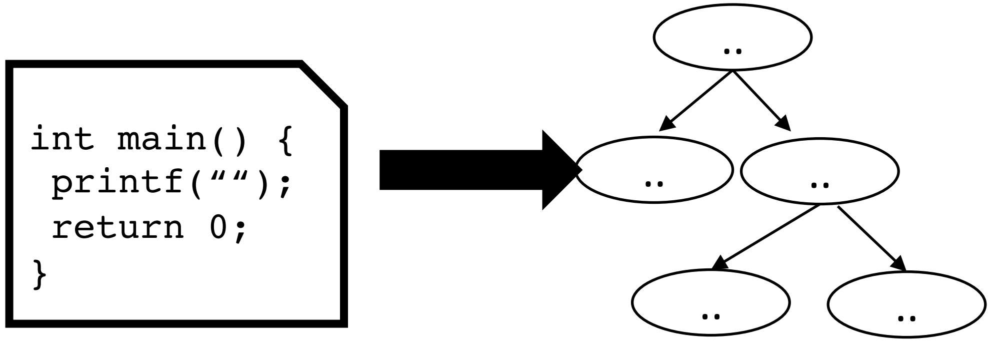
CSE110A: Compilers

April 18, 2022

Topics:

- *Top down parsing*
  - *Dealing with left recursion*
  - *Lookahead sets*



Announcements

- HW 1 is due today
  - No guaranteed help after business hours (e.g. after class at 5 PM)

- HW 2 is scheduled for release today by midnight
  - you have two weeks to do it.
  - due on May 2 at midnight
  - you have what you need for part 1 today
  - you should have what you need for part 2 on Wednesday
  - you should have what you need for part 3 on Friday

- Plenty of time for help for HW 2!

Announcements

- Homework clarification: token actions
  - You can use lists, functions, variables etc in tokens.py *as token actions*
  - These components get bound to the tokens array
  - You should only use the token array in your scanners, and you should be prepared to accept as input any token arrays
  - Your token array should be an array of tuples:

```
(TOKEN_ID     : string,
 TOKEN_REGEX  : string,
 TOKEN_ACTION : lexeme → lexeme)
```

Quiz

Unfortunately Monday's lecture put us behind and we weren't able to get through all the material we needed for the quiz again

To make up for it, I will make Friday's quiz due on Wednesday so that you can answer the extra questions with enough background

Quiz

Which of the following can be sources of ambiguity in grammars?

- [ ] operator associativity not being specified
- [ ] incorrect parenthesis matching
- [ ] operator precedence not being specified
- [ ] operator commutativity not being specified

Quiz

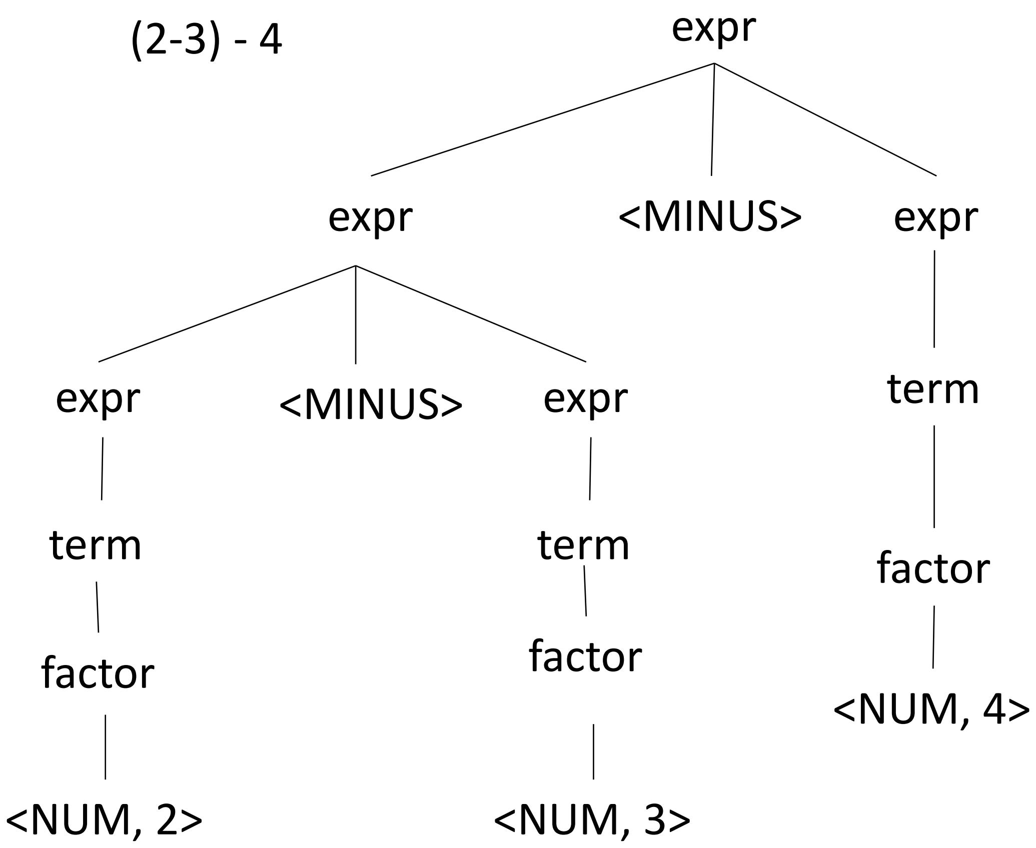
Which of the following can be sources of ambiguity in grammars?

