLLOW}(B) \subseteq T \cup \{\$ \}$ (\$ is like end-of-file/string)
 - $t \in \text{FOLLOW}(B)$ if we can derive a sentential form $\dots Bt\dots$
- By convention, $\$ \in \text{FOLLOW}(S)$
- First sentential form (convention):
 - $S \$$ so $\text{FOLLOW}(S) = \{\$ \}$
- Next sentential form.: say we use $S \rightarrow A_1 \dots A_i \boxed{A_{i+1} \dots A_m}$
 - String becomes $A_1 \dots A_i A_m \$$
 - What is $\text{FOLLOW}(A_i)$?
 
 - Answer: $\text{FIRST}(A_{i+1} \dots A_m \$) = \text{FIRST}(A_{i+1} \dots A_m) +_1 \text{FOLLOW}(S)$
- In general: production $A \rightarrow X_1 \dots X_k B Y_1 \dots Y_m$
 - $t \in (\text{FIRST}(Y_1 \dots Y_m) +_1 \text{FOLLOW}(A)) \Rightarrow t \in \text{FOLLOW}(B)$

Inference rules for FOLLOW

$\text{FOLLOW} \subseteq T \cup \{\$\}$

$\$ \in \text{FOLLOW}(S)$

$$\frac{(A \rightarrow X_1 \dots X_k B Y_1 \dots Y_n) \in P \quad t \in \text{FIRST}(Y_1 \dots Y_n) + {}_1 \text{FOLLOW}(A)}{t \in \text{FOLLOW}(B)}$$

Example for FOLLOW

$S \rightarrow A$

$A \rightarrow BC \mid x$

$B \rightarrow t \mid \varepsilon$

$C \rightarrow v \mid \varepsilon$

NULLABLE = {A,B,C}

FIRST(A) = {x,t,v, ε }

FIRST(B) = {t, ε }

FIRST(C) = {v, ε }

FIRST(S) = {x,t,v, ε }

FOLLOW(A) = {\$}

FOLLOW(B) = {v,\$}

FOLLOW(C) = {\$}

FOLLOW(S) = {\$}

Computing all relations

```
for each A do {  
    NULLABLE[A] = false;  
    FIRST[A] = FOLLOW[A] = { };  
}  
FOLLOW[S] = {$};  
repeat  
    for each production  $A \rightarrow Y_1..Y_n$  do {  
        if NULLABLE( $Y_1..Y_n$ ) then NULLABLE[A] = true;  
  
        FIRST[A] = FIRST[A]  $\cup$  FIRST( $Y_1..Y_n$ )  
  
        for each  $Y_i$  do  
            FOLLOW[ $Y_i$ ] = FOLLOW[ $Y_i$ ]  $\cup$  (FIRST( $Y_{i+1}..Y_k$ )  $\cup$  FOLLOW[A])  
        }  
    until sets do not change
```

Summary

- Given context-free grammar
 - compute certain relations needed for parsing
- Commonly used relations
 - NULLABLE
 - FIRST
 - FOLLOW
 - Can be generalized to k symbols: FIRST_k and FOLLOW_k
- Specifying these relations: inference rules
 - Separate what needs to be computed from how it is computed
- Important abstract concepts independent of parsing
 - Lifting
 - Inference rules

Recursive-descent parsing

SLL(1) Parsing Goal

- String view:
 - Determine a Leftmost derivation of the input while reading the input from Left to right while looking ahead at most 1 input token
- Tree view:
 - Beginning with the start symbol, grow a parse tree top-down in left-to-right preorder while looking ahead at most 1 input token beyond the input prefix matched by the parse tree derived so far

Expression Grammar

- Consider

Grammar: $E \rightarrow (E + E) \mid \text{int}$

String: $(2 + 3)$

- Leftmost derivation

$E \Rightarrow (E + E) \Rightarrow (2 + E) \Rightarrow (2 + 3)$

- How can we decide which production to use in first step?

$E \rightarrow \text{int}$

$E \rightarrow (E+E)$

- Answer: examine next unread token in input. Three cases:

int : use the production $E \rightarrow \text{int}$

(: use the production $E \rightarrow (E+E)$

Otherwise: parse error.

