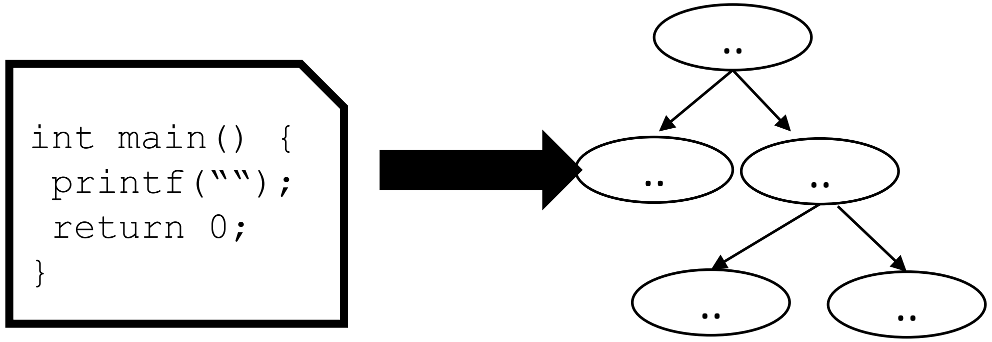
CSE110A: Compilers

Topics: Algorithms for Parsing

- *Syntactic Analysis continued*
  - *Top down parsing*
    - *Oracle parser*
    - *Rewriting to avoid left recursion*



New topic: Algorithms for parsing

One goal:

- Given a string *s* and a CFG *G,* determine if *G* can derive *s*

- We will do that be implicitly attempting to derive a parse tree for *s*

- Two different approaches, each with different trade-offs:
  - Top down
  - Bottom up

Top-down parsing

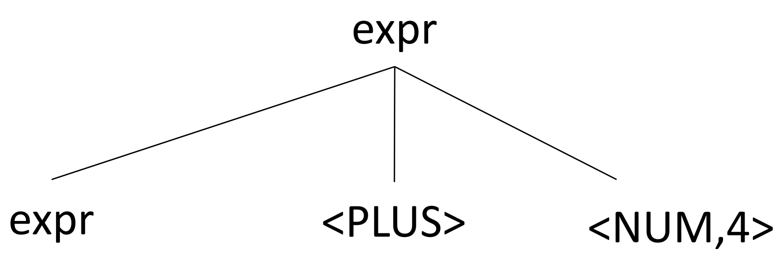
`input: 2+3+4`

expr

Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`

Top-down parsing

`input: 2+3+4`

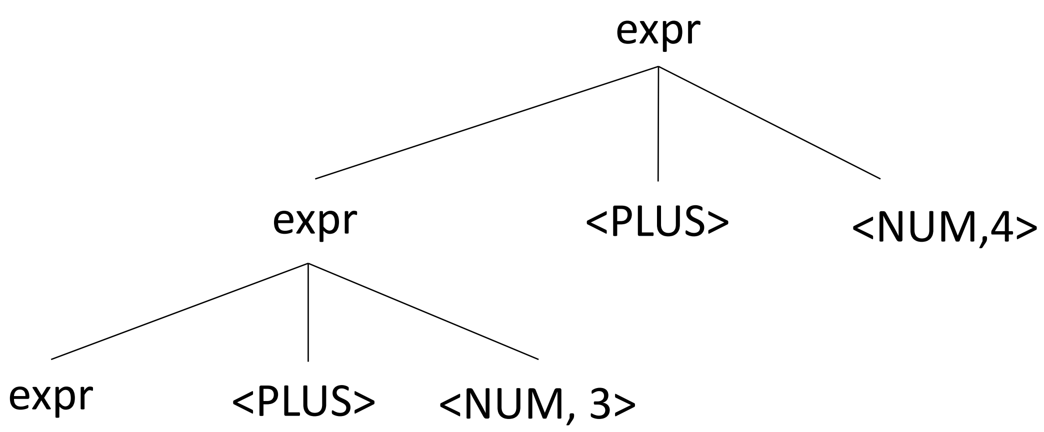


<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>+</td><td><code>expr</code></td><td><code>: expr PLUS NUM</code>
<code>| NUM</code></td></tr>
</table>

