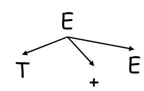
Top-down parsing

Top-down parsing

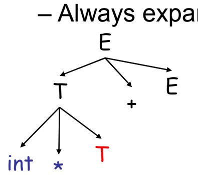
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



int * int + int

Top-down parsing II

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

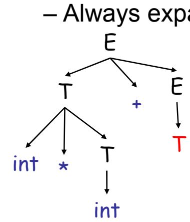


- The leaves at any point form a string $\beta A \gamma$
  - β contains only terminals
  - The input string is $\beta b \delta$
  - The prefix β matches
  - The next token is b

int * int + int

Top-down parsing III

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

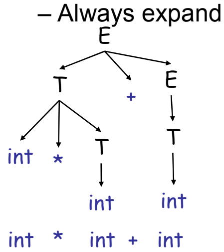


- The leaves at any point form a string $\beta A \gamma$ (A=T, $\gamma=\varepsilon$)
  - β contains only terminals
  - γ contains any symbols
  - The input string is $\beta b \delta$ (b=int)
  - So $A\gamma$ must derive $b\delta$

int * int + int

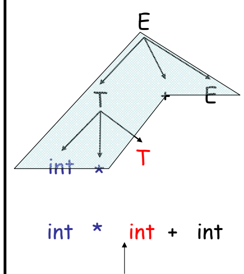
Top-down parsing IV

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



- So choose production for T that can eventually derive something that starts with int

LL(k) parsing



Current sentential form: int * T + E

Look-ahead (1): int
Look-ahead (2): int +
Look-ahead (3): int + int

LL(1) parser: determines next production in leftmost derivation, looking ahead by one terminal
Key question: How do we choose the next production systematically?

Overview

- We will focus on LL(1) parsers.
  - Generalization: LL(k) parsers
- LL(1) parsers require three sets called
  - nullable
  - FIRST
  - FOLLOW
- Given these sets, you can write down a recursive-descent parser
- Simplification
  - nullable and FOLLOW are only required if the grammar has ε productions
- Game plan
  - start with grammars without ε productions
  - then add ε productions
  - end with an iterative, stack-based implementation of top-down parsing

Example 1

- Restriction on grammar:
  - for each non-terminal
    - productions begin with terminals
    - no two productions begin with same terminal
  - so no ε productions
- Algorithm for parsing:
  - one procedure for each non-terminal
  - In each procedure, peek at the next token to determine which rule to apply
- Example:

  S → id := E |if E then S else S |while E do S

```
procedure S
    case peekAtToken() of
       id : match(id); match(:=); E; break;
      if: match(if); E; match(then); S; match(else); S; break;
      while: match (while); E; match(do); S; break;
      otherwise error
```

- Can we describe this more formally to set the stage for more complex grammars?

LL(1) Parsing Table

T	id	:=	if	then	else	do	while
S	id:= E		if E then S else S				while E do S

S → id := E |if E then S else S |while E do S

- Consider the T[S, if] entry
  - Means "When current non-terminal is S and next input token is "if", use production S → if E then S else S"
- Given this table, we can construct the recursive code trivially.
- How do we generate parsing tables automatically?

FIRST sets

- FIRST: non-terminal → subset of terminals
  - b ε FIRST(N) if N →* bδ
- Construction:
  - for each non-terminal A
    - for each rule A → tγ, add constraint: t is in FIRST(A)
  - find smallest sets that satisfy all constraints
- For our example grammar,

  S → id := E |if E then S else S |while E do S

  set of terminals = {id, :=, if, then, else, while, do}

  Constraints:
  - id is in FIRST(S)
  - if is in FIRST(S)
  - while is in FIRST(S)
- There are many sets that satisfy these constraints

  (eg) {id,if,while}, {id,if,while,:=}, {id,if,while,do,:=},….
- We want the smallest set that satisfies all constraints
  - FIRST(S) = {id,if,while}
- Extension: it is convenient to extend FIRST to any string γ:
  - b ε FIRST(γ) if γ →* bδ

Constructing Parsing Tables

- Construct a parsing table T for CFG G
- For each production $A \rightarrow \alpha$ in G do:
  - For each terminal $b \in First(\alpha)$ do
    - T[A, b] = A → α
- Conflict: two or more productions in one table entry
  - Grammar is not LL(1)
  - We may or may not be able to rewrite grammar to be LL(1)