- This rule works for *all* derivation steps, not just the first.
- Next unread token in input is called “look-ahead”

Recursive-Descent Recognizer

$E \rightarrow (E + E) \mid \text{int}$

```
token = input.read(); //global variable
parse_E();

//precondition: global variable "token" has look-ahead token
void parse_E() {
    switch (token) {
        case int: token = input.read(); return;
        case '(':
            {token = input.read();
             parse_E();
             if (token != '+') throw new ParseError();
             token = input.read();
             parse_E();
             if (token != ')') throw new ParseError();
             token = input.read(); return;
            }
        default: throw new ParseError(); }
}

//postcondition: global variable "token" has look-ahead token
```

Non-SLL(1) Grammar

- Consider the grammar

$$S \rightarrow E + S \mid E$$

$$E \rightarrow \text{num} \mid (S)$$

- and the two derivations

$$S \Rightarrow E \Rightarrow (S) \Rightarrow (E) \Rightarrow (3)$$

$$S \Rightarrow E+S \Rightarrow (S)+S \Rightarrow (E)+E \Rightarrow (3)+E \Rightarrow (3)+4$$

- How could we decide between

$$S \rightarrow E$$

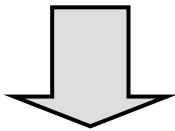
$$S \rightarrow E+S$$

as the first derivation step based on one (or even some finite number k) of look-ahead tokens?

- We can't!
 - The sample grammar is not SLL(1)
 - The sample grammar is not SLL(k) for any k .

Making a grammar SLL(1)

$S \rightarrow E+S$
 $S \rightarrow E$
 $E \rightarrow \text{num}$
 $E \rightarrow (S)$



$S \rightarrow ES'$
 $S' \rightarrow \varepsilon$
 $S' \rightarrow +S$
 $E \rightarrow \text{num}$
 $E \rightarrow (S)$

- **Left-factoring**: Factor common S prefix E , add new non-terminal S' for what follows that prefix
- Convert **left-recursion** to right-recursion
- Not all context-free languages have an SLL(1) grammar

General picture: parser for SLL(1) grammar

- One procedure for each non-terminal
- Global variable *token* contains look-ahead token
- Procedure for non-terminal N
 - Precondition: variable token has look-ahead token
 - Action of procedure: read in a sequence of terminals that can be derived from non-terminal N
 - Postcondition: variable token has look-ahead token
- Body of procedure is a big case statement
- Each case:
 - one possible look-ahead token (say t)
 - invokes parsing actions for a production of N
- Question: how do we determine which production to use for a given [N,t] combination?

Abstraction: predictive parsing table

Grammar

$S \rightarrow ES'$
 $S' \rightarrow \varepsilon$
 $S' \rightarrow +S$
 $E \rightarrow \text{num}$
 $E \rightarrow (S)$

	num	+	()	\$
S	$\rightarrow ES'$			$\rightarrow ES'$	
S'		$\rightarrow +S$		$\rightarrow \varepsilon$	$\rightarrow \varepsilon$
E	$\rightarrow \text{num}$			$\rightarrow (S)$	

One row for each non-terminal in V

One column for each terminal in $T \cup \{\$\}$

Table $[r,c]$ is the production to be used

when expanding non-terminal r and look-ahead token is c

(empty table entries: throw parsing error)

Given parsing table, it is easy to generate recursive-descent parser

Recursive-Descent Parser

```
void parse_S () {
    switch (token) {
        case num: parse_E(); parse_S'(); return;
        case '(': parse_E(); parse_S'(); return;
        default: throw new ParseError();
    }
}
```

lookahead token

	num	+	()	EOF
→ S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ num		→ (S)		

Recursive-Descent Parser

```
void parse_S'() {
    switch (token) {
        case '+': token = input.read(); parse_S(); return;
        case ')': return;
        case EOF: return;
        default: throw new ParseError();
    }
}
```