- [ ] operator associativity not being specified
- [ ] incorrect parenthesis matching
- [ ] operator precedence not being specified
- [ ] operator commutativity not being specified

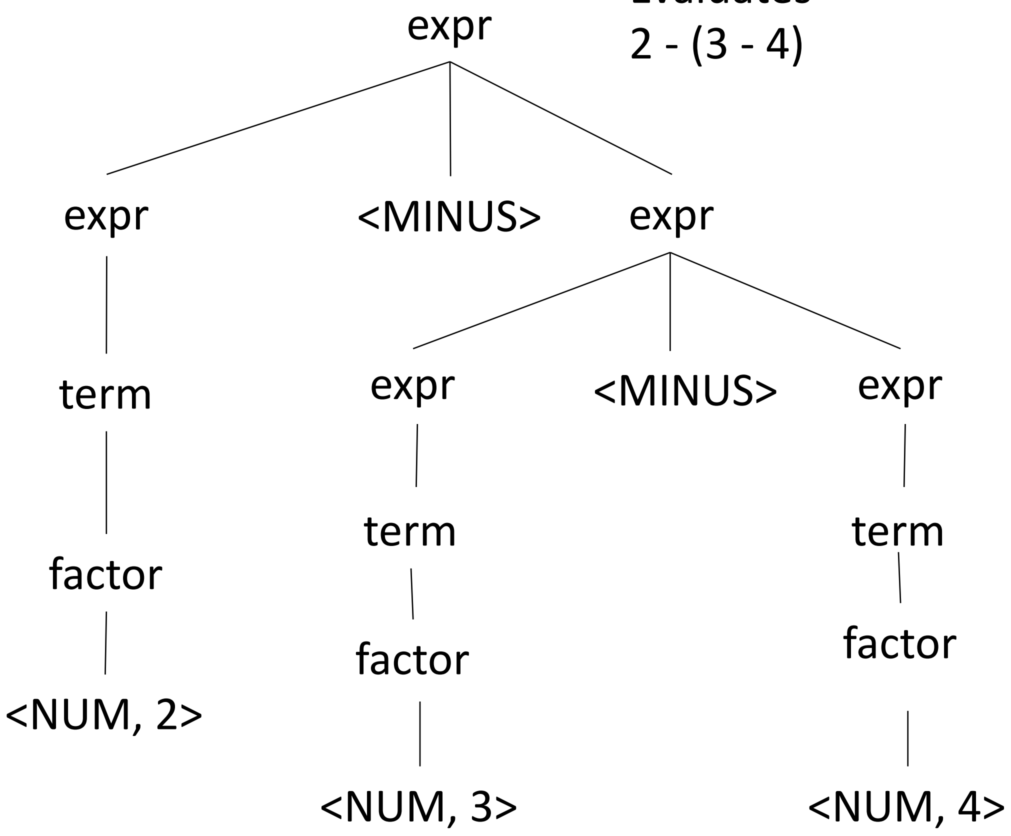
What about for a different operator?