Example 2

- Some productions may begin with non-terminal
- Example:

  S → XY | YX

  X → a b

  Y → b a

  It is clear that we can parse S as follows:

```
procedure S
   case peekAtToken() of
     a: X ; Y
     b: Y ; X
  otherwise error
```

FIRST sets

- Construction: for each non-terminal A
  - for each rule A → tγ, t is in FIRST(A)
  - for each rule A → Bγ, FIRST(B) is a subset of FIRST(A)
- For our example, rules give
  - FIRST(X) is a subset of FIRST(S)
  - FIRST(Y) is a subset of FIRST(S)
  - a is in FIRST(X)
  - b is in FIRST(Y)
- If we solve these constraints, we get
  - FIRST(X) = {a}
  - FIRST(Y) = {b}
  - FIRST(S) = {a,b}

Constructing Parsing Tables

- Same as before
- For each production $A \rightarrow \alpha$ in G do:
  - For each terminal $t \in First(\alpha)$ do
    - T[A, t] = A → α

T	a	b
S	XY	YX
X	a b	
Y		b a

What if a grammar is not LL(1)?

- Table conflicts:
  - two or more productions in some T[A,t]
- Example:
  S → a b | a c
  T[S,a] contains both productions so grammar is not LL(1)
- Some non-LL(1) grammars can be rewritten to be LL(1)
- Example can be left-factored
  S → a S'
  S' → b | c
- When writing recursive parser by hand, you can hack code to avoid left-factoring

```
procedure S
  match(a);
  case input_roken of
    b: match(b);
    c: match(c);
    otherwise error
```

Left-recursion

- Grammar is left-recursive if for some non-terminal A
  A →* Aγ
- Example: lists
  T → L ;
  L → id | L , id
- Grammars can be rewritten to eliminate left-recursion
  T → id R
  R → ; | , id R
- Hack to avoid doing this in code

```
procedure L
   match(id);
   while (input_token == ,) {
      match(,); match(id);
   }
```

ε productions

- Non-terminal N is nullable if N →+ ε
- Example:
  S → AB$
  A → a | ε
  B → b
- When should you use the A → ε production?
- One solution:
  - Ignore ε productions and compute FIRST
  - Table[A,a] = A→a
  - all other entries for A: A → ε
- This is bad practice
  - errors should be caught as soon as possible
  - what if next input token was $?
- Solution:
  - if we use A → ε production to derive a legal string, next token in input must be b
  - if next token is b, use A → ε production; otherwise report error
- How do we describe this formally?

FOLLOW sets

- FOLLOW: Non-terminal → subset of terminals
- b ε FOLLOW(A) if S →* …Ab…
- To compute FOLLOW(A), we must look at RHS of productions that contain A
- Example:
  S → AB$
  A → a | ε
  B → b
- FOLLOW(B) = {$}
- FOLLOW(A) = FIRST(B)
- But ε rules change FIRST computation as well!
  - FIRST(S) needs to take into account the fact that A is nullable
- How do we get all this straight?

Game plan

1. Compute set of nullable non-terminals
2. Use nullable set to compute FIRST
3. Use FIRST to compute FOLLOW
4. Use FIRST and FOLLOW sets to populate LL(1) parsing table

Computing Nullable

- Set up constraints for nullable set of non-terminals as follows:
  - nullable is a subset of non-terminals
  - A → ε
    A is in nullable
  - A → ..t…
    no constraint
  - A→BC..M
    if B,C,…,M are in nullable, then A is in nullable
- Find least set of non-terminals that satisfy all constraints

Example

$Z \to d$	no constraint
$Y \to \varepsilon$	nullable contains Y
$X \to Y$	if nullable contains Y, nullable contains X
$Z \to X\ Y\ Z$	if nullable contains X,Y,Z, nullable contains Z
$Y \to c$	no constraint
$X \to a$	no constraint

So constraints are
nullable contains Y
if nullable contains Y, nullable contains X
if nullable contains X,Y,Z, nullable contains Z

Solution: nullable = {X,Y}

Computing First Sets

Definition First(X) = $\{ b \mid X \to^* b\alpha \}$

1. First(b) = { b } for b any terminal symbol
2. For all productions $X \to A_1 \dots A_n$
   - First(A_1) is a subset of First(X)
   - First(A_2) is a subset of First(X) if A_1 is nullable
   - …
   - First(A_n) is a subset of First(X) if $A_1 \dots A_{n-1}$ are nullable