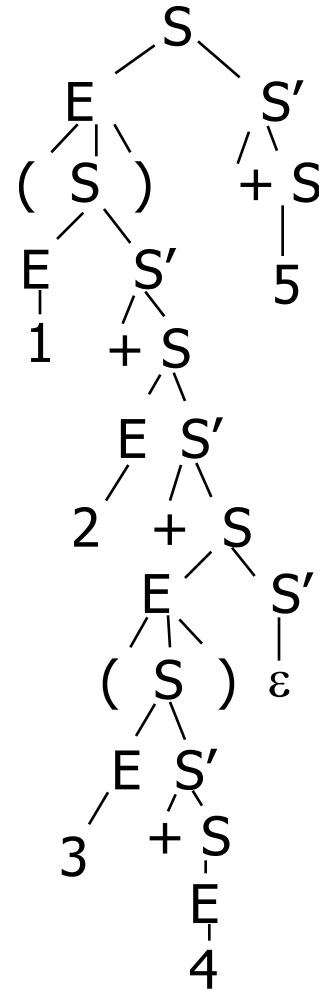
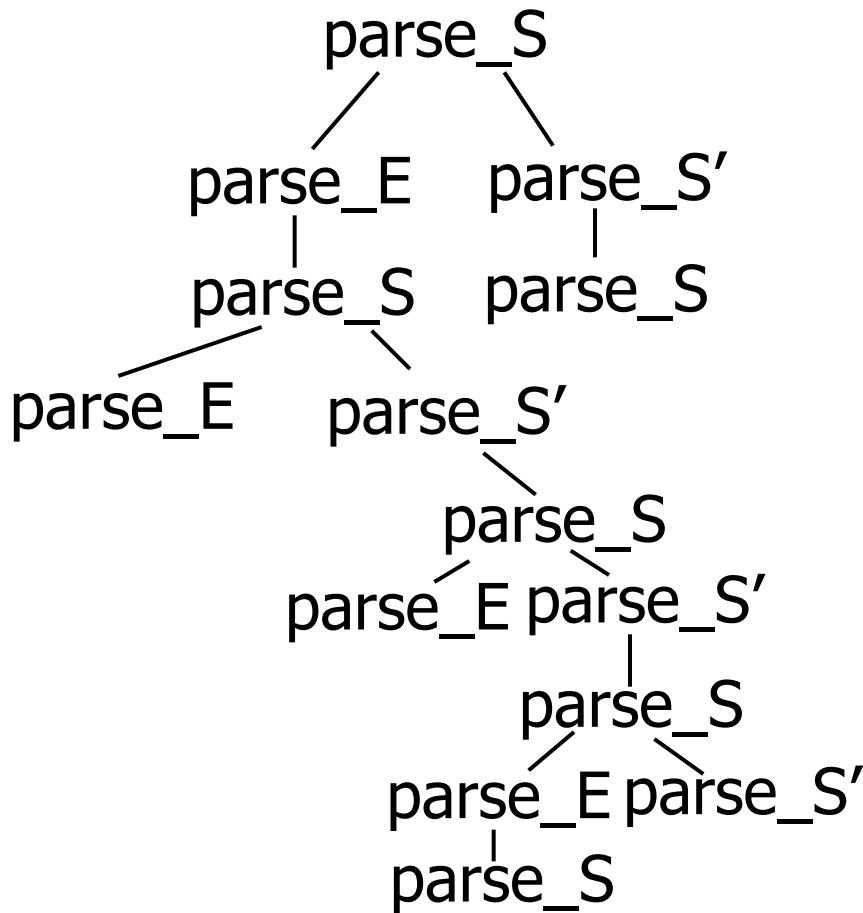
	num	+	()	EOF
S	→ ES'			→ ES'	
→ S'		→ +S		→ ε	→ ε
E	→ num			→ (S)	

Recursive-Descent Parser

```
void parse_E() {  
    switch (token) {  
        case number: token = input.read(); return;  
        case '(': token = input.read(); parse_S();  
            if (token != ')') throw new ParseError();  
            token = input.read(); return;  
        default: throw new ParseError(); }  
}
```

	num	+	()	EOF
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ num		→ (S)		

Call Tree = Parse Tree

$$(1+2+(3+4))+5$$


Constructing parsing tables

$S \rightarrow ES'$
 $S' \rightarrow \varepsilon \mid +S$
 $E \rightarrow \mathbf{num} \mid (S)$



	N	+	()	EOF
S	ES'		ES'		
S'		+S		ε	ε
E	N		(S)		

Parsing table (easy case)

- Grammar
 - has no ϵ -productions
 - every production begins with a terminal symbol
 - (eg) $E \rightarrow aXY \mid bYY \mid stX$
- Easy to fill in parsing table
 - in $\text{Table}[E,a]$ put production $E \rightarrow aXY$ etc.
 - if there are two or more productions in a given spot in table, grammar is not SLL(1)

Generalizing construction (I)

- What if some productions begin with non-terminal (assume no ϵ -productions)?

