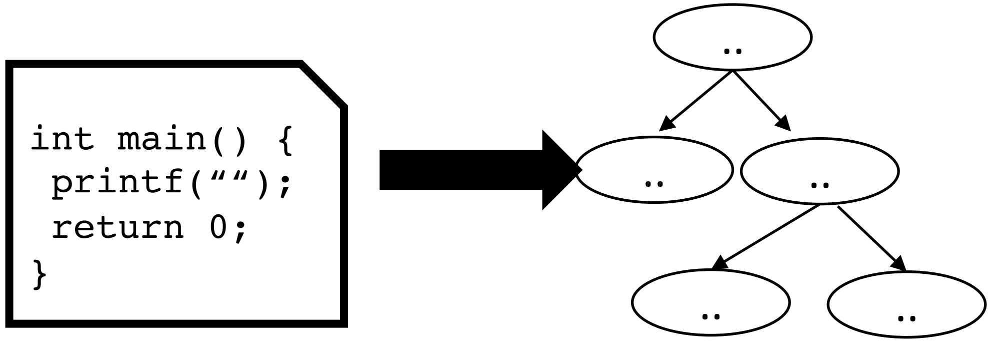
CSE110A: Compilers

April 20, 2022

Topics:

- *Top down parsing*
  - *Lookahead sets*
  - *Recursive descent parsers*

- *Symbol Tables*



Announcements

- HW 2 is out!
  - due on May 2 at midnight
  - you will have what you need for all of part 1 after today
  - you should have what you need for part 2 after today
  - you should have what you need for part 3 on Friday

- Plenty of time for help for HW 2
  - Conceptually and implementation-wise it is bigger than HW 1. I suggest you get started on it early

- Midterm will be given on May 2
  - Take home midterm.
  - Assigned on Monday and due on Friday
  - No late midterms are accepted

Quiz

We'll revisit a few of the questions from the last quiz

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

What is the issue with left recursion?

```
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

```
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
…	
…	
…	

infinite recursion

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:        |   ""
```

Lets do this one as an example:

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

→

```
Fee   ::= B Fee2

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

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_base Op Unit
3: Expr_op   ::= Expr_base Op Unit
4: Unit      ::= ‘(‘ Expr_base ‘)’
5:            |    ID
6: Op        ::= ‘+’
7:           |   ‘*’
```

Identify indirect left left recursion

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

inline indirect non-terminal

It is always possible to eliminate left recursion

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

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: {}
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

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.

Class discussion

New material

- The Follow set
- The First+ set
- Recursive descent parser

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!

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.

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?

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

```
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
```

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:          |   ‘*’
```

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

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: {'*'}
```

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

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:          |    ‘*’
```

How do we parse an Op?

```
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


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

Moving on: Scope

Scope

- What is scope?

- Can it be determined at compile time? Can it be determined at runtime?

- C vs. Python

- Anyone have any interesting scoping rules they know of?

One consideration: Scope

- Lexical scope example

```
int x = 0;
int y = 0;
{
  int y = 0;
  x+=1;
  y+=1;
}
x+=1;
y+=1;
```

What are the final values in x and y?

How to track scope?

- Symbol table object

- two methods:
  - **`lookup(id)` :** `lookup an id in the symbol table. Returns None if the id is not in the symbol table.`

  - **`insert(id,info)` :** `insert a new id (or overwrite an existing id) into the symbol table along with a set of information about the id.`

What information might we store about an id?

a very simple programming language

ID = [a-z]+

INCREMENT = “\+\+”

TYPE = “int”

LBRAC = “{“

RBRAC = “}”

SEMI = “;”

```
int x;
x++;
int y;
y++;
```

statements are either a declaration or an increment

a very simple programming language

ID = [a-z]+

INCREMENT = “\+\+”

TYPE = “int”

LBRAC = “{“

RBRAC = “}”

SEMI = “;”

```
int x;
{
  int y;
  x++;
  y++;
}
y++;
```

statements are either a declaration or an increment

a very simple programming language

ID = [a-z]+

INCREMENT = “\+\+”

TYPE = “int”

LBRAC = “{“

RBRAC = “}”

SEMI = “;”

```
int x;
{
  int y;
  x++;
  y++;
}
y++;
```

error!

statements are either a declaration or an increment

How to track scope?

- `SymbolTable ST;`

Say we are matched the statement:
`int x;`

declare_statement ::= TYPE ID SEMI