Top-down parsing

`input: 2+3+4`

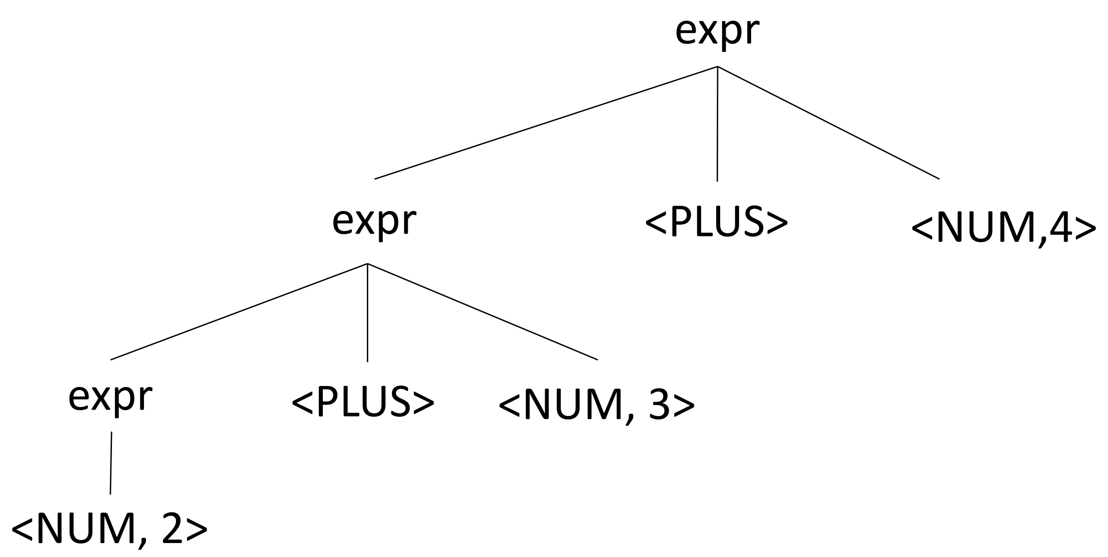
<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>+</td><td><code>expr</code></td><td><code>: expr PLUS NUM</code>
<code>| NUM</code></td></tr>
</table>



Top-down parsing

`input: 2+3+4`

Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`



Top-down parsing

Pros:

- Algorithm is simpler
- Faster than bottom-up
- Easier recovery

Cons:

- Not efficient on arbitrary grammars
- Many grammars need to be re-written

Bottom-up parsing

`input: 2+3+4`

Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`

<NUM, 2> <PLUS> <NUM, 3> <PLUS> <NUM,4>

Bottom-up parsing

`input: 2+3+4`

Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`

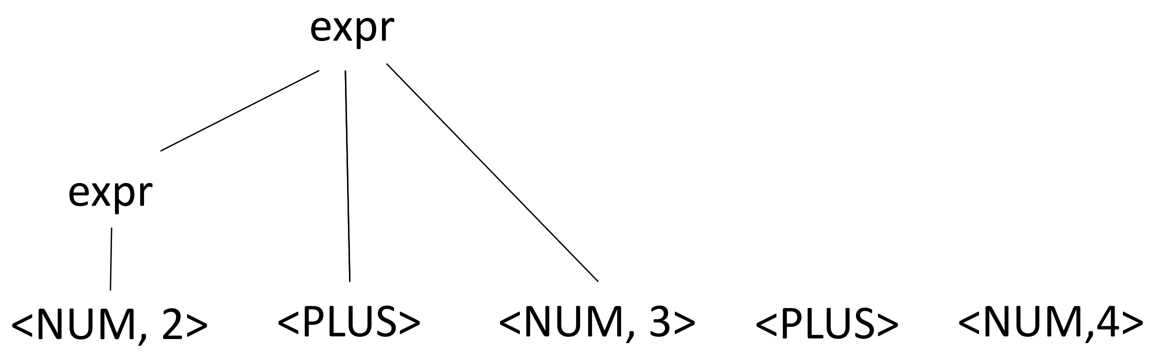
expr

<NUM, 2> <PLUS> <NUM, 3> <PLUS> <NUM,4>

Bottom-up parsing

`input: 2+3+4`

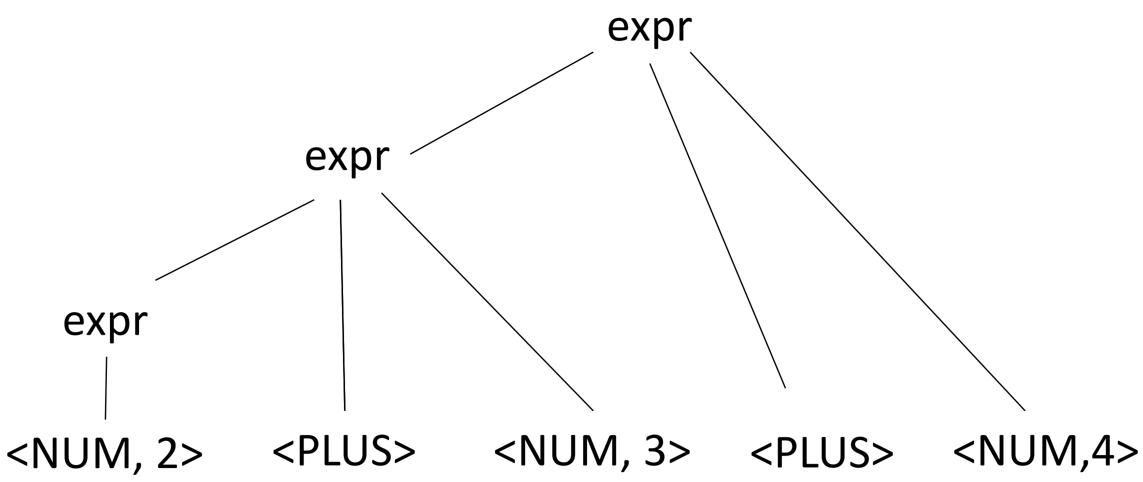
Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`



Bottom-up parsing

`input: 2+3+4`

Operator	Name	Productions
+	`expr`	`: expr PLUS NUM` `| NUM`



Bottom up

Pros:

- can handle grammars expressed more naturally
- can encode precedence and associativity even if grammar is ambiguous

Cons:

- algorithm is complicated
- in many cases slower than top down

Let’s start with top down

```
root = start symbol;
focus = root;
push(None);
to_match = s.token();

while (true):
  if (focus is a nonterminal)
    pick next rule (A ::= B1,B2,B3...BN);
    push(BN... B3, B2);
    focus = B1

  else if (focus == to_match)
    to_match = s.token()
    focus = pop()

  else if (to_match == None and focus == None)
    Accept
```

Variable	Value
focus	
to_match	
s.istring	
stack	

```
1: Expr   ::= Expr Op Unit
2:          |   Unit
3: Unit   ::= '(' Expr ')'
4:          |    ID
5: Op     ::= '+'
6:          |   '*'
```

Can we derive the string `(a+b)*c`

Expanded Rule	Sentential Form
start	Expr

```
root = start symbol;
focus = root;
push(None);
to_match = s.token();

while (true):
  if (focus is a nonterminal)
    pick next rule (A ::= B1,B2,B3...BN);
    push(BN... B3, B2);
    focus = B1

  else if (focus == to_match)
    to_match = s.token()
    focus = pop()

  else if (to_match == None and focus == None)
    Accept
```

Currently we assume this is magic and picks the right rule every time

Variable	Value
focus	
to_match	
s.istring	
stack	

```
1: Expr   ::= Expr Op Unit
2:          |   Unit
3: Unit   ::= '(' Expr ')'
4:          |   ID
5: Op     ::= '+'
6:          |   '*'
```

Can we derive the string `(a+b)*c`

Expanded Rule	Sentential Form
start	Expr

```
root = start symbol;
focus = root;
push(None);
to_match = s.token();

while (true):
  if (focus is a nonterminal)
    pick next rule (A ::= B1,B2,B3...BN);
    push(BN... B3, B2);
    focus = B1

  else if (focus == to_match)
    to_match = s.token()
    focus = pop()

  else if (to_match == None and focus == None)
    Accept
```

Currently we assume this is magic and picks the right rule every time

```
1: Expr   ::= Expr Op Unit
2:         |   Unit
3: Unit   ::= '(' Expr ')'
4:         |    ID
5: Op     ::= '+'
6:         |   '*'
```

Can we derive the string (a+b)*c

Variable	Value
focus	Op
to_match	'+'
s.istring	*b)*c*
stack	Unit ')' Op, Expr, None

Expanded Rule	Sentential Form
start	Expr
1	Expr Op Unit
2	Unit Op Unit
3	'(' Expr ')' Op Unit
1	'(' Expr Op Unit ')' Op Unit
2	'(' Unit Op Unit ')' Op Unit
4	'(' ID Op Unit ')' Op Unit

And so on...

```
root = start symbol;
focus = root;
push(None);
to_match = s.token();

while (true):
  if (focus is a nonterminal)
    pick next rule (A ::= B1,B2,B3...BN);
    push(BN... B3, B2);
    focus = B1

  else if (focus == to_match)
    to_match = s.token()
    focus = pop()

  else if (to_match == None and focus == None)
    Accept
```

What can go wrong if we don't have a magic choice

```
1: Expr   ::= Expr Op Unit
2:          |   Unit
3: Unit   ::= '(' Expr ')'
4:          |    ID
5: Op     ::= '+'
6:          |   '*'
```