Example grammar: $S \rightarrow dx \mid Ay$ $A \rightarrow ax \mid bx$

- For what look-ahead symbols should we use $S \rightarrow Ay$?
- Obvious answer: for any terminal $t \in \text{FIRST}(A)$

- Constructing parsing table:

- for each production $A \rightarrow Y_1 Y_2 \dots Y_n$

Enter production into $\text{Table}[A, t]$ for each terminal t in $\text{FIRST}(Y_1)$

Example

- Enter production $A \rightarrow Y_1 \dots Y_n$ into $\text{Table}[A, t]$ for all t for which $t \in \text{FIRST}(Y_1)$

$S \rightarrow A$

$A \rightarrow BC \mid x$

$B \rightarrow t$

$C \rightarrow v$

$\text{FIRST}(A) = \{x, t\}$

$\text{FIRST}(B) = \{t\}$

$\text{FIRST}(C) = \{v\}$

$\text{FIRST}(S) = \{x, t\}$

$\text{Table}[S, x] = S \rightarrow A$

$\text{Table}[S, t] = S \rightarrow A$

$\text{Table}[A, t] = A \rightarrow BC$

$\text{Table}[A, x] = A \rightarrow x$

$\text{Table}[B, t] = B \rightarrow t$

$\text{Table}[C, v] = C \rightarrow v$

Generalizing construction (II)

- Handle ϵ -productions

Example grammar: $S \rightarrow Ax$ $A \rightarrow ab \mid \epsilon$ (A is nullable)

Language has two strings $\{abx, x\}$

$S \rightarrow Ax \rightarrow abx$

$S \rightarrow Ax \rightarrow x$

- Two problems:

- Where should production $A \rightarrow \epsilon$ go in parsing table??

- Insert $A \rightarrow \epsilon$ into $\text{Table}[A,x]$ if x in $\text{FOLLOW}(A)$

- Where should $S \rightarrow Ax$ go in parsing table?

- If we just look at $\text{FIRST}(A)$, we miss the fact that the lookahead symbol might also be x because A is NULLABLE

- Enter production $S \rightarrow Ax$ into $\text{Table}[A,t]$ for all t for which $t \in \text{FIRST}(Ax)$

Constructing parsing table

- Enter production $A \rightarrow Y_1 \dots Y_n$ into $\text{Table}[A, t]$ for all t such that
 - $t \in \text{FIRST}(Y_1 \dots Y_n)$
 - $\varepsilon \in \text{FIRST}(Y_1 \dots Y_n)$ and $t \in \text{FOLLOW}(A)$

- Example

$S \rightarrow A$

$A \rightarrow BC \mid x$

$B \rightarrow t \mid \varepsilon$

$C \rightarrow v \mid \varepsilon$

NULLABLE = {A,B,C}

FIRST(A)={x,t,v, ε }

FIRST(B)={t, ε }

FIRST(C)={v, ε }

FIRST(S)={x,t,v,\$}

FOLLOW(A)={\$}

FOLLOW(B)={v,\$}

FOLLOW(C)={\$}

FOLLOW(S)={\$}

$A \rightarrow BC$ is in table entries for t,v,\$
 $B \rightarrow \varepsilon$ is in table entries for v,\$

$S \rightarrow A$ is in table entries for x,t,v,\$
 $C \rightarrow \varepsilon$ is in table entries for \$

Non-trivial example

- Grammar for arithmetic expressions

$S \rightarrow E$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid \text{int}$

- Grammar is not LL(1)

- Massaged grammar

$S \rightarrow E \$$

$E \rightarrow T E'$

$E' \rightarrow + T E' \mid \epsilon$

$T \rightarrow F T'$

$T' \rightarrow * F T' \mid \epsilon$

$F \rightarrow (E) \mid \text{int}$

- Nullable = $\{E', T'\}$

- FIRST

– $E = \{ (, \text{int} \}$

- FOLLOW

– $E' = \{ \$,) \}$

– $T' = \{ \$,), + \}$

– $S = \{ \$ \}$

	+	*	()	int	\$
S			$S \rightarrow E \$$		$S \rightarrow E \$$	
E			$E \rightarrow T E'$		$E \rightarrow T E'$	
E'	$+ T E'$			$E' \rightarrow \epsilon$		$E' \rightarrow \epsilon$
T			$T \rightarrow F T'$		$T \rightarrow F T'$	
T'	$T' \rightarrow \epsilon$	$T' \rightarrow * F T'$		$T' \rightarrow \epsilon$		$T' \rightarrow \epsilon$
F			$F \rightarrow (E)$		$F \rightarrow \text{int}$	

