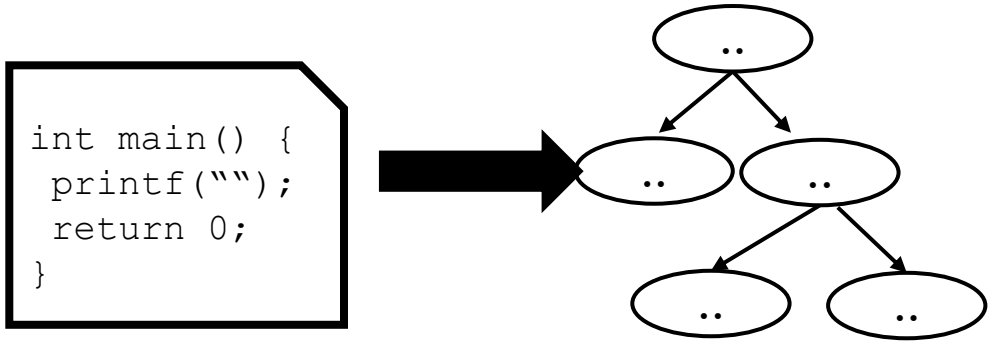
CSE110A: Compilers

Topics:

- *Making a backtrack-free parser*
- *A simple recursive descent parser*



Making a backtrack free parser

```
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
3:       |   ""
```

Could we make a smarter choice here?

Can we match: "a"?

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:       |   ‘*’
```

```
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 Follow 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 Follow 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: {'*'}
```

Do we need backtracking?

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!

Do we need backtracking?

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!

Do we need backtracking?

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!

Recursive Descent allows for easy predictive lookahead of two or more.

Sometimes the grammar needs to be refactored

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

Sometimes the grammar needs to be refactored

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

Left Factoring

Left Factoring: the process of extracting and isolating common prefixes in a set of productions

$$A ::= a\ B_1 \mid a\ B_2 \mid a\ B_3 \mid c_1 \mid c_2 \mid c_3$$

So, define new Non-Terminal for Common Suffixes

Left Factor this way:

$$A ::= a\ B$$
$$B ::= B_1 \mid B_2 \mid B_3$$
$$C ::= c_1 \mid c_2 \mid c_3$$

Often permits a disjoint First Set for every production (RHS) avoiding backtracking.

11	*Factor*	→	name
12		|	name [*ArgList*]
13		|	name (*ArgList*)

11	*Factor*	→	name *Arguments*
12	*Arguments*	→	[*ArgList*]
13		|	(*ArgList*)
14		|	ϵ

Sometimes the grammar needs to be refactored

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

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

Sometimes the grammar needs to be refactored

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

We can refactor

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

Sometimes the grammar needs to be refactored

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

We can refactor

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

A Pattern for Left Factoring

A ::= $a\ B_1$ | $a\ B_2$ | $a\ B_3$ | c_1 | c_2 | c_3

Left Factor this way:

A ::= $a\ B$

B ::= B_1 | B_2 | B_3

C ::= c_1 | c_2 | c_3

S : Expr

1. Expr : Expr + Term
2. | Expr - Term
3. | Term
4. Term : Term * Factor
5. | term / Factor
6. | Factor
7. Factor : (Expr)
8. | num
9. | name

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.

Moving on to a simpler implementation:

Recursive Descent Parser

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
    else:
        raise ParserException(self.linennumber    # 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:          |   '*'
```

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

```
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
```

How do we parse a Unit?

```
def eat(self, expected_token):
    if self.to_match == expected_token:
        self.advance()  # move to the next token
    else:
        raise ParserException(
                self.current_line,    # line# (for you to do)
                self.to_match,        # observed token
                [expected_token])  #   i.e. PAR or RPAR
```

Let's look at the grammar

How do we parse a Unit?

```
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: {‘*’}
```

```
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
```

Note: function eat must ensure that to_match has the expected token ID and advances to the next token, i.e. something like this:

```
def eat(self, expected_token):
    if self.to_match == expected_token:
        self.advance()  # move to the next token
    else:
        raise ParserException(
                    self.current_line,      # line# (for you to do)
                    self.to_match,          # observed token
                    [expected_token])       # LPAR or RPAR or ID)
```

```
class ParserException(Exception):
    def __init__(self, line_number, found_token, expected_tokens):
        self.line_number = line_number
        self.found_token = found_token
        self.expected_tokens = expected_tokens
        # Create a readable error message
        message = (f"Parse error on line {line_number}: found
'{found_token}', "
                   f"expected one of {expected_tokens}")
        super().__init__(message)
```

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 Op?

```
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:       |   '*'
```

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

```
def parse_Op(self):

    token_id = get_token_id(self.to_match)

    # Op  ::= '+'
    if token_id == "PLUS":
        self.eat("PLUS")
        return

    # Op  ::= '*'
    if token_id == "MULT":
        self.eat("MULT")
        return
```

How do we parse an Op?

```
def eat(self, expected_token):
    ...
    raise ParserException(
            self.current_line,
            self.to_match,      # observed token
            [expected_token]) # expected token
    ...
```

Recursive Descent

IS THAT SIMPLE

and allows lookahead > 1