`input: 2-3-4`

Evaluates
(2-3) - 4



Evaluates
2 - (3 - 4)

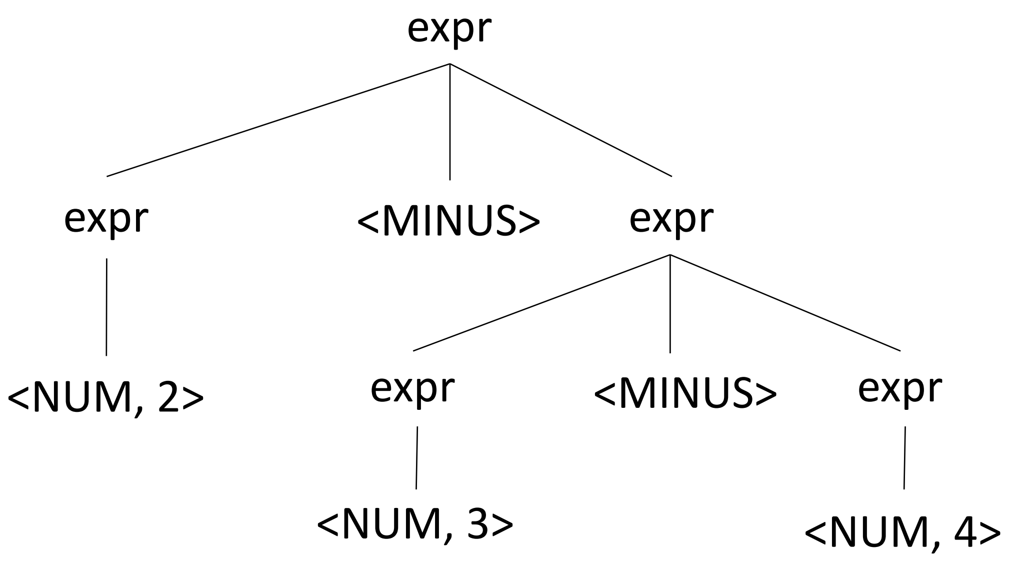


Which one is right?

Associativity for a single operator

Operator	Name	Productions
-	`expr`	`: expr MINUS NUM` `| NUM`

`input: 2-3-4`



No longer allowed

Quiz

Which of the following can be sources of ambiguity in grammars?

- [ ] operator associativity not being specified
- [ ] incorrect parenthesis matching
- [ ] operator precedence not being specified
- [ ] operator commutativity not being specified

Quiz

Which of the following can be sources of ambiguity in grammars?

- [ ] operator associativity not being specified
- [ ] incorrect parenthesis matching
- [ ] operator precedence not being specified
- [ ] operator commutativity not being specified

Not really a cause of ambiguous grammars

Quiz

Which of the following can be sources of ambiguity in grammars?

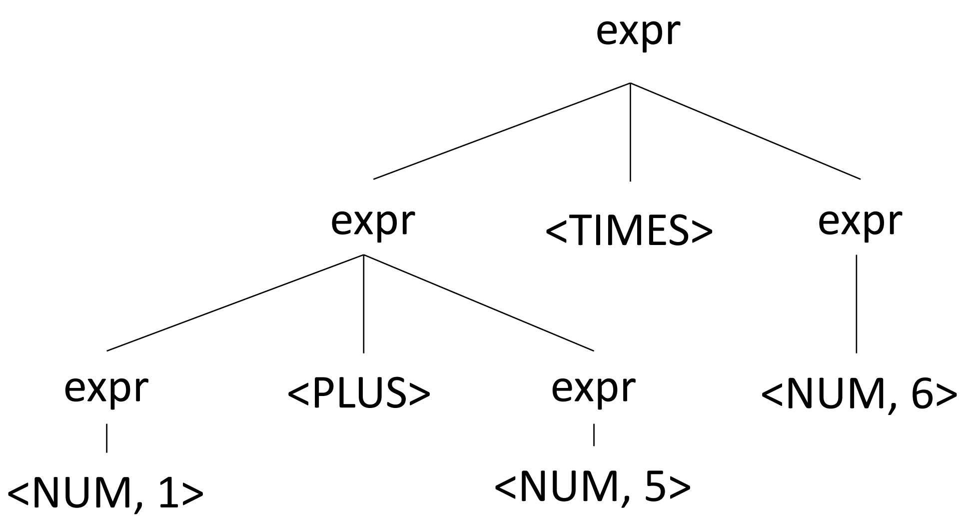
- [ ] operator associativity not being specified
- [ ] incorrect parenthesis matching
- [ ] operator precedence not being specified
- [ ] operator commutativity not being specified

