Top down & intro to bottom-up parsers

Table driven LL(1) parsing
Intro to bottom-up parsing

The content is mostly copied from
https://web.stanford.edu/class/cs143/lectures/lecture07.pdf
https://web.stanford.edu/class/cs143/lectures/lecture08.pdf
Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 3
https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Summary of recursive descent

Predictive parsing

Table driven top-down parsing

Intro to bottom up parsing

Parser Family Tree

```
Parsing Strategies
│
├── Top-Down (LL)
│   ├── Recursive Descent (hand-written)
│   └── Predictive Parsing (LL(1), LL(k))
│
└── Bottom-Up (LR)
    ├── Shift-Reduce
    ├── LR(0), SLR(1)
    ├── LR(1), LALR(1)
    ├── Parser Generators (Bison)
```

Parser Roadmap: Where We're going

```
Top-Down Parsing (Summary in This Lecture)
├── Recursive Descent (hand-written)
├── Predictive Parsing (LL(1))
└── Table-driven LL(1)
```

```
Bottom-Up Parsing (intro in this lecture)
├── Shift-Reduce Parsing
├── LR Parsing (LR(0), SLR, LR(1))
└── Parser Generators (Bison/Yacc)
```

Semantic actions (syntax-directed translation)

Semantic actions can be used to build ASTs

• And many other things as well

– Also used for type checking, code generation,computation, …

Process is called **syntax-directed translation (SDT)**

– Substantial generalization over CFGs

https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

Annotating grammar with actions

Consider the grammar

$E \rightarrow int \mid E + E \mid (E)$

For each symbol X define an attribute X.val

- For terminals,
  - val is the associated lexeme
- For non-terminals,
  - val is the expression's value (and is computed from values of subexpressions)

We annotate the grammar with actions:

E → int { E.val = int.val }

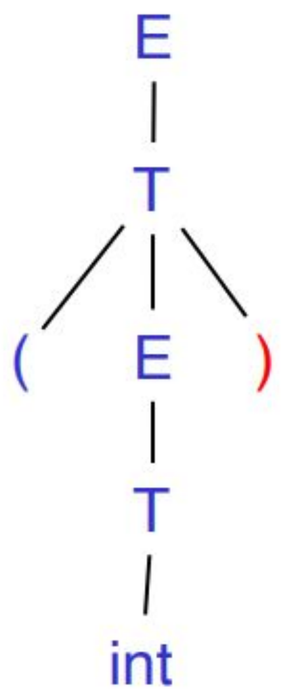
| E1 + E2 { E.val = E1.val + E2.val }

| (E1) { E.val = E1.val }

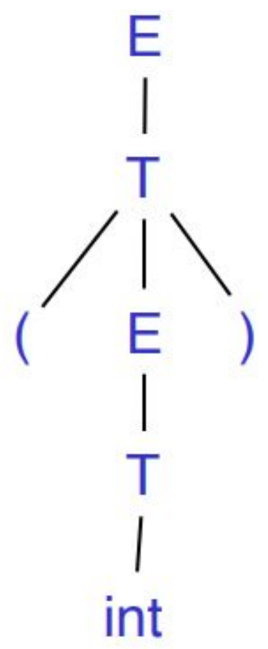
https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

Recursive descent

- Match:)
- Advance input
    - Move



- End of input, accept!



https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

Recursive descent

Grammar

```
stmt ::= id = exp ;
| return exp ;
| if ( exp ) stmt
| while ( exp ) stmt
```

```
// parse stmt ::= id=exp; | …
void stmt( ) {
    switch(nextToken) {
    RETURN:
        returnStmt();
        break;
    IF:
        ifStmt();
        break;
    WHILE:
        whileStmt();
        break;
    ID:
        assignStmt();
        break;
    }
}
```

https://courses.cs.washington.edu/courses/cse401/22au/lectures/F-ll-parsing.pdf

Recursive descent

```
// parse while (exp) stmt
void whileStmt() {
   // skip “while” “(”
   getNextToken();
   getNextToken();
   // parse condition
   exp();
   // skip “)”
   getNextToken();
   // parse stmt
   stmt();
}
```

```
// parse return exp ;
void returnStmt() {
   // skip “return”
   getNextToken();
   // parse expression
   exp();
   // skip “;”
   getNextToken();
}
```

https://courses.cs.washington.edu/courses/cse401/22au/lectures/F-ll-parsing.pdf

Possible problems: left recursion

expr ::= expr + term

| term

```
// parse expr ::= ...
void expr() {
  expr();
  if (current token is PLUS) {
     getNextToken();
     term();
  }
}
```

FIXED: Transform grammar first

expr ::= term { + term }*

```
// parse
void expr() {
   term();
   while (next symbol is PLUS) {
       getNextToken();
       term();
   }
}
```

https://courses.cs.washington.edu/courses/cse401/22au/lectures/F-ll-parsing.pdf

Another problem left factoring

```
ifStmt ::= if ( expr ) stmt
         | if ( expr ) stmt else stmt
```

Formal solution: Factor the common prefix into a separate production

```
Factored grammar
ifStmt ::= if ( expr ) stmt ifTail
ifTail ::= else stmt | ε
```

https://courses.cs.washington.edu/courses/cse401/22au/lectures/F-ll-parsing.pdf

Backtrack free parsing

The major source of inefficiency in the leftmost, top-down parser arises from its need to backtrack