Non-SLL(1) grammars

- Construction of predictive parse table for grammar results in conflicts

$$S \rightarrow S+S \mid S*S \mid \text{num}$$

$$\text{FIRST}(S+S) = \text{FIRST}(S*S) = \text{FIRST}(\text{num}) = \{ \text{num} \}$$

	num	+	*	ϵ
S	$\rightarrow \text{num}, \rightarrow S+S, \rightarrow S*S$			

Summary

- SLL(k) grammars
 - left-to-right scanning
 - leftmost derivation
 - can determine what production to apply from the next k symbols
 - Can automatically build predictive parsing tables
- Predictive parsers
 - Can be easily built for SLL(1) grammars from the parsing tables
 - Also called recursive-descent, or top-down parsers

Assignment

- For Bali grammar, we can write simple recursive-descent parser that consists of a set of mutually recursive procedures
 - one procedure for each non-terminal in the grammar
 - responsible for reading in a substring and parsing it as that non-terminal
 - body of procedure is a switch/case statement that looks at the next token and decides which production to use for that non-terminal
- Hand-crafted recursive-descent parsers can handle some non-SLL(1) grammars using ad hoc techniques
 - more difficult to do in table-driven approach

Helper class: SamTokenizer

- Read the on-line code for
 - [Tokenizer: interface](#)
 - SamTokenizer: code
- Code lets you
 - open file for input:
 - `SamTokenizer f = new SamTokenizer(String-for-file-name)`
 - examine what the next thing in file is: `f.peekAtKind()` → `TokenType`
 - `TokenType: enum {INTEGER, FLOAT, WORD, OPERATOR,...}`
 - INTEGER: such as 3, -34, 46
 - WORD: such as x, r45, y78z (variable name in Java)
 - OPERATOR: such as +, -, *, (,), etc.
 -
 - read next thing from file (or throw `TokenizerException`):
 - `f.getInt/peekInt ()` → `int`
 - `f.getWord/peekWord:()` → `String`
 - `f.getOp/peekOp:()` → `char`
 - `get` eats up token from file, while `peek` does not advance the pointer into the file

- Useful methods in SamTokenizer class:
 - `f.check(char c): char → boolean`
 - Example: `f.check('*');` //true if next thing in input is *
 - Check if next thing in input is c
 - if so, eat it up and return true
 - otherwise, return false
 - `f.check(String s): String → boolean`
 - Example of its use: `f.check("if");`
 - Check if next word in input matches s
 - if so, eat it up and return true
 - otherwise, return false
 - `f.checkInt(): () → boolean`
 - check if next token is an integer and if so, eat it up and return true
 - otherwise, return false
 - `f.match(char c): char → void`
 - like `f.check` but throws `TokenizerException` if next token in input is not "c"
 - `f.match(String s): string → void`
 - (eg) `f.match("if")`

Recognizer for simple expressions

Expression \rightarrow integer

Expression \rightarrow (Expression + Expression)

- Input: file
- Output: true if a file contains a single expression as defined by this grammar, false otherwise
- Note: file must contain exactly one expression

File: (2+3) (3+4)