```
{}
```

`lookup(id)` `: lookup an id in the symbol table. Returns None if the id is not in the symbol table.`

`insert(id,info)` `: insert a new id (or overwrite an existing id) into the symbol table along with a set of information about the id.`

How to track scope?

- `SymbolTable ST;`

Say we are matched the statement:
`int x;`

declare_statement ::= TYPE ID SEMI

```
{
  self.eat(TYPE)
  variable_name = self.to_match[1] # lexeme value
  self.eat(ID)
  ST.insert(variable_name,None)
  self.eat(SEMI)
}
```

How to track scope?

- `SymbolTable ST;`

Say we are matched string:
`x++;`

inc_statement ::= ID INCREMENT SEMI

`{}`

`lookup(id)` `: lookup an id in the symbol table. Returns None if the id is not in the symbol table.`

`insert(id,info)` `: insert a new id (or overwrite an existing id) into the symbol table along with a set of information about the id.`

How to track scope?

- `SymbolTable ST;`

Say we are matched string:
`x++;`

inc_statement ::= ID INCREMENT SEMI

```
{
  variable_name = self.to_match[1] # lexeme value
  if ST.lookup(variable_name) is None:
      raise SymbolTableException(variable_name)
  self.eat(ID)
  self.eat(INCREMENT)
  self.eat(SEMI)
}
```

How to track scope?

- `SymbolTable ST;`

statement : LBRAC statement_list RBRAC

```
int x;
{
   int y;
   x++;
   y++;
}
y++;
```

How to track scope?

- `SymbolTable ST;`

statement : LBRAC statement_list RBRAC