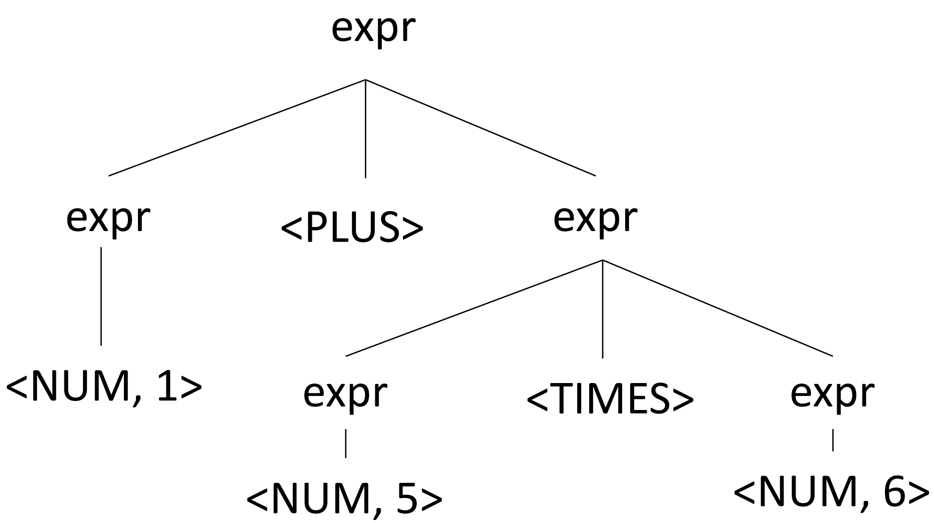
Ambiguous grammars

- `input: 1 + 5 * 6`

```
expr ::= NUM
       | expr PLUS expr
       | expr TIMES expr
       | LPAREN expr RPAREN
```

Evaluations are different!





Avoiding Ambiguity

- new production rules
  - One non-terminal for each level of precedence
  - lowest precedence at the top
  - highest precedence at the bottom

Precedence increases going down

Operator	Name	Productions
+,-	expr	: expr PLUS expr | expr MINUS expr | term
*	term	: term TIMES term | pow
^	pow	: pow ^ pow | factor
()	factor	: LPAREN expr RPAREN | NUM

Quiz

Which of the following can be sources of ambiguity in grammars?

- [ ] operator associativity not being specified
- [ ] incorrect parenthesis matching
- [ ] operator precedence not being specified
- [ ] operator commutativity not being specified

What is commutativity?

Quiz

Which of the following can be sources of ambiguity in grammars?

- [ ] operator associativity not being specified
- [ ] incorrect parenthesis matching
- [ ] operator precedence not being specified
- [ ] operator commutativity not being specified

What is commutativity? `a + b == b + a`

Quiz

Which of the following can be sources of ambiguity in grammars?

- [ ] operator associativity not being specified
- [ ] incorrect parenthesis matching
- [ ] operator precedence not being specified
- [ ] operator commutativity not being specified

What is commutativity? `a + b == b + a`

Parsing doesn't really consider commutativity, but optimizations will

Quiz

We're doing this a little out of order

It is only possible to write a top-down parser if you can determine exactly which production rule to apply at each step.

