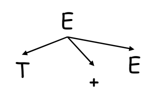
# Top-down parsing

# Top-down parsing

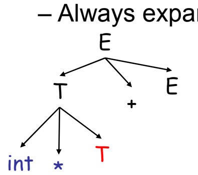
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal




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

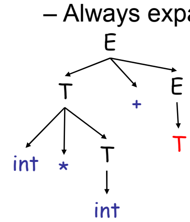



- The leaves at any point form a string $\beta A\gamma$
  - $\beta$ contains only terminals
  - The input string is $\beta b\delta$
  - The prefix $\beta$ matches
  - The next token is $b$

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

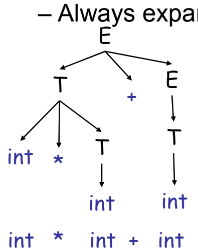



- The leaves at any point form a string $\beta A\gamma$ (A=T, $\gamma=\varepsilon$)
  - $\beta$ contains only terminals
  - $\gamma$ contains any symbols
  - The input string is $\beta b\delta$ (b=int)
  - So $A\gamma$ must derive $b\delta$

int * int + int

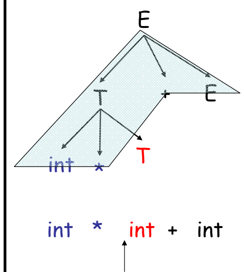
# Top-down parsing IV

- 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 (we saw this informally)
  - 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
```

# 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 $\in$ FIRST(N) if N →* b$\delta$
- Construction:
  - for each non-terminal A
    - for each rule A → t$\gamma$, 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 $\in$ FIRST(S)
  - if $\in$ FIRST(S)
  - while $\in$ 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 $\gamma$:
  - b $\in$ FIRST($\gamma$) if $\gamma$ →* b$\delta$

# Constructing Parsing Tables

- Construct a parsing table T for CFG G
- For each production A → $\alpha$ in G do:
  - For each terminal b $\in$ First($\alpha$) do
    - T[A, b] = A → $\alpha$
- 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 ∈ FIRST(A)
  - for each rule A → Bγ, FIRST(B) ⊆ FIRST(A)
- For our example, rules give
  - FIRST(X) ⊆ FIRST(S)
  - FIRST(Y) ⊆ FIRST(S)
  - a ∈ FIRST(X)
  - b ∈ 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 → α in G do:
  - For each terminal t ∈ First(α) 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 ⊆ Non-terminals
  - A → ε
    A ∈ Nullable
  - A → ..t…
    no constraint
  - A→BC..M
    if B,C,…,M ∈ Nullable, then A ∈ Nullable
- Find least set that satisfies all constraints

## Example

| | |
|---|---|
| $Z \rightarrow d$ | no constraint |
| $Y \rightarrow \varepsilon$ | $Y \in$ Nullable |
| $X \rightarrow Y$ | if $Y \in$ Nullable, $X \in$ Nullable |
| $Z \rightarrow X\ Y\ Z$ | if $X,Y,Z \in$ Nullable, $Z \in$ Nullable |
| $Y \rightarrow c$ | no constraint |
| $X \rightarrow a$ | no constraint |

So constraints are
$Y \in$ Nullable
if $Y \in$ Nullable then $X \in$ Nullable
if $X,Y,Z \in$ Nullable then $Z \in$ Nullable
Solution: nullable = {X,Y}

## Computing First Sets

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

1. First(b) = { b } for b any terminal symbol
2. For all productions $X \rightarrow A_1 \ldots A_n$
   - $First(A_1) \subseteq First(X)$
   - $First(A_2) \subseteq First(X)$ if $A_1 \in$ Nullable
   - …
   - $First(A_n) \subseteq First(X)$ if $A_1 \ldots A_{n-1} \in$ Nullable

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

## Example

| | |
|---|---|
| $Z \rightarrow d$ | $\{d\} \subseteq FIRST(Z)$ |
| $Y \rightarrow \varepsilon$ | no constraint |
| $X \rightarrow Y$ | $FIRST(Y) \subseteq FIRST(X)$ |
| $Z \rightarrow X\ Y\ Z$ | $FIRST(X) \subseteq FIRST(Z)$ |
| | $FIRST(Y) \subseteq FIRST(Z)$ |
| | $FIRST(Z) \subseteq FIRST(Z)$ |
| $Y \rightarrow c$ | $\{c\} \subseteq FIRST(Y)$ |
| $X \rightarrow a$ | $\{a\} \subseteq FIRST(X)$ |

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

## Computing Follow Sets

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

1. For all productions $Y \rightarrow \ldots X A_1 \ldots A_n$

   $First(A_1) \subseteq Follow(X)$
   $First(A_2) \subseteq Follow(X)$ if $A_1 \in$ nullable
   …
   $First(A_n) \subseteq Follow(X)$ if $A_1,..,A_{n-1} \in$ nullable
   $Follow(Y) \subseteq Follow(X)$ if $A_1,..,A_n \in$ nullable

2. Solve.

# Example

| | |
|---|---|
| Z → d | no constraint |
| Y → ε | no constraint |
| X → Y | FOLLOW(X) ⊆ FOLLOW(Y) |
| Z → X Y Z | FIRST(Y) ⊆ FOLLOW(X) |