   Note: $X \to \varepsilon$ does not generate any constraint
3. Solve

Example

$Z \to d$	{d} is a subset of FIRST(Z)
$Y \to \varepsilon$	no constraint
$X \to Y$	FIRST(Y) is a subset of FIRST(X)
$Z \to X\ Y\ Z$	FIRST(X) is a subset of FIRST(Z)
	FIRST(Y) is a subset of FIRST(Z)
	FIRST(Z) is a subset of FIRST(Z)
$Y \to c$	{c} is a subset of FIRST(Y)
$X \to a$	{a} is a subset of FIRST(X)

Solution:
FIRST(X) = {a,c}
FIRST(Y) = {c}
FIRST(Z) = {a,c,d}

Computing Follow Sets

Definition Follow(X) = $\{ b \mid S \to^* \beta\, X\, b\, \omega \}$

1. For all productions $Y \to \dots X A_1 \dots A_n$
   First(A_1) is a subset of Follow(X)
   First(A_2) is a subset of Follow(X) if A_1 is nullable
   …
   First(A_n) is a subset of Follow(X) if $A_1,\dots,A_{n-1}$ are nullable
   Follow(Y) is a subset of Follow(X) if $A_1,\dots,A_n$ are nullable
2. Solve.

Example

Z → d	no constraint
Y → ε	no constraint
X → Y	FOLLOW(X) is a subset of FOLLOW(Y)
Z → X Y Z	FIRST(Y) is a subset of FOLLOW(X) FIRST(Z) is a subset of FOLLOW(X) FIRST(Z) is a subset of FOLLOW(Y)
Y → c	no constraint
X → a	no constraint

Solution:
FOLLOW(X) = {a,c,d}
FOLLOW(Y) = {a,c,d}
FOLLOW(Z) = {}

Computing nullable,FIRST,FOLLOW

```
for each symbol X
   FIRST[X] := { }, FOLLOW[X] := { }, nullable[X] := false

for each terminal symbol t
    FIRST[t] := {t}

repeat
   for each production X → Y1 Y2 … Yk,
         if all Yi are nullable then
            nullable[X] := true
         if Y1..Yi-1 are nullable then
            FIRST[X] := FIRST[X] U FIRST[Yi]
         if Yi+1..Yk are all nullable then
            FOLLOW[Yi] := FOLLOW[Yi] U FOLLOW[X]
         if Yi+1..Yj-1 are all nullable then
           FOLLOW[Yi] := FOLLOW[Yi] U FIRST[Yj]

until FIRST, FOLLOW, nullable do not change
```

Constructing Parsing Table

- For each production $A \to \alpha$ in G do:
  - For each terminal b $\in$ First(α) do
    - T[A, b] = α
  - If A is nullable, for each b $\in$ Follow(A) do
    - T[A, b] = ε

LL(1) Parsing Table Example

E → T X   X → + E | ε
T → (E) | int Y   Y → * T | ε

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

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

First(T) = {int, (}
First(E) = {int, (}
First(X) = {+}
First(Y) = {*}
X and Y are nullable

Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1). This happens
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored
  - *And in other cases as well*
- Most programming language grammars are not LL(1)
- There are tools that build LL(1) tables

Stack-based parser

- We can read off the recursive parser from the parsing table.
- We can also use a stack-based iterative parser that is driven by the parsing table.
- Advantage:
  - smaller space requirements
  - usually faster

LL(1) Parsing Algorithm

initialize stack = <S,$>
repeat
  case stack of
    <X, rest> : if T[X,next()] == T → $Y_1 ... Y_n$:
                  stack ← <$Y_1 ... Y_n$ rest>;
                else: error ();
    <t, rest> : scan (t); stack ← <rest>;
until stack == < >

LL(1) Parsing Example

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

Another picture of LL(1) parsers

- To transition smoothly to bottom-up LR parsers, it is convenient to think about LL(1) parsers as follows:
  - One FSA for each production in grammar
  - FSA symbols are both terminals and non-terminals
  - When FSA1 needs to recognize a non-terminal, it "invokes" the appropriate FSA, saving its own state on stack
  - When that FSA is done, state for FSA1 is popped from stack and it continues
- So at any point in parsing, there may be activated multiple FSA's, although only one will be executing