Can we derive the string `(a+b)*c`

Variable	Value
focus	
to_match	
s.istring	
stack	

Expanded Rule	Sentential Form
start	Expr

```
root = start symbol;
focus = root;
push(None);
to_match = s.token();

while (true):
  if (focus is a nonterminal)
    pick next rule (A ::= B1,B2,B3...BN);
    push(BN... B3, B2);
    focus = B1

  else if (focus == to_match)
    to_match = s.token()
    focus = pop()

  else if (to_match == None and focus == None)
    Accept
```

What can go wrong

Variable	Value
focus	
to_match	
s.istring	
stack	

```
1: Expr   ::= Expr Op Unit
2:          |   Unit
3: Unit   ::= '(' Expr ')'
4:          |    ID
5: Op     ::= '+'
6:          |   '*'
```

Can we derive the string `(a+b)*c`

Expanded Rule	Sentential Form
start	Expr
2	Expr Op Unit
2	Expr Op Unit Op Unit
2	Expr Op Unit Op Unit Op Unit
2	Expr Op Unit ....

Infinite recursion!

Top down parsing does not handle left recursion

```
1: Expr   ::= Expr Op Unit
2:          |   Unit
3: Unit   ::= '(' Expr ')'
4:          |    ID
5: Op     ::= '+'
6:          |   '*'
```

direct left recursion

```
1: Expr_base ::= Unit
2:             |   Expr_op
3: Expr_op   ::= Expr_base Op Unit
4: Unit      ::= '(' Expr_base ')'
5:             |    ID
6: Op        ::= '+'
7:             |   '*'
```

indirect left recursion

Top down parsing cannot handle either

Top down parsing does not handle left recursion

- In general, any CFG can be re-written without left recursion

Eliminating direct left recursion

```
Fee ::= Fee “a”
    |   “b”
```

What does this grammar describe?

Eliminating direct left recursion

```
Fee ::= Fee "a"
     |   "b"
```

The grammar can be rewritten as

```
Fee  ::= "b" Fee2

Fee2 ::= "a" Fee2
     |     ""
```

Eliminating direct left recursion

In general, A and B can be any sequence of non-terminals and terminals

```
Fee ::= Fee A
     |   B
```

```
Fee  ::= B Fee2

Fee2 ::= A Fee2
      |   ""
```

Eliminating direct left recursion

```
1: Expr  ::= Expr Op Unit
2:       |   Unit
3: Unit  ::= '(' Expr ')'
4:       |    ID
5: Op    ::= '+'
6:       |   '*'
```

Lets do this one as an example:

```
Fee ::= Fee A
    |   B
```

➡

```
Fee   ::= B Fee2

Fee2  ::= A Fee2
      |    ""
```

Eliminating direct left recursion

```
A = ?
B = ?
```

```
1: Expr  ::= Expr Op Unit
2:       |   Unit
3: Unit  ::= '(' Expr ')'
4:       |    ID
5: Op    ::= '+'
6:       |   '*'
```

```
1: Expr  ::= ?
2: Expr2 ::= ?
3:       | ?
4: Unit  ::= '(' Expr ')'
5:       |    ID
6: Op    ::= '+'
7:       |   '*'
```

Lets do this one as an example:

```
Fee ::= Fee A
    |   B
```

```
Fee   ::= B Fee2

Fee2  ::= A Fee2
      |    ""
```

Eliminating direct left recursion

```
A = Op Unit
B = Unit
```

```
1: Expr  ::= Expr Op Unit
2:       |   Unit
3: Unit  ::= '(' Expr ')'
4:       |    ID
5: Op    ::= '+'
6:       |   '*'
```

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:       | ""
4: Unit  ::= '(' Expr ')'
5:       |    ID
6: Op    ::= '+'
7:       |   '*'
```

Lets do this one as an example:

```
Fee ::= Fee A
    |   B
```

```
Fee   ::= B Fee2

Fee2  ::= A Fee2
      |     ""
```

```
root = start symbol;
focus = root;
push(None);
to_match = s.token();

while (true):
  if (focus is a nonterminal)
    pick next rule (A ::= B1,B2,B3...BN);
    push(BN... B3, B2);
    focus = B1

  else if (focus == to_match)
    to_match = s.token()
    focus = pop()

  else if (to_match == None and focus == None)
    Accept
```

Variable	Value
focus	
to_match	
s.istring	
stack	

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | ""
4: Unit   ::= '(' Expr ')'
5:          |   ID
6: Op     ::= '+'
7:          |   '*'
```

Expanded Rule	Sentential Form
start	Expr

```
root = start symbol;
focus = root;
push(None);
to_match = s.token();

while (true):
  if (focus is a nonterminal)
    pick next rule (A ::= B1,B2,B3...BN);
    push(BN... B3, B2);
    focus = B1

  else if (focus == to_match)
    to_match = s.token()
    focus = pop()

  else if (to_match == None and focus == None)
    Accept
```

How to handle this case?