- ○ True
- ○ False

```
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	Expr
to_match	'a'
s.istring	*"+b)*c"*
stack	Op Unit) Op Unit 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

Quiz

We're doing this a little out of order

It is only possible to write a top-down parser if you can determine exactly which production rule to apply at each step.

- ◯ True
- ◯ False

Answer:

- *true with what we've seen so far*
- *true if you want an efficient parser*
- *false in general*

Quiz

We will answer these ones today in class

To prepare a grammar for a top-down parser, you must ensure that there is no recursion, except in the right-most element of any production rule.

- ○ True
- ○ False

In many cases, a top-down parser requires the grammar to be re-written. Write a few sentences about why this might be an issue when developing a compiler and how the issues might be addressed.

Review

- Let’s do a few more examples of top down parsing

```
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 '+' ID
2:          |   ID
```

Can we derive the string a

Variable	Value
focus	Expr
to_match	
s.istring	
stack	

Expanded Rule	Sentential Form
start	Expr

One more example

```
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 '+' ID
2:          |   ID
```

Can we derive the string a+b

Variable	Value
focus	Expr

Expanded Rule	Sentential Form
start	Expr

New material

- We are going to zoom in on:

```
pick next rule (A ::= B1,B2,B3...BN);
```

So far this rule has been magic. Let's start by turning that magic off

New material

- We are going to zoom in on:

```
pick next rule (A ::= B1,B2,B3...BN);
```

So far this rule has been
magic. Let’s start
turning that magic off

what could the most demonic
choice do...

```
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 could a demonic choice do?

Variable	Value
focus	
to_match	
s.istring	
stack	

```
1: Expr   ::= Expr '+' ID
2:          |   ID
```

Can we derive the string a

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 could a demonic choice do?

```
1: Expr   ::= Expr '+' ID
2:         |    ID
```

Can we derive the string a

Variable	Value
focus	
to_match	
s.istring	
stack	

Expanded Rule	Sentential Form
start	Expr
1	Expr '+' ID
1	Expr '+' ID '+' ID
1	Expr '+' ID '+' ID '+' ID
1	....
1	....
1	....

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

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

Luckily

- 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

The grammar can be rewritten as

```
Fee ::= Fee “a”
    |   “b”
```

```
Fee  ::= “b” Fee2

Fee2 ::= “a” Fee2
     |    “”
```

Eliminating direct left recursion

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 = 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
      |     ""
```

Eliminating direct left recursion

```
1: Expr   ::= Expr '+' ID
2:        |   ID
```

Lets do this one as an example:

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

→

```
Fee   ::= B Fee2

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

Eliminating direct left recursion

```
A = ‘+ ID
B = ID
```

```
1: Expr   ::= Expr ‘+’ ID
2:        |   ID
```

```
1: Expr   ::= ID Expr2
2: Expr2 ::= ‘+’ Expr2
3:           |   ””
```

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  ::= ID Expr2
2: Expr2 ::= '+' Expr2
3:           |   ""
```

Can we match: “a”?

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  ::= ID Expr2
2: Expr2 ::= '+' Expr2
3:          | ""
```

Can we match: "a"?

Variable	Value
focus	Expr2
to_match	None
s.istring	""
stack	None

Expanded Rule	Sentential Form
start	Expr
1	ID Expr2

```
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 B1 == "": 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
```

How to handle this case?

```
1: Expr  ::= ID Expr2
2: Expr2 ::= '+' Expr2
3:        |   ""
```

Can we match: "a"?

Expanded Rule	Sentential Form
start	Expr
1	ID Expr2

Variable	Value
focus	Expr2
to_match	None
s.istring	""
stack	None

```
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 B1 == "": 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
```

How to handle this case?

```
1: Expr  ::= ID Expr2
2: Expr2 ::= '+' Expr2
3:       |    ""
```

Can we match: "a"?

Variable	Value
focus	Expr2
to_match	None
s.istring	""
stack	None

Expanded Rule	Sentential Form
start	Expr
1	ID Expr2
3	ID

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:           |   '*'
```

```
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?

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?
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

What else do we need to do

```
pick next rule (A ::= B1,B2,B3...BN);
```

We cannot have infinite recursion.

What is next?

What else do we need to do

```
pick next rule (A ::= B1,B2,B3...BN);
```

We cannot have infinite recursion.

What is next?