- on the mismatch, it must undo the actions

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr*$'$
2	*Expr*$'$	$\rightarrow$	+ *Term Expr*$'$
3		|	- *Term Expr*$'$
4		|	ϵ
5	*Term*	$\rightarrow$	*Factor Term*$'$
6	*Term*$'$	$\rightarrow$	x *Factor Term*$'$
7		|	÷ *Factor Term*$'$
8		|	ϵ
9	*Factor*	$\rightarrow$	(*Expr*)
10		|	num
11		|	name

Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 3

Using a lookahead symbol

- For this grammar, the parser can avoid backtracking by
  - considering both the focus symbol and the next input symbol(lookahead symbol).
- Using one symbol lookahead, the parser can disambiguate all of the choices that arise in parsing the right-recursive expression grammar.
  - the grammar is backtrack free with a lookahead of one symbol

0	$Goal$	$\rightarrow$	$Expr$
1	$Expr$	$\rightarrow$	$Term\ Expr'$
2	$Expr'$	$\rightarrow$	+ $Term\ Expr'$
3		|	- $Term\ Expr'$
4		|	ϵ
5	$Term$	$\rightarrow$	$Factor\ Term'$
6	$Term'$	$\rightarrow$	x $Factor\ Term'$
7		|	÷ $Factor\ Term'$
8		|	ϵ
9	$Factor$	$\rightarrow$	($Expr$)
10		|	num
11		|	name

Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 3

First/Follow Sets

For each grammar symbol α,

first(α)

- the set of terminal symbols that can appear as the first word in some string derived from **α**

If **α** is either a terminal, **ε**, or **eof**,

- then **first(α)** is **{α}**.

For a nonterminal A,

- **first(A)** contains the complete set of terminal symbols that can appear as the leading symbol in a sentential form derived from A.

```
Step 1: FIRST(terminal) = {terminal}

Step 2: For A → α:

     - If α starts with terminal t: add t to FIRST(A)

     - If α starts with non-terminal B: add FIRST(B) to FIRST(A)

     - If α can be empty: add ε to FIRST(A)
```

Grammar:

$E \rightarrow T\ X$

$X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow \text{int}\ Y \mid (\ E\ )$

$Y \rightarrow *\ T \mid \varepsilon$

Compute FIRST(E):

1. $E \rightarrow T\ X$, so FIRST(E) = FIRST(T)
2. $T \rightarrow \text{int}\ Y \mid (\ E\ )$, so FIRST(T) = {int, (}
3. ∴ FIRST(E) = {int, (}

example

0	*Goal*	$\rightarrow$	*Expr*	
1	*Expr*	$\rightarrow$	*Term Expr'*	
2	*Expr'*	$\rightarrow$	+ *Term Expr'*	
3		|	- *Term Expr'*	
4		|	ϵ	
5	*Term*	$\rightarrow$	*Factor Term'*	
6	*Term'*	$\rightarrow$	x *Factor Term'*	
7		|	÷ *Factor Term'*	
8		|	ϵ	
9	*Factor*	$\rightarrow$	(*Expr*)	
10		|	num	
11		|	name	

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST	num	name	+	-	x	÷	(	)	eof	ϵ

	Expr	***Expr'***	***Term***	***Term'***	***Factor***
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 3

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ϵ
5	*Term*	$\rightarrow$	*Factor Term'*

6	*Term'*	$\rightarrow$	x *Factor Term'*
7		|	÷ *Factor Term'*
8		|	ϵ
9	*Factor*	$\rightarrow$	(*Expr*)
10		|	num
11		|	name

first(ε) = {ε}

- matches no word returned by the scanner.

after a valid application of rule 4

★ parser needs to know which words can appear as the leading symbol

★ We need the set of symbols that can follow an **Expr'**

Computing Follow Sets for a Grammar G

```
FOLLOW(A) is the set of terminals that can come after non-terminal A
FOLLOW(S) = {$} where S is the start symbol.

Repeat:
   if A → αBβ then:
       add FIRST(β) (excepting ε) to FOLLOW(B).
   if A → αB or FIRST(β) contains ε then:
       add FOLLOW(A) to FOLLOW(B).
until no more changes occur.
```

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Follow sets of nonterminals

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr′*
2	*Expr′*	$\rightarrow$	+ *Term Expr′*
3		|	- *Term Expr′*
4		|	ϵ
5	*Term*	$\rightarrow$	*Factor Term′*
6	*Term′*	$\rightarrow$	x *Factor Term′*
7		|	÷ *Factor Term′*
8		|	ϵ
9	*Factor*	$\rightarrow$	(*Expr*)
10		|	num
11		|	name

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FOLLOW	eof,)	eof,)	eof, +, -,)	eof, +, -,)	eof, +, -, x, ÷,)

Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 3

Using first and follow

For productions

$$\text{FIRST}^+(A \rightarrow \beta) = \begin{cases} \text{FIRST}(\beta) & \text{if } \epsilon \notin \text{FIRST}(\beta) \\ \text{FIRST}(\beta) \cup \text{FOLLOW}(A) & \textit{otherwise} \end{cases}$$

In backtrack free grammar, any nonterminal A $A \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

$$\text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \emptyset, \ \forall \ 1 \leq i,j \leq n, \ i \neq j.$$

