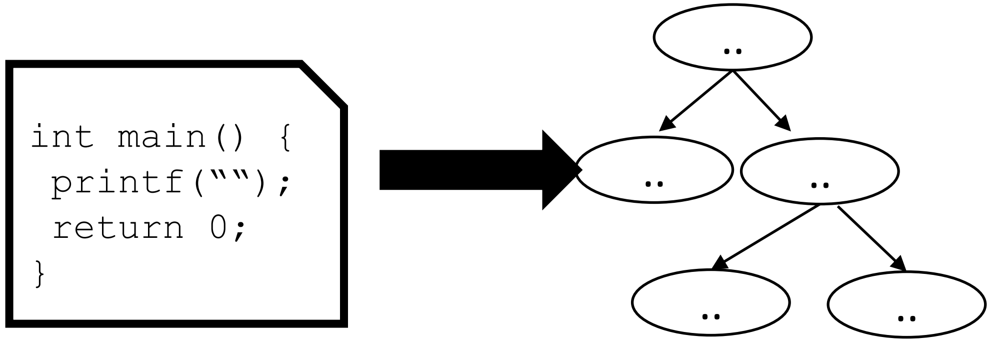
CSE110A: Compilers

April 26, 2024

Topics:

- *Syntactic Analysis continued*
  - *First and follow sets*
  - *Recursive decent parsing*



Announcements

- HW 2 is due on **Monday** by midnight
  - Please get started earlier!
  - You have everything you need

- For help
  - Several office hours before due date
  - Ask on Piazza
    - ***No guaranteed help over the weekend or off business hours***

- ***You do not have to return anything from your parser. Right now it is all about specification. You either match the string or you don’t***

Announcements

- Autograder should be working well now
  - Thanks to those who have worked with us

- We’ve added a few test cases to grade scope
  - However designing test cases, asking questions about the grammar is part of compilers
  - Design your own test cases and try them out
  - Fine to discuss individual test cases on piazza or with classmates

Midterm study guide (so far)

Any of the following are fair game. Anything not listed below but in the lectures are fair game. Any combination of topics is fair game. This is only meant to be an overview of what we have discussed so far.

Midterm study guide (so far)

- Regular expressions
  - Operators, how to specify, how match vs full match works
- Scanners
  - What the API is, how strings are tokenized, how to specify tokens, token actions
- Grammars
  - How to specify a grammar, how to identify/avoid ambiguous grammars, how to show a derivation for match, parse trees
  - How to re-write grammars not to be left recursive, how to identify first+ sets
  - How the top down parsing algorithm works, how a recursive decent parser works
- Symbol tables
  - How scope can be tracked and manage during parse time, symbol table specification and implementation

- First 2 classes of module 3

Homework questions?

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

```
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	
to_match	
s.istring	
stack	

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

The grammar can be rewritten as

```
Fee ::= Fee "a"
     |  "b"
```

```
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 = ??
```

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

```
1: Expr   ::= ??
2: 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
       |    ""
```

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()
```

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

Keep track of what choices we've done

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
       |    ""
```

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
```

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

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

```
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
```

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

```
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?

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

```python
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
```

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: {‘*’}
```

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?

```
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
```

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

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`

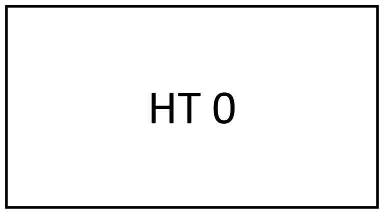
  - **`pop_scope()`** `: pop a scope from the symbol table`

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

base scope



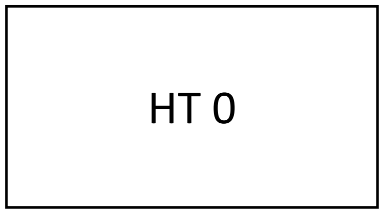
Stack of hash tables

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

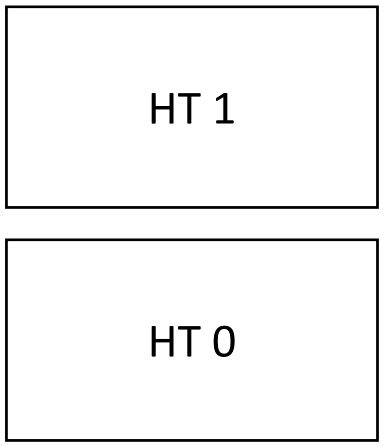
How to implement a symbol table?

