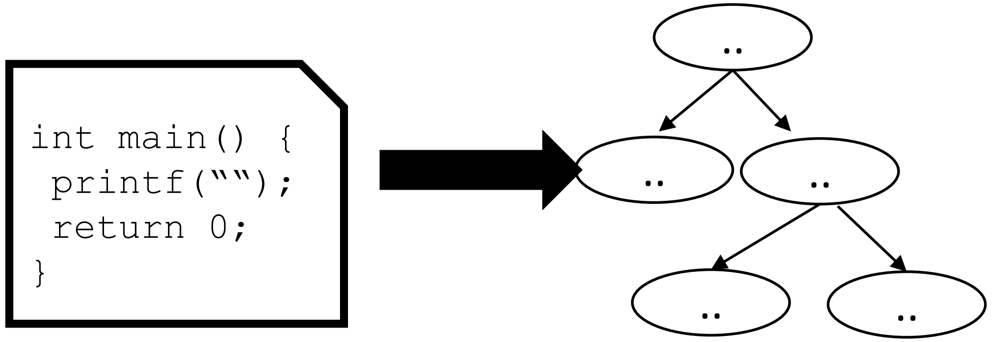
CSE110A: Compilers

Topics:

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



```
int main() {
 printf("");
 return 0;
}
```

It is always possible to eliminate left recursion

```
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
3	ID

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
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

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: {‘*’}
```

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:	Follow sets:
1:	{‘(‘, ID}	NA
2:	{‘+’, ‘*’}	NA
3:	{“”}	{}
4:	{‘(‘}	NA
5:	{ID}	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!*

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: {}
...                                         ...
```

Sometimes the grammar needs to be refactored

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

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

// We will need to compute the follow set

Sometimes the grammar needs to be refactored

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

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

We can refactor

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