0	*Goal*	$\rightarrow$	*Expr*	6	*Term'*	$\rightarrow$	$\times$ *Factor Term'*
1	*Expr*	$\rightarrow$	*Term Expr'*	7		|	$\div$ *Factor Term'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*	8		|	ϵ
3		|	- *Term Expr'*	9	*Factor*	$\rightarrow$	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	$\rightarrow$	*Factor Term'*	11		|	name

only productions 4 and 8 have different **first+** and **first** sets

	Production	FIRST set	FIRST$^+$ set
4	$Expr' \rightarrow \epsilon$	$\{\epsilon\}$	$\{\epsilon$, eof,)$\}$
8	$Term' \rightarrow \epsilon$	$\{\epsilon\}$	$\{\epsilon$, eof, +, -,)$\}$

Eliminating common prefixes (left factoring)

transform these productions to create disjoint first+ sets.

11	*Factor*	$\rightarrow$	name
12		|	name [*ArgList*]
13		|	name (*ArgList*)
15	*ArgList*	$\rightarrow$	*Expr MoreArgs*
16	*MoreArgs*	$\rightarrow$	, *Expr MoreArgs*
17		|	ϵ

11	*Factor*	$\rightarrow$	name *Arguments*
12	*Arguments*	$\rightarrow$	[*ArgList*]
13		|	(*ArgList*)
14		|	ϵ

Take a non terminal

$$A \rightarrow \alpha\beta1 \mid \alpha\beta2 \mid \cdot\cdot\cdot \mid \alpha\beta n$$
$$\mid \gamma1 \mid \gamma2 \mid \cdot\cdot\cdot \mid \gamma j$$

Rewrite the original as

$$A \rightarrow \alpha\ B \mid \gamma1 \mid \gamma2 \mid \cdot\cdot\cdot \mid \gamma j$$
$$B \rightarrow \beta1 \mid \beta2 \mid \cdot\cdot\cdot \mid \beta n$$

https://web.stanford.edu/class/cs143/lectures/lecture06.pdf

Summary

Backtrack-free grammars lend themselves to simple and efficient parsing with a recursive descent

By using first, first+ and follow sets we can generate **predictive top down-parser(LL(1) parser)**

Predictive top-down parser in general

Like recursive-descent but parser can "predict" which production to use

– By looking at the next few tokens

– No backtracking

Predictive parsers accept LL(k) grammars

– L means "left-to-right" scan of input

– L means "leftmost derivation"

– k means "predict based on k tokens of lookahead"

– In practice, LL(1) is used

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Recursive Descent vs. LL(1)

In recursive-descent,

– At each step, many choices of production to use

– Backtracking used to undo bad choices

In LL(1),

– At each step, only one choice of production

- When a non-terminal **A** is leftmost in a derivation
- And the next input symbol is **t**
- There is a unique production **A** → **α** to use

– Or no production to use (an error state)

• **LL(1) is a recursive descent variant without backtracking**

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Predictive Parsing and Left Factoring

Recall the 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

- We need to left-factor the grammar

Factor out common prefixes of productions

$E \rightarrow T\ X$

$X \rightarrow + E \mid \varepsilon$

$T \rightarrow int\ Y \mid (E)$

$Y \rightarrow * T \mid \varepsilon$

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

E → T X

X → + E | ε

T → int Y | (E)

Y → * T | ε

- The LL(1) parsing table:

next input token

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

leftmost non-terminal

rhs of production to use

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

E → T X

X → + E | ε

T → int Y | (E)

Y → * T | ε

[E, int] entry

When current

- non-terminal is **E**
- and next input is **int**,

use production E → T X

– This can generate an **int** in the first position

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

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

LL(1) Parsing Tables. Errors

[Y,+] entry

– "When current non-terminal is Y and current token is +, get rid of Y"

– **Y** can be followed by **+** only if **Y → ε**

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

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

LL(1) Parsing Tables. Errors

Blank entries indicate error situations

[Y,(] entry
– "There is no way to derive a string starting with **(** from non-terminal **Y**"

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

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Using parsing tables

Method similar to recursive descent, except

– For the leftmost non-terminal S

– We look at the next input token a

– And choose the production shown at [S,a]

• A **stack** records **frontier** of parse tree

– Non-terminals that have yet to be expanded

– Terminals that have yet to matched against the input

– Top of stack =

- **leftmost pending terminal**
- **or non-terminal**

• Reject on reaching error state

• Accept on end of input & empty stack

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

LL(1) parsing algorithm with table

```
initialize stack = <S $> and next
repeat
   case stack of
   <X, rest> : if T[X,*next] = Y1…Yn
            then stack ← <Y1…Yn, rest>;
            else error ();
   <t, rest> : if t == *next ++
            then stack ← <rest>;
            else error ();
until stack == < >
```

$ marks bottom of stack

For non-terminal X on top of stack, lookup production

Pop X, push production rhs on stack. Note leftmost symbol of rhs is on top of the stack.