| | FIRST(Z) ⊆ FOLLOW(X) |
| | FIRST(Z) ⊆ 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($\alpha$) do
    - T[A, b] = $\alpha$
  - If $\alpha$ is nullable, for each $b \in$ Follow(A) do
    - T[A, b] = $\alpha$

# 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)
- We can produce the recursive parser systematically from the parsing table.

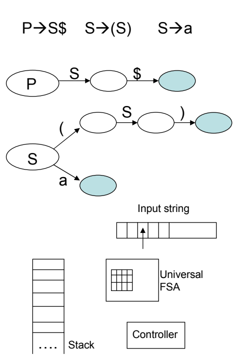
# Iterative LL(1) parser

- It is also possible to design an iterative parser that uses an explicit stack and
  - pushes and pops stuff from the stack
  - examines token from input

  to decide how to parse the program.
- Useful to study this to make a connection with bottom-up parsing, which are always presented using an iterative parser.

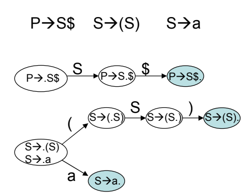
# Pushdown automata

- Here's one way of thinking about context-free grammars and parsing
  - write down a "transition diagram" for each production (note that it has transitions labeled with non-terminals)
  - the pushdown automaton begins execution with the transition diagram for the start symbol by pushing that state on the stack
  - as long as the states it encounters have transitions labeled with terminals, it behaves just like a real FSA
  - however, when it encounters a transition labeled with a non-terminal (say N), it begins execution with the "transition diagram" for N by pushing the start state of that transition diagram on the stack
  - when the transition diagram for N reaches an accepting state, it is popped from the stack, and previous transition diagram continues execution by taking an N transition
  - the string is accepted if the pushdown automaton reaches the end of the input, and the stack only contains the final state for the transition diagram of the start symbol




# Transition diagram

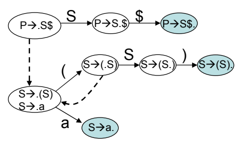
- Convenient to label states using productions with dots to show how far parsing has gotten
  - (eg) P→S.$: we have seen S and we are expecting to see a $



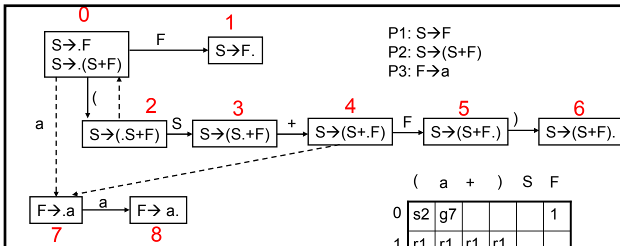
## Building an iterative LL(1) parser

- Draw dashed arrows as shown to denote the pushdown of state
  - these would have been procedure calls in the recursive code
- Now you can just number the states and perform combinations of
  - eat one token from input
  - push a new state on the pushdown stack
  - topmost transition diagram accepts a substring of input

P→S$  S→(S)  S→a




---

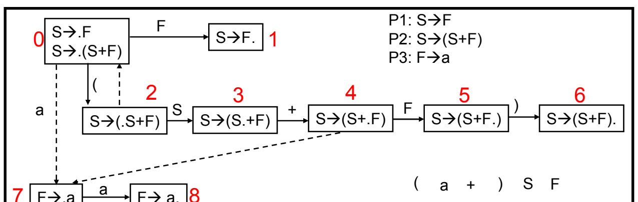



P1: S→F
P2: S→(S+F)
P3: F→a

| | ( | a | + | ) | S | F |
|---|---|---|---|---|---|---|
| 0 | s2 | g7 | | | | 1 |
| 1 | r1 | r1 | r1 | r1 | | |
| 2 | g0 | g0 | g0 | g0 | 3 | |
| 3 | | | s4 | | | |
| 4 | g7 | g7 | g7 | g7 | | 5 |
| 5 | | | | s6 | | |
| 6 | r2 | r2 | r2 | r2 | | |
| 7 | | s8 | | | | |
| 8 | r3 | r3 | r3 | r3 | | |

Actions:
sn : eat one token from input and go to state n
gn: push state n
rm: pop topmost state on stack
 let new topmost state on stack be s
 let non-terminal of production m be N
 replace topmost state on stack by
 state T[s,N]

---




P1: S→F
P2: S→(S+F)
P3: F→a

| Stack | Input |
|---|---|
| 0 | (a+a) |
| 2 | a+a) |
| 2 0 | a+a) |
| 2 0 7 | a+a) |
| 2 0 8 | +a) |
| 2 1 | +a) |
| 3 | +a) |
| 4 | a) |
| 4 7 | a) |
| 4 8 | ) |
| 5 | ) |
| 6 | |
| empty | empty |

| | ( | a | + | ) | S | F |
|---|---|---|---|---|---|---|
| 0 | s2 | g7 | | | | 1 |
| 1 | r1 | r1 | r1 | r1 | | |
| 2 | g0 | g0 | g0 | g0 | 3 | |
| 3 | | | s4 | | | |
| 4 | g7 | g7 | g7 | g7 | | 5 |
| 5 | | | | s6 | | |
| 6 | r2 | r2 | r2 | r2 | | |
| 7 | | s8 | | | | |
| 8 | r3 | r3 | r3 | r3 | | |

---

## Summary

- Given an LL(1) grammar, you can
  - generate parsing table for grammar
    - compute NULLABLE, FIRST, FOLLOW
  - write a recursive-descent parser from that table, using template
- LL(1) parser-generator
  - given LL(1) grammar
    - computes NULLABLE, FIRST, FOLLOW sets
    - uses those sets and transition diagram of grammar to produce an iterative parser that maintains an explicit stack
  - examples: ANTLR, JAVACC

## Iterative parser

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