will return false

Parser for expression language

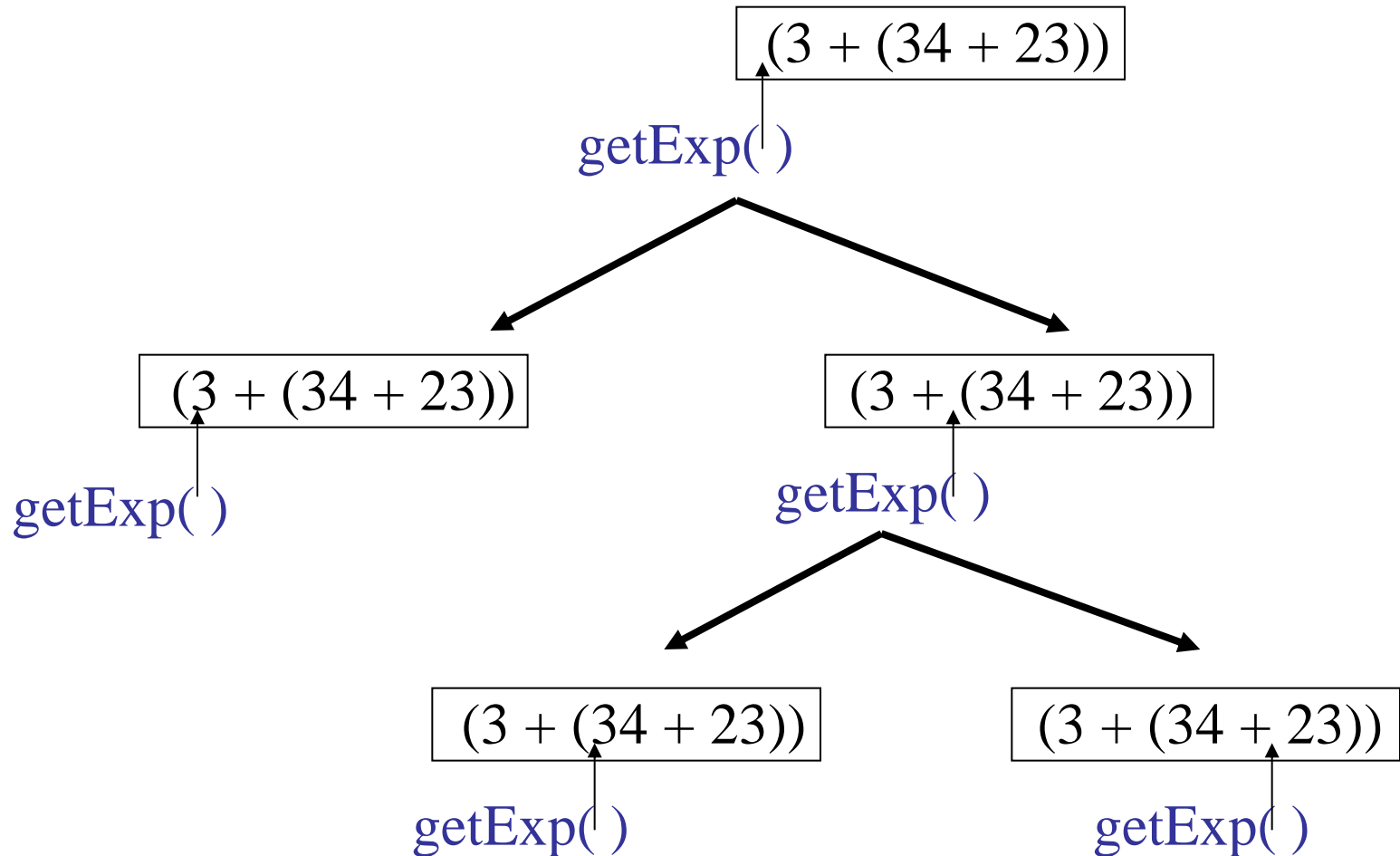
```
static boolean expParser(String fileName) { //parser for expression in file
    try {
        SamTokenizer f = new SamTokenizer (fileName);
        return getExp(f) && (f.peekAtKind() == Tokenizer.TokenType.EOF) ; //must be at EOF
    } catch (Exception e) {
        System.out.println("Aaargh");
        return false;
    }
}

static boolean getExp(SamTokenizer f) {
    switch (f.peekAtKind()) {
        case INTEGER: //E -> integer
            {f.checkInt();
             return true;
            }
        case OPERATOR: //E -> (E+E)
            return f.check('(') && getExp(f) && f.check('+') && getExp(f) && f.check(')');
        default:
            return false;
    }
}
```

Note on boolean operators

- Java supports two kinds of boolean operators:
 - E1 & E2:
 - Evaluate both E1 and E2 and compute their conjunction (i.e., “and”)
 - E1 && E2:
 - Evaluate E1. If E1 is false, E2 is not evaluated, and value of expression is false. If E1 is true, E2 is evaluated, and value of expression is the conjunction of the values of E1 and E2.
- In our parser code, we use &&
 - if “f.check('(') returns false, we simply return false without trying to read anything more from input file. This gives a graceful way to handling errors.

Tracing recursive calls to getExp



Modifying parser to do SaM code generation

- Let us modify the parser so that it generates SaM code to evaluate arithmetic expressions: (eg)