- 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()`



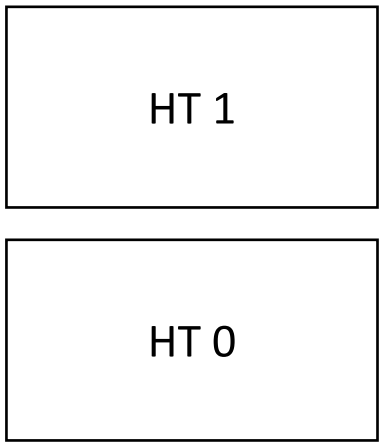
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`insert(id,data)`



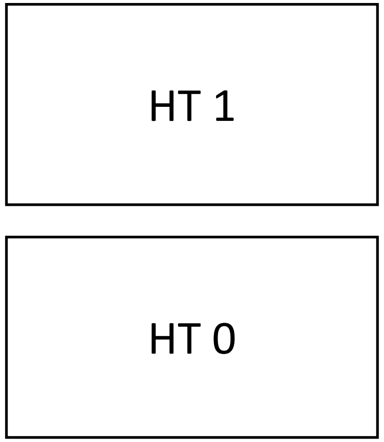
Stack of hash tables

How to implement a symbol table?

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

`insert(id,data)`

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



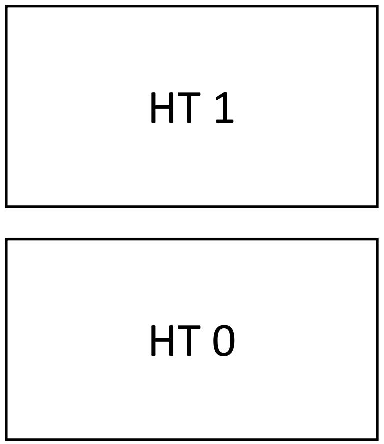
Stack of hash tables

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

HT 1

`lookup(id)`

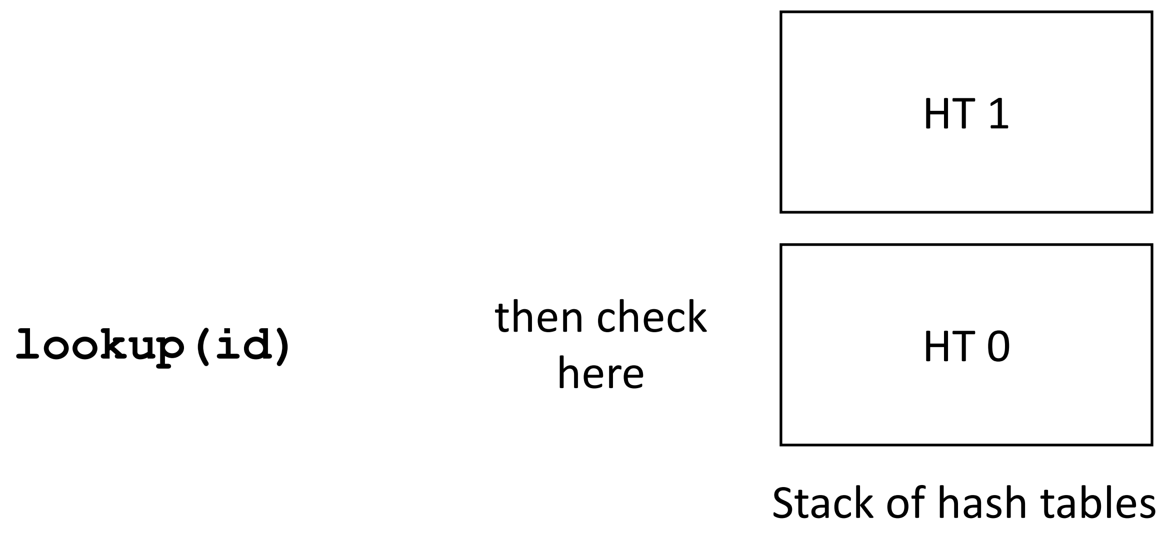
HT 0

Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

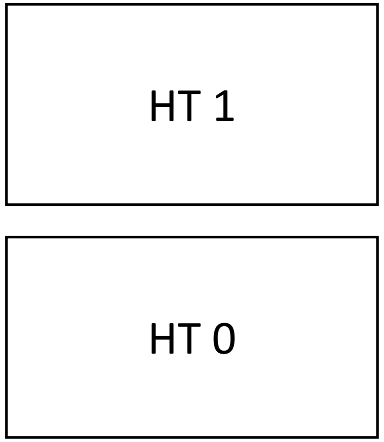
- A good way is a stack of hash tables:



How to implement a symbol table?

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

`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:

HT 0

Stack of hash tables

How to implement a symbol table?

- Example

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

HT 0

Stack of hash tables

See you on Friday!

- We will discuss parser generators