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          |  ""
4: Unit   ::= '(' Expr ')'
5:          |   ID
6: Op     ::= '+'
7:          |  '*'
```

Variable	Value
focus	
to_match	
s.istring	
stack	

Expanded Rule	Sentential Form
start	Expr

```
root = start symbol;
focus = root;
push(None);
to_match = s.token();

while (true):
  if (focus is a nonterminal)
    pick next rule (A ::= B1,B2,B3...BN);
    if A == "": focus=pop(); continue;
    push(BN... B3, B2);
    focus = B1

  else if (focus == to_match)
    to_match = s.token()
    focus = pop()

  else if (to_match == None and focus == None)
    Accept
```

Variable	Value
focus	
to_match	
s.istring	
stack	

How to handle this case?

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | ""
4: Unit   ::= '(' Expr ')'
5:          |   ID
6: Op     ::= '+'
7:          |   '*'
```

Expanded Rule	Sentential Form
start	Expr

How about indirect left recursion?

```
1: Expr   ::= Expr Op Unit
2:          |   Unit
3: Unit   ::= '(' Expr ')'
4:          |    ID
5: Op     ::= '+'
6:          |   '*'
```

direct left recursion

```
1: Expr_base ::= Unit
2:             |   Expr_op
3: Expr_op   ::= Expr_base Op Unit
4: Unit      ::= '(' Expr_base ')'
5:              |    ID
6: Op        ::= '+'
7:             |   '*'
```

indirect left recursion

Top down parsing cannot handle either

How about indirect left recursion?

```
1: Expr_base ::= Unit
2:            |   Expr_op
3: Expr_op    ::= Expr_base Op Unit
4: Unit       ::= '(' Expr_base ')'
5:             |   ID
6: Op         ::= '+'
7:            |   '*'
```

Identify indirect left left recursion

$$Expr\_base \rightarrow_{lhs} Expr\_op \rightarrow_{lhs} Expr\_base$$

How about indirect left recursion?

```
1: Expr_base  ::= Unit
2:             |   Expr_op
3: Expr_op    ::= Expr_base Op Unit
4: Unit       ::= '(' Expr_base ')'
5:             |   ID
6: Op         ::= '+'
7:             |   '*'
```

Identify indirect left left recursion

$$Expr\_base \rightarrow_{lhs} Expr\_op \rightarrow_{lhs} Expr\_base$$

Substitute indirect non-terminal closer to initial non-terminal

How about indirect left recursion?

```
1: Expr_base ::= Unit
2:           |   Expr_op
3: Expr_op   ::= Expr_base Op Unit
4: Unit      ::= '(' Expr_base ')'
5:            |  ID
6: Op        ::= '+'
7:           |   '*'
```

Identify indirect left left recursion

```
1: Expr_base ::= Unit
2:           |   Expr_base Op Unit
3: Expr_op   ::= Expr_base Op Unit
4: Unit      ::= '(' Expr_base ')'
5:            |  ID
6: Op        ::= '+'
7:           |   '*'
```

What to do with production rule 3?

$$Expr\_base \rightarrow_{lhs} Expr\_op \rightarrow_{lhs} Expr\_base$$

Substitute indirect non-terminal closer to initial non-terminal

How about indirect left recursion?

```
1: Expr_base ::= Unit
2:           |   Expr_op
3: Expr_op   ::= Expr_base Op Unit
4: Unit      ::= '(' Expr_base ')'
5:            |  ID
6: Op        ::= '+'
7:           |   '*'
```

Identify indirect left left recursion

```
1: Expr_base ::= Unit
2:           |   Expr_base Op Unit
3: Expr_op   ::= Expr_base Op Unit
4: Unit      ::= '(' Expr_base ')'
5:            |  ID
6: Op        ::= '+'
7:           |   '*'
```

What to do with production rule 3?
It may need to stay if another production rule references it!

$$Expr\_base \rightarrow_{lhs} Expr\_op \rightarrow_{lhs} Expr\_base$$

Substitute indirect non-terminal closer to initial non-terminal

Next time: algorithms for syntactic analysis

- Continue with our top down parser.
  - Backtracking
  - Lookahead sets