2 : PUSHIMM 2
 STOP

(2 + 3) : PUSHIMM 2
 PUSHIMM 3
 ADD
 STOP

Idea

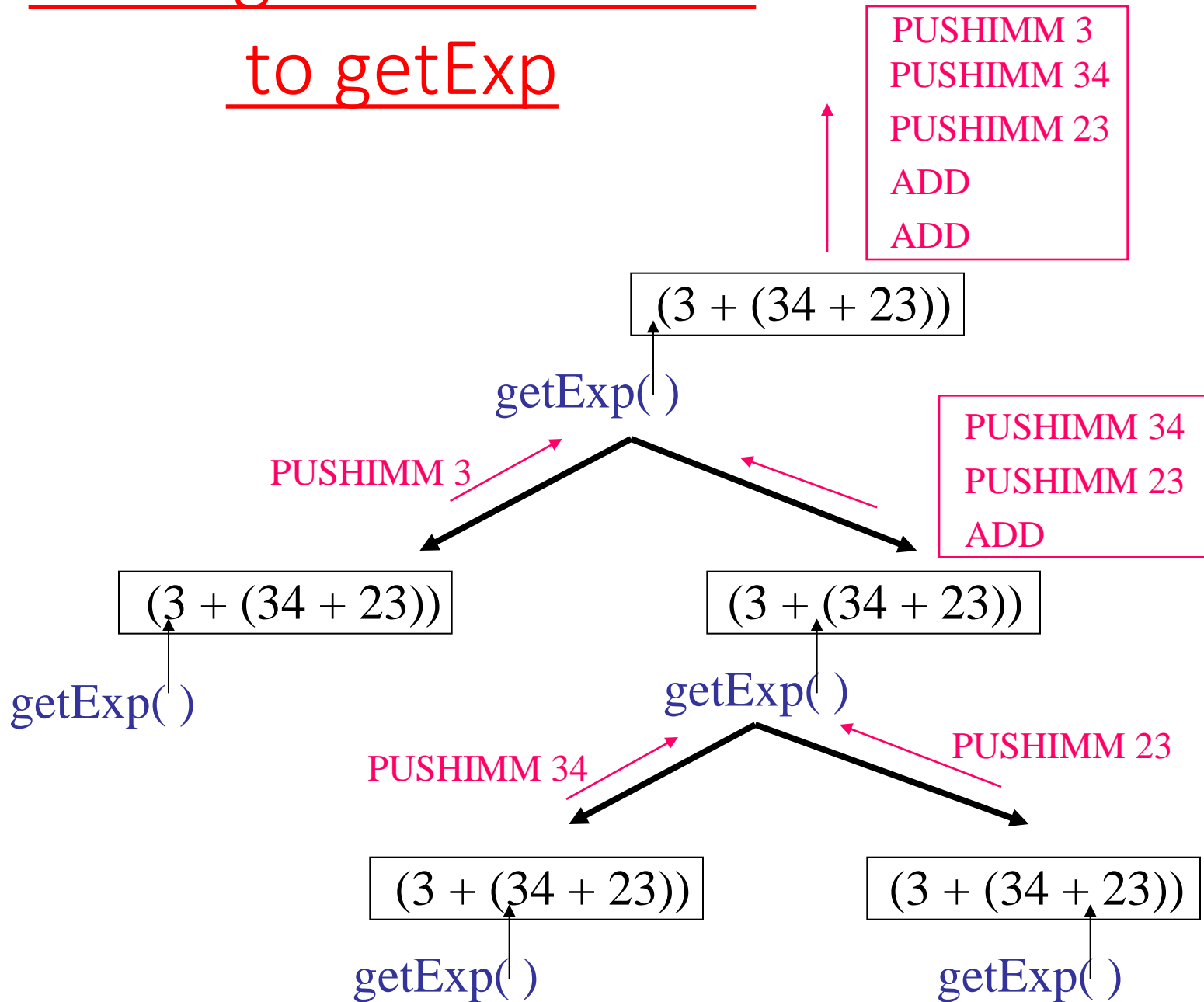
- Recursive method `getExp` should return a string containing SaM code for expression it has parsed.
- Top-level method `expParser` should tack on a `STOP` command after code it receives from `getExp`.
- Method `getExp` generates code in a recursive way:
 - For integer i , it returns string `"PUSHIMM" + i + "\n"`
 - For $(E1 + E2)$,
 - recursive calls return code for $E1$ and $E2$
 - say these are strings $S1$ and $S2$
 - method returns $S1 + S2 + "ADD\n"$

CodeGen for expression language

```
static String expCodeGen(String fileName) { //returns SaM code for expression in file
    try {
        SamTokenizer f = new SamTokenizer (fileName);
        String pgm = getExp(f);
        return pgm + "STOP\n";
    } catch (Exception e) {
        System.out.println("Aaargh");
        return "STOP\n";
    }
}
```

```
static String getExp(SamTokenizer f) {
    switch (f.peekAtKind()) {
        case INTEGER: //E -> integer
            return "PUSHIMM " + f.getInt() + "\n";
        case OPERATOR: //E -> (E+E)
            {
                f.match('('); // must be '('
                String s1 = getExp(f);
                f.match('+'); //must be '+'
                String s2 = getExp(f);
                f.match(')'); //must be ')'
                return s1 + s2 + "ADD\n";
            }
        default: return "ERROR\n";
    }
}
```