For terminal t on top of stack, check t matches next input token

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

LL(1) parsing example

$E \rightarrow T\ X$

$X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow \text{int}\ Y \mid (\ E\ )$

$Y \rightarrow *\ T \mid \varepsilon$

Stack	Input	Action
E $	int * int $	
	$	ACCEPT

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

LL(1) parsing example

E → T X

X → + E | ε

T → int Y | (E)

Y → * T | 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

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Constructing Parsing Tables: The Intuition

Consider non-terminal A and token t production
$A \rightarrow \alpha$,

Add **$T[A,t] = \alpha$**

1. if **$A \rightarrow \alpha \rightarrow^* t\ \beta$**

- α can derive a t in the first position
- $t \in \text{First}(\alpha)$

2. If **$A \rightarrow \alpha \rightarrow^* \varepsilon$** and **$S \rightarrow^* \gamma\ A\ t\ \delta$**

- 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 deriving ε)
  - Can work only if $t \in \text{Follow}(A)$

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Computing first set

Definition

$\text{First}(X) = \{\ t \mid X \rightarrow^* t\alpha\} \cup \{\varepsilon \mid X \rightarrow^* \varepsilon\}$

Algorithm sketch:

1. $\text{First}(t) = \{\ t\ \}$

2. $\varepsilon \in \text{First}(X)$
   - if $X \rightarrow \varepsilon$ or
   - if $X \rightarrow A1 \ldots An$ and $\varepsilon \in \text{First}(Ai)$ for all $1 \leq i \leq n$

3. $\text{First}(\alpha) \subseteq \text{First}(X)$
   - if $X \rightarrow \alpha$ or
   - if $X \rightarrow A1 \ldots An\ \alpha$ and $\varepsilon \in \text{First}(Ai)$ for all $1 \leq i \leq n$

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Computing First sets

```
For Terminals:
   For each terminal a ∈ Σ: FIRST(a) = {a}

For Non-Terminals:
Repeat:
   For each rule X → Y1Y2...Yk in a grammar G:
        if a is in FIRST(Y1) OR a is in FIRST(Yn) and Y1...Yn-1 ⇒ ε
          then add a to FIRST(X)
        if Y1...Yk ⇒ ε
          then add ε to FIRST(X).
until no more changes occur
```

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Example find first sets

Terminals?

Nonterminals?

$E \rightarrow T\ X$

$X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow \text{int}\ Y \mid (\ E\ )$

$Y \rightarrow *\ T \mid \varepsilon$

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Example first sets (solution)

$E \rightarrow T\ X$

$X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow \text{int}\ Y \mid (\ E\ )$

$Y \rightarrow *\ T \mid \varepsilon$

```
Terminals

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

Nonterminals

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

Example Find first-sets

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ϵ
5	*Term*	$\rightarrow$	*Factor Term'*
6	*Term'*	$\rightarrow$	x *Factor Term'*
7		|	÷ *Factor Term'*
8		|	ϵ
9	*Factor*	$\rightarrow$	(*Expr*)
10		|	num
11		|	name

example

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ϵ
5	*Term*	$\rightarrow$	*Factor Term'*
6	*Term'*	$\rightarrow$	x *Factor Term'*
7		|	÷ *Factor Term'*
8		|	ϵ
9	*Factor*	$\rightarrow$	(*Expr*)
10		|	num
11		|	name

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST	num	name	+	-	x	÷	(	)	eof	ϵ

	Expr	***Expr'***	***Term***	***Term'***	***Factor***
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

Follow sets

Definition:

Follow(X) = { t | S →* βXtδ }

Intuition

- if X → AB
  - then **First(B) ⊆ Follow(A)** and **Follow(X) ⊆ Follow(B)**
  - if B →*ε
    - then **Follow(X) ⊆ Follow(A)**

- if S is the start symbol then **$ ∈ Follow(S)**

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Follow set algorithm sketch

1. $\$ \in \mathrm{Follow}(S)$

2. For each production $A \rightarrow \alpha X\beta$
- $\mathrm{First}(\beta) - \{\varepsilon\} \subseteq \mathrm{Follow}(X)$

3. For each production $A \rightarrow \alpha X\beta$ where $\varepsilon \in \mathrm{First}(\beta)$
- $\mathrm{Follow}(A) \subseteq \mathrm{Follow}(X)$

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Computing Follow Sets for a Grammar G

```
FOLLOW(A) is the set of terminals that can come after non-terminal A
FOLLOW(S) = {$} where S is the start symbol.