```
int x;
{
   int y;
   x++;
   y++;
}
y++;
```

start a new scope S

remove the scope S

How to track scope?

- Symbol table
- four methods:
  - **lookup(id)** : lookup an id in the symbol table. Returns None if the id is not in the symbol table.
  - **insert(id,info)** : insert a new id into the symbol table along with a set of information about the id.
  - **push_scope()** : push a new scope to the symbol table
  - **pop_scope()** : pop a scope from the symbol table

How to track scope?

- `SymbolTable ST;`

statement : LBRAC statement_list RBRAC

You will be adding the functions to push and pop scopes in your homework

How to implement a symbol table?

- Thoughts? What data structures are good at mapping strings?

- Symbol table
- four methods:
  - **`lookup(id)`** `: lookup an id in the symbol table. Returns None if the id is not in the symbol table.`

  - **`insert(id,info)`** `: insert a new id into the symbol table along with a set of information about the id.`

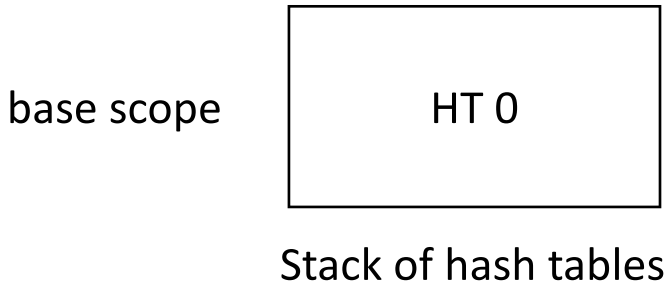
  - **`push_scope()`** `: push a new scope to the symbol table`

  - **`pop_scope()`** `: pop a scope from the symbol table`

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

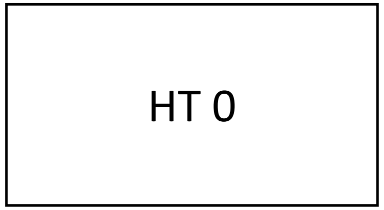


How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`push_scope()`



Stack of hash tables

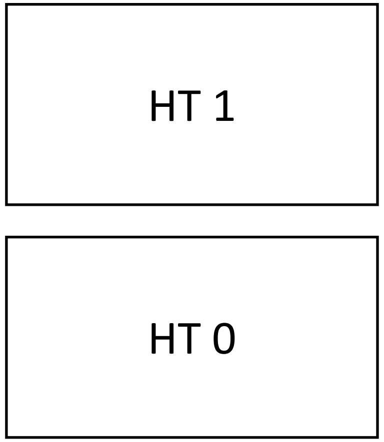
How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

adds a new
Hash Table
to the top of the stack

`push_scope()`



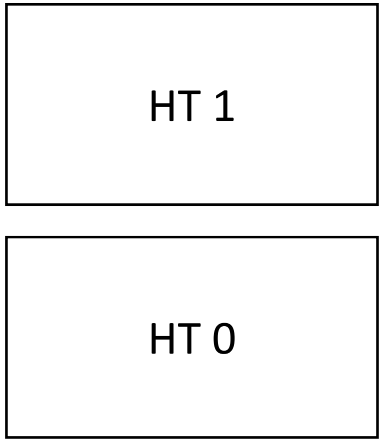
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`insert(id,data)`



Stack of hash tables

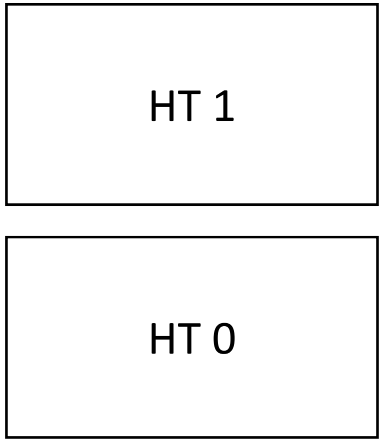
How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

insert `(id -> data)` at top hash table

`insert(id,data)`



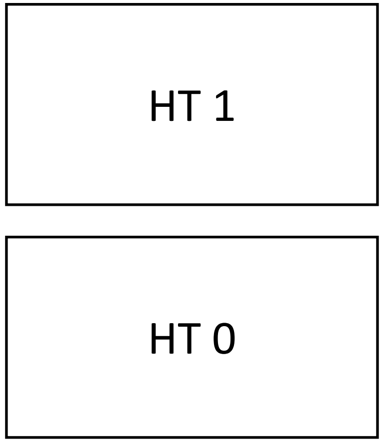
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`lookup(id)`



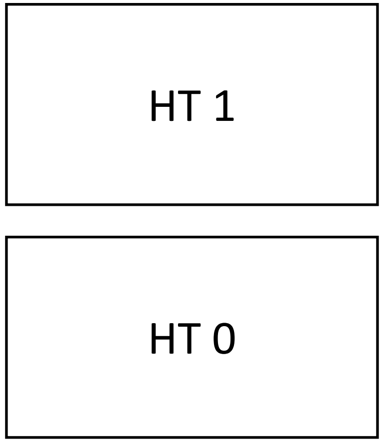
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:
- A good way is a stack of hash tables:

check here
first

`lookup(id)`



Stack of hash tables

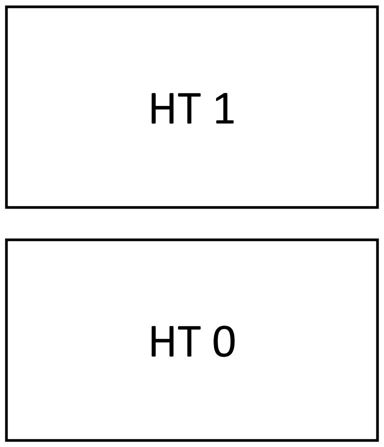
How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`lookup(id)`

then check here



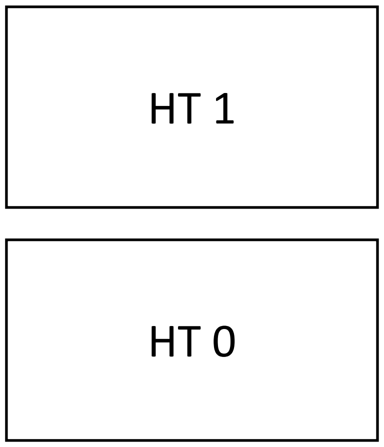
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`pop_scope()`

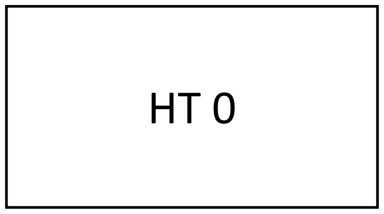


Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

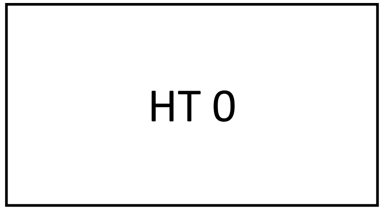


Stack of hash tables

How to implement a symbol table?

- Example

```
int x = 0;
int y = 0;
{
  int y = 0;
  x++;
  y++;
}
x++;
y++;
```



Stack of hash tables

See you on Friday!

- We will discuss parser generators