We need to deal with incorrect choices

```
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 B1 == "": 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	Expr2
to_match	None
s.istring	“”
stack	None

```
1: Expr  ::= ID Expr2
2: Expr2 ::= '+' Expr2
           |  ""
```

Can we match: “a”?

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 B1 == “”: 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	None
s.istring	“”
stack	Expr2

```
1: Expr  ::= ID Expr2
2: Expr2 ::= ‘+’ Expr2
           |   ””
```

Can we match: “a”?

Expanded Rule	Sentential Form
start	Expr
1	ID Expr2
2	ID ‘+’ Expr2

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

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

  else if (we have a cached state)
     backtrack();

  else
    parser_error()
```

Keep track of what choices we’ve done

```
1: Expr  ::= ID Expr2
2: Expr2 ::= ‘+’ Expr2
           |   ””
```

Can we match: “a”?

Expanded Rule	Sentential Form
start	Expr
1	ID Expr2
2	ID ’+’ Expr2

Backtracking gets complicated...

- Do we need to backtrack?
  - In the general case, **yes**
  - In many useful cases, **no**

```
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 B1 == "": 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
```

Could we make a smarter choice here?

```
1: Expr  ::= ID Expr2
2: Expr2 ::= '+' Expr2
           |   ""
```

Can we match: "a"?

Variable	Value
focus	Expr2
to_match	None
s.istring	""
stack	None

Expanded Rule	Sentential Form
start	Expr
1	ID Expr2

The First Set

For each production choice, find the set of tokens that each production can start with

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |    ID
6: Op    ::= ‘+’
7:         |   ‘*’
```

The First Set

For each production choice, find the set of tokens that each production can start with

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

```
First sets:
1: {}
2: {}
3: {}
4: {}
5: {}
6: {}
7: {}
```

The First Set

For each production choice, find the set of tokens that each production can start with

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |    ID
6: Op    ::= ‘+’
7:         |   ‘*’
```

```
First sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

We can use first sets to decide which rule to pick!

```
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:          |  '*'
```

```
First sets:
1: {'(', ID}
2: {'+', '*'}
3: {}
4: {'('}
5: {ID}
6: {'+'}
7: {'*'}
```

We simply use to_match and compare it to the first sets for each choice

For example, OP, and UNIT

The First Set

Rules with “” in their First set need special attention

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:       | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:       |    ID
6: Op    ::= ‘+’
7:       |    ‘*’
```

```
First sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {“”}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

```
Follow sets:
1: NA
2: NA
3: {}
4: NA
5: NA
6: NA
7: NA
```

The First Set

Rules with “” in their First set need special attention

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:         | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:         |    ID
6: Op    ::= ‘+’
7:         |    ‘*’
```

```
First sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {“”}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

```
Follow sets:
1: NA
2: NA
3: {}
4: NA
5: NA
6: NA
7: NA
```

We need to find the tokens that any string that follows the production can start with.

The First Set

Rules with “” in their First set need special attention

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:       | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:       |    ID
6: Op    ::= ‘+’
7:       |    ‘*’
```

```
First sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {“”}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

```
Follow sets:
1: NA
2: NA
3: {None, ’)’}
4: NA
5: NA
6: NA
7: NA
```

We need to find the tokens that any string that follows the production can start with.

The First Set

The First+ set is the combination of First and Follow sets

```
                                 First sets:    Follow sets:     First+ sets:
1: Expr  ::= Unit Expr2          1: {'(', ID}   1: NA            1: {'(', ID}
2: Expr2 ::= Op Unit Expr2       2: {'+', '*'}  2: NA            2: {'+', '*'}
3:       | ""                    3: {""}        3: {None, ')'}   3: {None, ')'}
4: Unit  ::= '(' Expr ')'        4: {'('}       4: NA            4: {'('}
5:       |    ID                 5: {ID}        5: NA            5: {ID}
6: Op    ::= '+'                 6: {'+'}       6: NA            6: {'+'}
7:       |   '*'                 7: {'*'}       7: NA            7: {'*'}
```

The First Set

The First+ set is the combination of First and Follow sets