Repeat:
   if A → αBβ then:
       add FIRST(β) (excepting ε) to FOLLOW(B).
   if A → αB or FIRST(β) contains ε then:
       add FOLLOW(A) to FOLLOW(B).
until no more changes occur.
```

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Example follow sets

$E \rightarrow T\ X$

$X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow \text{int}\ Y \mid (\ E\ )$

$Y \rightarrow *\ T \mid \varepsilon$

Follow sets?

$\$ \in \text{Follow}(E)$

$\text{First}(X) \subseteq \text{Follow}(T)$

$\text{Follow}(E) \subseteq \text{Follow}(X)$

$\text{Follow}(E) \subseteq \text{Follow}(T)$

$) \in \text{Follow}(E)$

$\text{Follow}(T) \subseteq \text{Follow}(Y)$

$\text{Follow}(X) \subseteq \text{Follow}(E)$

$\text{Follow}(Y) \subseteq \text{Follow}(T)$

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Example follow sets

$E \rightarrow T\ X$

$X \rightarrow +\ E \mid \varepsilon$

$T \rightarrow \text{int}\ Y \mid (\ E\ )$

$Y \rightarrow *\ T \mid \varepsilon$

Follow(+) = { int, (}

Follow(*) = { int, (}

Follow(( ) = { int, (}

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

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

Follow(E) = {), $}

Follow(X) = {$,) }

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

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

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Follow sets of nonterminals

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ϵ
5	*Term*	$\rightarrow$	*Factor Term'*
6	*Term'*	$\rightarrow$	x *Factor Term'*
7		|	÷ *Factor Term'*
8		|	ϵ
9	*Factor*	$\rightarrow$	(*Expr*)
10		|	num
11		|	name

	Expr	*Expr'*	*Term*	*Term'*	*Factor*
FOLLOW	eof,)	eof,)	eof, +, -,)	eof, +, -,)	eof, +, -, x, ÷,)

Algorithm for Constructing LL(1) Parsing Tables

Construct a parsing table T for CFG G

- For each production $A \to \alpha$ in G do:
  - For each terminal $t \in \mathrm{First}(\alpha)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \mathrm{First}(\alpha)$, then for each $t \in \mathrm{Follow}(A)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \mathrm{First}(\alpha)$ and $\$ \in \mathrm{Follow}(A)$ do
    - $T[A, \$] = \alpha$

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Notes on LL(1) Parsing Tables

★ If any entry is multiply defined then G is not LL(1)

– If G is ambiguous

– If G is left recursive

– If G is not left-factored

– And in other cases as well

★ Most programming language CFGs are not LL(1)

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Bottom-Up Parsing

Bottom-up parsers don’t need left-factored grammars

more general than top-down parsing

– And just as efficient

– Builds on ideas in top-down parsing

- Bottom-up is the preferred method

- Concepts today, algorithms next time

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

The idea of bottom-up parsing

Revert to the "natural" grammar for our example:

$E \rightarrow$ `T + E | T`

$T \rightarrow$ `int * T | int | (E)`

Consider the string: **`int * int + int`**

reduce a string to the start symbol by inverting productions:

➔	`int * int + int`	$T \rightarrow$ `int`
➔	`int * T + int`	$T \rightarrow$ `int * T`
➔	`T + int`	$T \rightarrow$ `int`
➔	`T + T`	$E \rightarrow$ `T`
➔	`T + E`	$E \rightarrow$ `T + E`
➔	`E`	

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

Revert to the “natural” grammar for our example:

$E \rightarrow T + E \mid T$

$T \rightarrow int * T \mid int \mid (E)$

Consider the string: **int * int + int**

> ★ Read the productions in reverse (from bottom to top)
> ★ This is a reverse rightmost derivation!

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

reduce a string to the start symbol by inverting productions:

➔ int * int + int
- ◆ $T \rightarrow int$

➔ int * T + int
- ◆ $T \rightarrow int * T$

➔ T + int
- ◆ $T \rightarrow int$

➔ T + T
- ◆ $E \rightarrow T$

➔ T + E
- ◆ $E \rightarrow T + E$

➔ E

For derivation

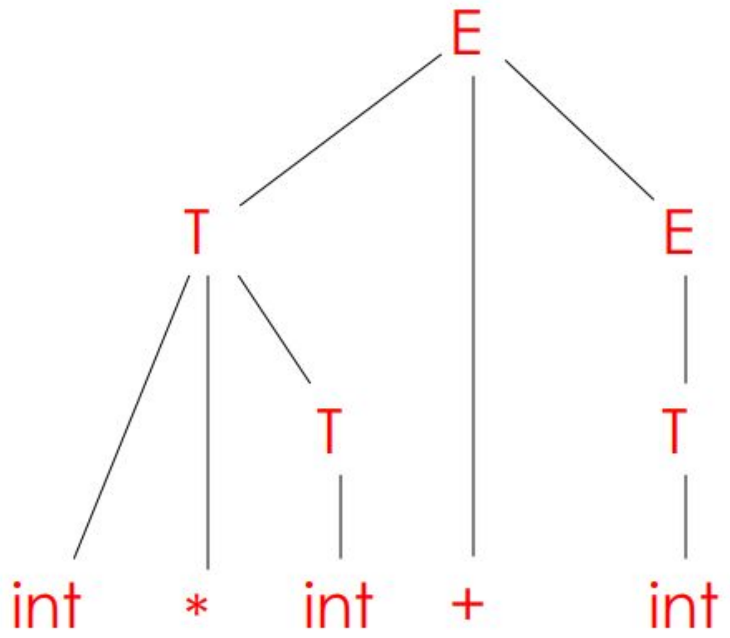
$\textbf{Goal} = \gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_{n-1} \rightarrow \gamma_n = \textbf{sentence}$,

The bottom-up parser discovers $\gamma_i \rightarrow \gamma_{i+1}$ before it discovers $\gamma_{i-1} \rightarrow \gamma_i$

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse

```
➔  int * int + int
     ◆  T → int
➔  int * T + int
     ◆  T → int * T
➔  T + int
     ◆  T → int
➔  T + T
     ◆  E → T
➔  T + E
     ◆  E → T + E
➔  E
```



https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Where Do Reductions Happen?

Important Fact #1 has an interesting consequence:

– Let $\alpha\beta\omega$ be a step of a bottom-up parse

– Assume the next reduction is by $X \to \beta$

That is $\boldsymbol{\alpha X \omega \to \alpha\beta\omega}$

– Then ω is a string of terminals

Why?

★ Because $\alpha X \omega \to \alpha\beta\omega$ is a step in a right-most derivation

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Shift-Reduce Parsing

x1x2 ... xi | xi+1 ... xn

Notation:

Idea: Split string into two substrings

The dividing point is marked by a |

- The | is not part of the string

Left substring

- has terminals and non-terminals

Right substring

- is as yet unexamined by parsing (a string of terminals)

- Initially, all input is unexamined
  - |x1x2 . . . xn

Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Shift and Reduce

Shift:

Move | one place to the right

Shifts a terminal to the left string

$$ABC|xyz \Rightarrow ABCx|yz$$

Reduce:

Apply an inverse production at the right end of the left string

If $A \rightarrow xy$ is a production, then

$$Cbxy|ijk \Rightarrow CbA|ijk$$

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Example with reductions

- ➔ `int * int | + int`
  - ◆ reduce $T \to$ int
- ➔ `int * T | + int`
  - ◆ reduce $T \to$ int * T
- ➔ `T + int |`
  - ◆ reduce $T \to$ int
- ➔ `T + T |`
  - ◆ reduce $E \to T$
- ➔ `T + E |`
  - ◆ reduce $E \to T + E$
- ➔ `E |`

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

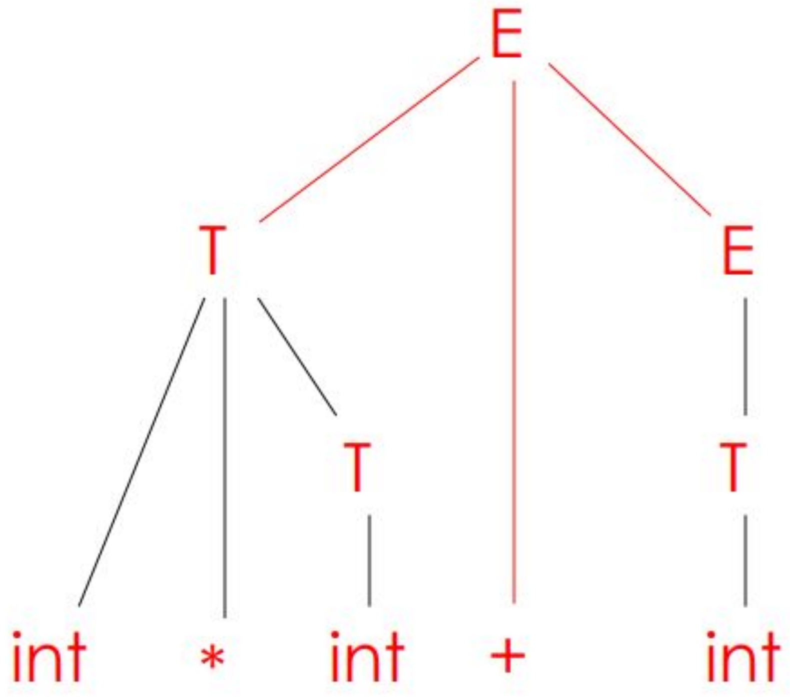
Example with shift-reduce parsing

- ➔ | int * int + int
  - ◆ shift
- ➔ int | * int + int
  - ◆ shift
- ➔ int * | int + int
  - ◆ shift
- ➔ int * int | + int
  - ◆ reduce T → int
- ➔ int * T | + int
  - ◆ reduce T → int * T
- ➔ T | + int
  - ◆ shift
- ➔ T + | int
  - ◆ shift
- ➔ T + int |
  - ◆ reduce T → int
- ➔ T + T |
  - ◆ reduce E → T
- ➔ T + E |
  - ◆ reduce E → T + E
- ➔ E |

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

The generation of parse tree