Tracing recursive calls to getExp



Top-Down Parsing

- We can use recursive descent to build an abstract syntax tree too

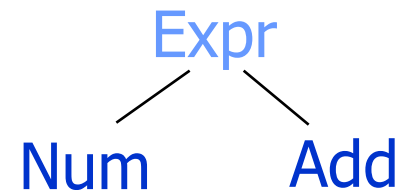
Creating the AST

```
abstract class Expr { }
```

```
class Add extends Expr {  
    Expr left, right;  
    Add(Expr L, Expr R) {  
        left = L; right = R;  
    }  
}
```

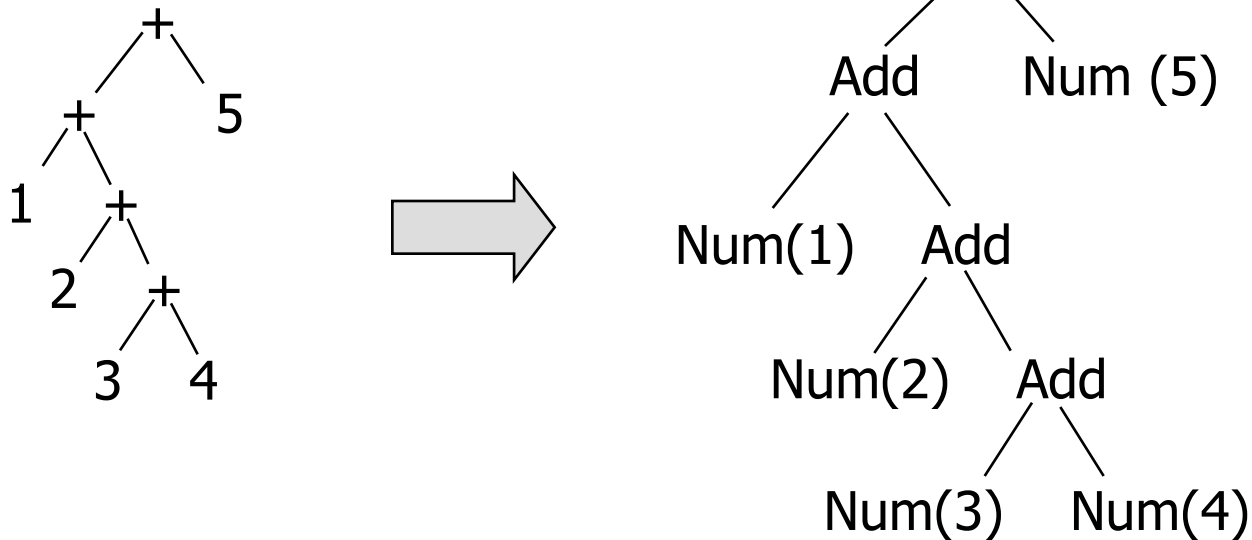
```
class Num extends Expr {  
    int value;  
    Num (int v) { value = v; }  
}
```

Class Hierarchy



AST Representation

$(1 + 2 + (3 + 4)) + 5$



How can we generate this structure during recursive-descent parsing?

Creating the AST

- Just add code to each parsing routine to create the appropriate nodes!
- Works because parse tree and call tree have same shape
- `parse_S`, `parse_S'`, `parse_E` all return an `Expr`:

`void parse_E()`

`void parse_S()`

`void parse_S'()`



`Expr parse_E()`

`Expr parse_S()`

`Expr parse_S'()`

AST Creation: parse_E

```
Expr parse_E() {  
    switch(token) {  
        case num: // E → num  
            Expr result = Num (token.value);  
            token = input.read(); return result;  
        case '(': // E → ( S )  
            token = input.read();  
            Expr result = parse_S();  
            if (token != ')') throw new ParseError();  
            token = input.read(); return result;  
        default: throw new ParseError();  
    }  
}
```

Conclusion

- There is a systematic way of generating parsing tables for recursive-descent parsers
- Recursive descent parsers were among the first parsers invented by compiler writers (Irons 61 for Algol-60)
- Ideally, generate parsers directly from grammar
 - software maintenance would be much easier
 - maintain the “parser-generator” for everyone
 - maintain only the specification of your grammar
- Today we have many parser-generators
 - Javacc, ANTLR: produce recursive-descent parsers from suitable grammars
 - Yacc, bison: bottom-up parsers from LALR(1) grammars
- History of parsing: (somewhat biased perspective!)
<https://jeffreykegler.github.io/Ocean-of-Awareness-blog/individual/2014/09/chron.html>