```
1: Expr  ::= Unit Expr2
2: Expr2 ::= Op Unit Expr2
3:       | “”
4: Unit  ::= ‘(‘ Expr ‘)’
5:       |    ID
6: Op    ::= ‘+’
7:       |    ‘*’
```

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

For each non-terminal: if every production has a disjoint First+ set then we do not need any backtracking!

The First Set

The First+ set is the combination of First and Follow sets

```
1: Expr   ::= Unit Expr2
2: Expr2  ::= Op Unit Expr2
3:          | “”
4: Unit   ::= ‘(‘ Expr ‘)’
5:          |    ID
6: Op     ::= ‘+’
7:          |   ‘*’
```

```
First+ sets:
1: {‘(‘, ID}
2: {‘+’, ‘*’}
3: {None, ’)’}
4: {‘(‘}
5: {ID}
6: {‘+’}
7: {‘*’}
```

For each non-terminal: if every production has a disjoint First+ set then we do not need any backtracking!

The First Set

The First+ set is the combination of First and Follow sets

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

```
First+ sets:
1: {'(', ID}
2: {'+', '*'}
3: {None, ')'}
4: {'('}
5: {ID}
6: {'+'}
7: {'*'}
```

These grammars are called LL(1)

- L - scanning the input left to right
- L - left derivation
- 1 - how many look ahead symbols

They are also called predictive grammars

Many programming languages are LL(1)

For each non-terminal: if every production has a disjoint First+ set then we do not need any backtracking!

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:         |   ID ‘[‘ Args ‘]’
3:         |   ID ‘(‘ Args ‘)’
...
```

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:         |   ID ‘[‘ Args ‘]’
3:         |   ID ‘(‘ Args ‘)’
...
```

```
First
1: {}
2: {}
3: {}
...
```

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:         |   ID '[' Args ']'
3:         |   ID '(' Args ')'
...
```

```
First
1: {ID}
2: {ID}
3: {ID}
...
```

We cannot select the next rule based on a single look ahead token!

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:         |   ID ‘[‘ Args ‘]’
3:         |   ID ‘(‘ Args ‘)’
...
```

```
First
1: {ID}
2: {ID}
3: {ID}
...
```

We can refactor

```
1: Factor      ::= ID Option_args
2: Option_args ::= ‘[‘ Args ‘]’
3:              |   ‘(‘ Args ‘)’
4:              |   “”
```

```
First
1: {}
2: {}
3: {}
4: {}
```

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:        |   ID '[' Args ']'
3:        |   ID '(' Args ')'
...
```

```
First
1: {ID}
2: {ID}
3: {ID}
...
```

We can refactor

```
1: Factor      ::= ID Option_args
2: Option_args ::= '[' Args ']'
3:             |   '(' Args ')'
4:             |   ""
```

```
First
1: {ID}
2: {'['}
3: {'('}
4: {""}
```

// We will need to compute the follow set

Sometimes the grammar needs to be refactored

```
1: Factor ::= ID
2:        |   ID '[' Args ']'
3:        |   ID '(' Args ')'
...
```

```
First
1: {ID}
2: {ID}
3: {ID}
...
```

It is not always possible to rewrite grammars into a predictive form, but many programming languages can be.

We can refactor

```
1: Factor      ::= ID Option_args
2: Option_args ::= '[' Args ']'
3:             |   '(' Args ')'
4:             |   ""
```

```
First
1: {ID}
2: {'['}
3: {'('}
4: {""}
```

// We will need to compute the follow set

We now have a full top-down parsing algorithm!

```
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
```

```
First+ sets:
1: {'(', ID}
2: {'+', '*'}
3: {None, ')'}
4: {'('}
5: {ID}
6: {'+'}
7: {'*'}
```

First+ sets for each production rule

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

input grammar, refactored to remove left recursion

To pick the next rule, compare `to_match` with the possible `first+` sets. Pick the rule whose `first+` set contains `to_match`.

If there is no such rule then it is a parsing error.

Quiz

To prepare a grammar for a top-down parser, you must ensure that there is no recursion, except in the right-most element of any production rule.

- ○ True
- ○ False

In many cases, a top-down parser requires the grammar to be re-written. Write a few sentences about why this might be an issue when developing a compiler and how the issues might be addressed.

Next time: recursive descent parser

- *Simpler implementation of a top down parser*