```
➔  | int * int + int
     ◆ shift
➔  int | * int + int
     ◆ shift
➔  int * | int + int
     ◆ shift
➔  int * int | + int
     ◆ reduce T → int
➔  int * T | + int
     ◆ reduce T → int * T
➔  T | + int
     ◆ shift
➔  T + | int
     ◆ shift
➔  T + int |
     ◆ reduce T → int
➔  T + T |
     ◆ reduce E → T
➔  T + E |
     ◆ reduce E → T + E
➔  E |
```



https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

The stack

Left string can be implemented by a stack

– Top of the stack is the |

- Shift
    - pushes a terminal on the stack

- Reduce
    - pops 0 or more symbols off of the stack (production rhs)
    - and pushes a non-terminal on the stack (production lhs)

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

1. P → E
2. E → E + T
3. E → T
4. T → id (E)
5. T → id

An example Shift-Reduce Parsing **with 1 lookahead**

Stack	**Input**	**Action**
	id (id + id) $	shift
id	(id + id) $	shift
id (	id + id) $	shift
id (id	+ id) $	reduce T → id
id (T	+ id) $	reduce E → T
id (E	+ id) $	shift
id (E +	id) $	shift
id (E + id	) $	reduce T → id
id (E + T	) $	reduce E → E + T
id (E	) $	shift
id (E)	$	reduce T → id(E)
T	$	reduce E → T
E	$	reduce P → E
P	$	accept

https://www3.nd.edu/~dthain/compilerbook/chapter4.pdf

Key issue

Example grammar:

E → T + E | T

T → int * T | int | (E)

Consider step

- ➔ **int | * int + int**
  - ◆ We could reduce by **T → int**
- ➔ **T | * int + int**
- ➔ A fatal mistake!
  - ◆ No way to reduce to the start symbol E
- ★ How do we decide when to shift or reduce?

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Conflicts

- In a given state, more than one action (shift or reduce) may lead to a valid parse
- **A shift-reduce conflict**
  - If it is legal to shift or reduce
- A **reduce-reduce conflict**
  - if it is legal to reduce by two different productions

You will see such conflicts in your project!

- – More next time . .

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Handles

★ Intuition: Want to reduce only if the result can still be reduced to the start symbol

Assume a rightmost derivation

$$\mathbf{S \to^* \alpha X \omega \to \alpha\beta\omega}$$

- Then $X \to \beta$ in the position after α is a handle of $\alpha\beta\omega$
- Can and must reduce at handles

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Handles formalize the intuition

– A handle is a string that can be reduced and also allows further reductions back to the start symbol (using a particular production at a specific spot)

We only want to reduce at handles

★ Note: We have said what a handle is, not how to find handles

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Summary of key ideas

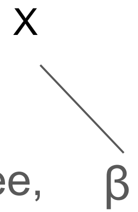
αβω Assume β is at position k, and we have a rule X → β.

- Parser looks the current **frontier**
  - If it finds β in the frontier,
  - it can replace β with X to create a **new frontier**.
- **Handle: <X → β, k>** this pair is a handle
  - if replacing β with X at position k is the next step in a **valid derivation for the input string**
  - then the parser should replace β with X.

- **Reduction:**
  - This replacement is called a reduction because it reduces the number of symbols on the frontier, unless |β| = 1.

In the parse tree,

- build a node for X,
- add that node to the tree,
- and connect the nodes representing β as X's children.



★ **Important Fact #2 about bottom-up parsing:**
  - In shift-reduce parsing, handles appear only at the top of the stack, never inside

Why?

Informal induction on # of reduce moves:

- initially, stack is empty
- Immediately after reducing a handle
  - right-most non-terminal on top of the stack
  - next handle must be to right of right-most non-terminal, because this is a right-most derivation
  - Sequence of shift moves reaches next handle

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Summary of Handles

In shift-reduce parsing, handles always appear at the top of the stack

Handles are never to the left of the rightmost nonterminal

- Therefore, shift-reduce moves are sufficient; the | need never move left

