Quiz-07-AMB-TOP-DOWN-REC-DESC

Topics:

- Ambiguity
- HW2
- Top-Down Parsing
- Recursive Descent Parsing

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

Homework 2

Quiz

Please make sure to download HW 2 and try writing simple grammars in PLY. Please mark true when you've tried executing some simply programs and have experimented with specifying some tokens and production rules.

Quiz

Has everyone seen the new program on FIRST, FOLLOW, FIRST+?

A shout out to our student Laurel Willey for the great work on helping
Make this happen.

Top-Down Parsing

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

```
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!

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 = ??
```

```
Expr  ::= Expr Op Unit
      |   Unit
Unit  ::= '(' Expr ')'
      |    ID
Op    ::= '+'
      |   '*'
```

```
Expr  ::= ??
Expr2 ::= ??
      | ??
Unit  ::= '(' Expr ')'
      |    ID
Op    ::= '+'
      |   '*'
```

Lets do this one as an example:

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

```
Fee   ::= B Fee2

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

Quiz

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)
    cache_state();
    pick next rule (A ::= B1,B2,B3...BN);
    if B1 == "": focus=pop(); continue;
    push(BN... B3, B2);
    focus = B1

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

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

  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

Quiz

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.

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
      |      ""
```

Recursive Descent and LL Parsing

Quiz

Is the following grammar backtrack free (as written)?

a → b A

b → D A B

| c B

c → C b

| a C

Quiz

First sets

$a \rightarrow b\ A$	{D,C}
$b \rightarrow D\ A\ B$	{D}
$\mid c\ B$	{C,D}
$c \rightarrow C\ b$	{C}
$\mid a\ C$	{D,C}

Quiz

	First sets
a → b A	{D,C}
b → D A B	{D}
| c B	{C,D}
c → C b	{C}
| a C	{D,C}

Quiz

Is the following grammar backtrack free?

a → b A

b → D A B

| c B

c → C b

| d

d → D b

Quiz

First sets

$a \rightarrow b\ A$	{}
$b \rightarrow D\ A\ B$	{}
$\quad \mid c\ B$	{}
$c \rightarrow C\ b$	{}
$\quad \mid d$	{}
$d \rightarrow D\ b$	{}

Quiz

First sets

a → b A	{C,D}
b → D A B	{D}
| c B	{C,D}
c → C b	{C}
| d	{D}
d → D b	{D}

Quiz

in a recursive descent parser, you make a function for each or what?

- ◯ production option
- ◯ CFG
- ◯ non-terminal
- ◯ terminal

Let’s look at the grammar

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

Let’s look at the grammar

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

How do we parse an Expr?

Let’s look at the grammar

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

How do we parse an Expr?
We parse a Unit followed by an Expr2

Let’s look at the grammar

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

How do we parse an Expr?
We parse a Unit followed by an Expr2

We can just write exactly that!

```
def parse_Expr(self):
    self.parse_Unit();
    self.parse_Expr2();
    return
```

Let's look at the grammar

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

How do we parse an Expr2?

Let's look at the grammar

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

How do we parse an Expr2?

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

Let’s look at the grammar

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

How do we parse an Expr2?

```
def parse_Expr2(self):

    token_id = get_token_id(self.to_match)

    # Expr2 ::= Op Unit Expr2
    if token_id in ["PLUS", "MULT"]:
        self.parse_Op()
        self.parse_Unit()
        self.parse_Expr2()
        return

        # Expr2 ::= ""
    if token_id in [None, ”RPAR"]:
        return

    raise ParserException(-1,                    # line number (for you to do)
                    self.to_match,           # observed token
                    ["PLUS", "MULT", ”RPAR"])   # expected token
```

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

Let’s look at the grammar

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

How do we parse a Unit?

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

Let’s look at the grammar

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

How do we parse a Unit?

```
def parse_Unit(self):

    token_id = get_token_id(self.to_match)

    # Unit  ::= '(' Expr ')'
    if token_id == "LPAR":
        self.eat("LPAR")
        self.parse_Expr()
        self.eat("RPAR")
        return

    # Unit :: = ID
    if token_id == "ID":
        self.eat("ID")
        return

    raise ParserException(-1,             # line number (for you to do)
                          self.to_match,  # observed token
                          ["LPAR", "ID"]) # expected token
```

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

Let’s look at the grammar

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

How do we parse a Unit?

```
def parse_Unit(self):

    token_id = get_token_id(self.to_match)

    # Unit  ::= ‘(‘ Expr ‘)’
    if token_id == "LPAR":
        self.eat("LPAR")
        self.parse_Expr()
        self.eat("RPAR")
        return

    # Unit :: = ID
    if token_id == "ID":
        self.eat("ID")
        return

    raise ParserException(-1,           # line number (for you to do)
                    self.to_match,    # observed token
                    ["LPAR", "ID"])   # expected token
```

ensure that to_match has token ID of “LPAREN” and get the next token

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

Quiz

An LL(1) grammar has a runtime proportional to:

- ○ The number of non-terminals
- ○ The length of the input string
- ○ The number of tokens in the input string
- ○ How many times a backtrack might occur