Bottom-up parsing algorithms are based on recognizing handles

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf

Recognizing handles

★ There are no known efficient algorithms to recognize handles

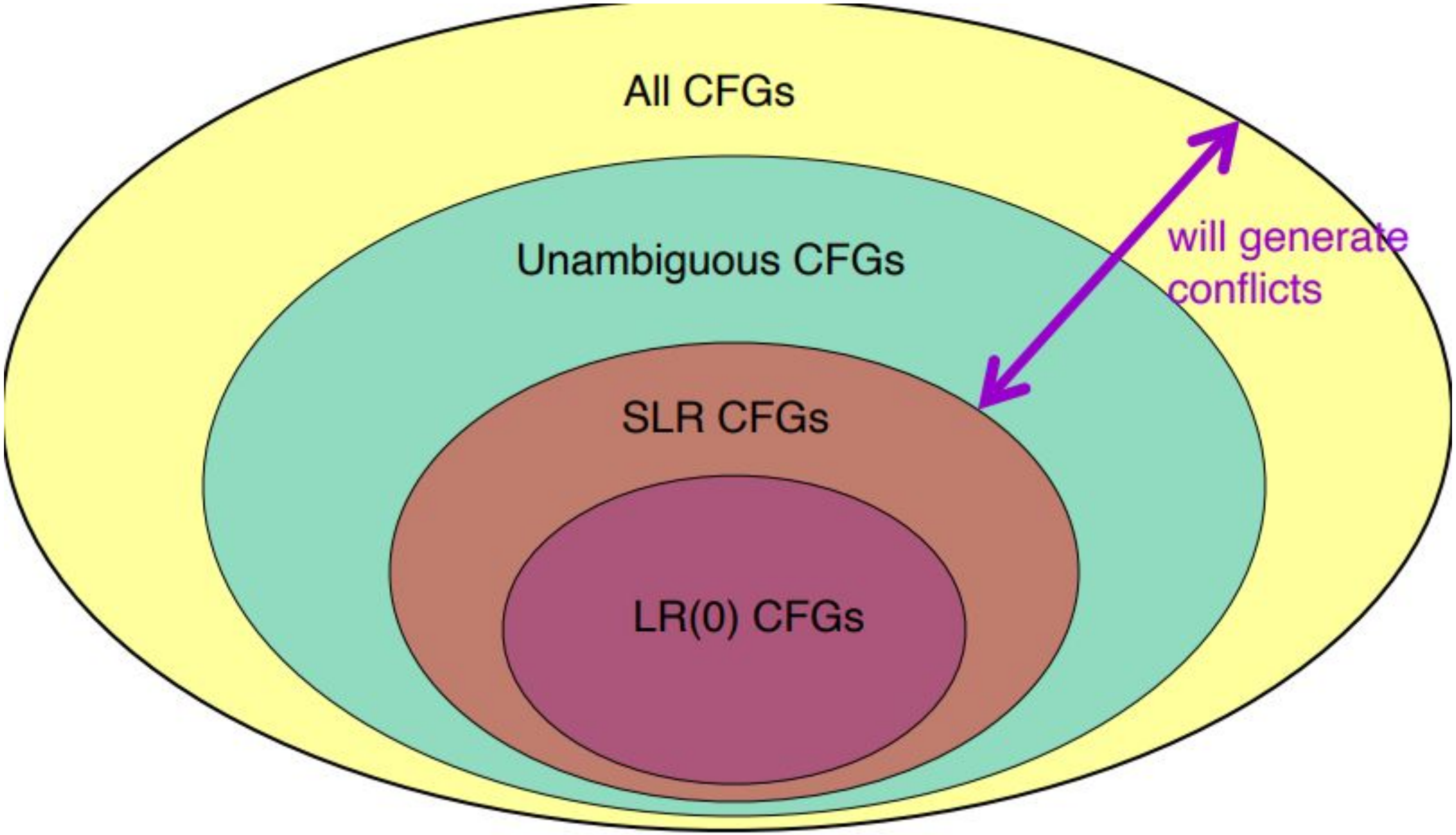
• **Solution:** use heuristics to guess which stacks are handles

On some CFGs, the heuristics always guess correctly

– For the heuristics we use here, these are the SLR grammars

– Other heuristics work for other grammars

https://web.stanford.edu/class/cs143/lectures/lecture07.pdf



https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

Viable Prefixes

It is not obvious how to detect handles

At each step the parser sees only the stack, not the entire input; start with that . . .

α is a viable prefix

- if there is an **ω** such that **α|ω** is a state of a shift-reduce parser

What does this mean?

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

What does this mean?

- the right end of the handle
  - A viable prefix does not extend past this point

- **It's viable** prefix because it is a **prefix of the handle**

- As long as a parser has viable prefixes on the stack no parsing error has been detected

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

before shifting + to the stack
we have reduced (E) to B

we can only have **(, (E, (E)** on stack

- but we cannot have **(E)+** on stack because (E) is a handle and the items in the stack cannot exceed beyond the handle

A⟶B+id⟶(E)+id
Right most derivation

Operation performed	Stack	Comments
(.E)+id	**(**	shift (
(E.)+id	**(E**	shift E
(E).+id	**(E)**	shift)
B.+id	B	reduce (E) to B
B+.id	B +	shift +
B+id.	B + id	shift id
A	A	reduce B + id to A

★ (, (E, (E) are all viable prefixes for the handle (E)
★ and only these prefixes are present in stack of shift reduce parser.

https://www.geeksforgeeks.org/viable-prefix-in-bottom-up-parsing/

we keep on shifting the items until we reach the handle or an error occurs.

Once a handle is reached we reduce it with a non-terminal using the suitable production.

Thus viable prefixes help in taking appropriate shift-reduce decisions.

As long as stack contains these prefixes there cannot be any error.

S → AA
A → bA | a

Input string: **bbbaa**

S.No.	Reverse Rightmost Derivation with Handles	Viable Prefix	Comments
1.	S -> bbb**a**a	b, bb, bbb, bbba	Here, a is the handle so viable prefix cannot exceed beyond a.
2.	S -> bb**bA**a	b, bb, bbb, bbbA	Here, bA is the handle so viable prefix cannot exceed beyond bA.
3.	S -> b**bA**a	b, bb, bbA	Here also, bA is the handle so viable prefix cannot exceed beyond bA.
4.	S -> **bA**a	b, bA	Here also, bA is the handle so viable prefix cannot exceed beyond bA.
5.	S -> A**a**	A, Aa	Here, a is the handle so viable prefix cannot exceed beyond a.
6.	S -> **AA**	A, AA	Here, AA is the handle so viable prefix cannot exceed beyond AA.

https://www.geeksforgeeks.org/viable-prefix-in-bottom-up-parsing/

Important Fact #3 about bottom-up parsing:

For any grammar, the set of viable prefixes is a regular language

-

Important Fact #3 is non-obvious

Next lecture

we will show how to compute automata that accept viable prefixes

And how to build action, goto tables

https://web.stanford.edu/class/cs143/lectures/lecture08.pdf

A simple table driven **LR(1)** parser.

- LR(1): left-to-right scan, reverse rightmost derivation, and 1 symbol of lookahead

We will build these tables!

1	*Goal*	→	*List*
2	*List*	→	*List Pair*
3		|	*Pair*
4	*Pair*	→	(*Pair*)
5		|	( )

	Action Table			Goto Table	
State	eof	(	)	***List***	***Pair***
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

Behaviour for the input string “**( )**”

Iteration	State	word	Stack	Handle	Action
initial	—	(	\$ 0	— *none* —	—
1	0	(	\$ 0	— *none* —	*shift 3*
2	3	)	\$ 0 (3	— *none* —	*shift 7*
3	7	eof	\$ 0 (3) 7	( )	*reduce 5*
4	2	eof	\$ 0 *Pair* 2	*Pair*	*reduce 3*
5	1	eof	\$ 0 *List* 1	*List*	*accept*

When it finds a handle <A →β, k>, it reduces β at k to A

Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 3