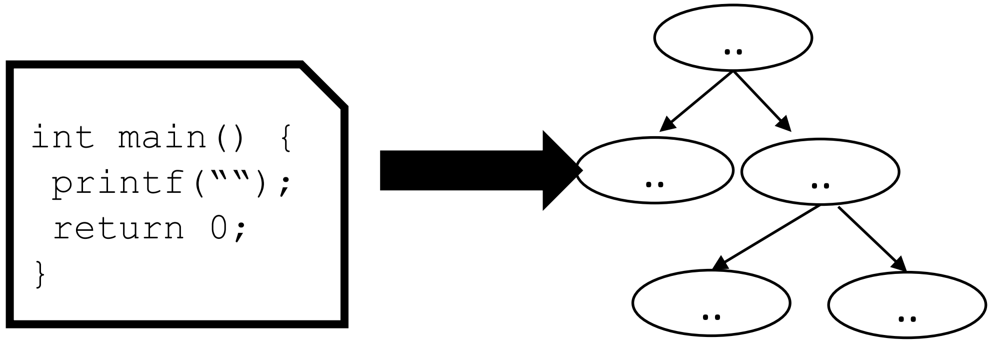
CSE110A: Compilers

April 29, 2024

Topics:

- Last day of module 2
- *Syntactic Analysis continued*
    - *Recursive decent parsing revisited*
    - *Symbol tables*



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

Announcements

- HW 2 is due on **Tonight** by midnight
- For part 3: you only need follow sets for rules that have “” in them
- For help
  - 1 Office hour before midnight
  - Ask on Piazza
    - ***No guaranteed help over the weekend or off business hours***

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

Homework questions?

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

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

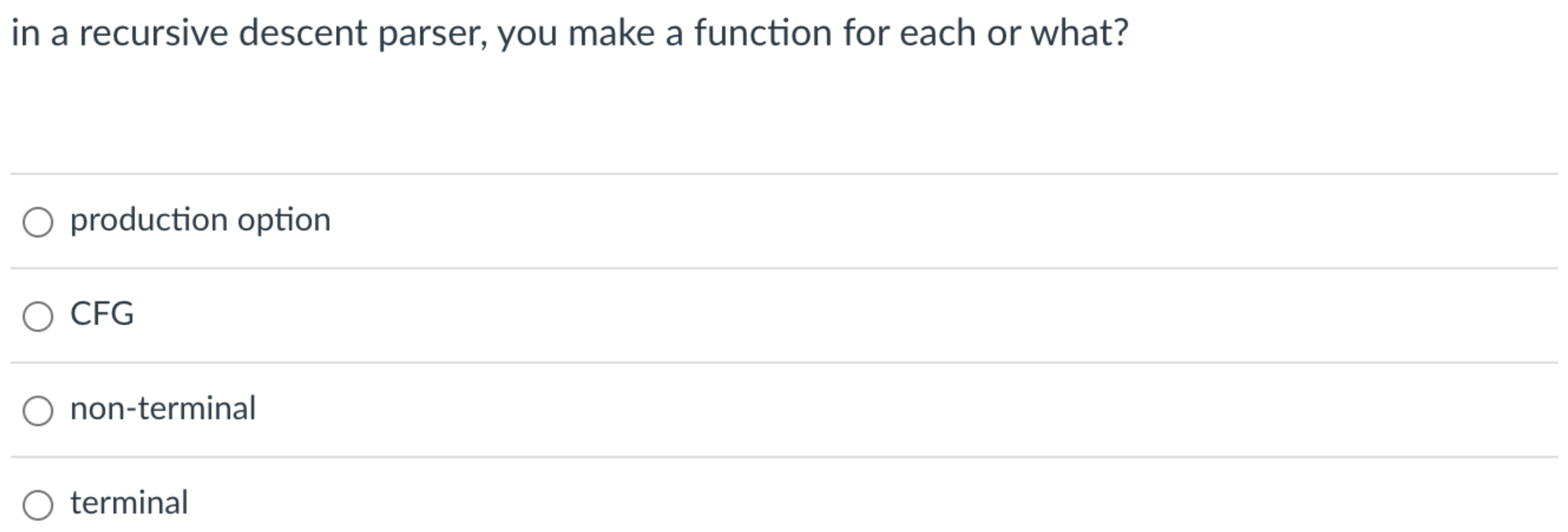
Quiz

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

```
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:          |   ‘*’
```

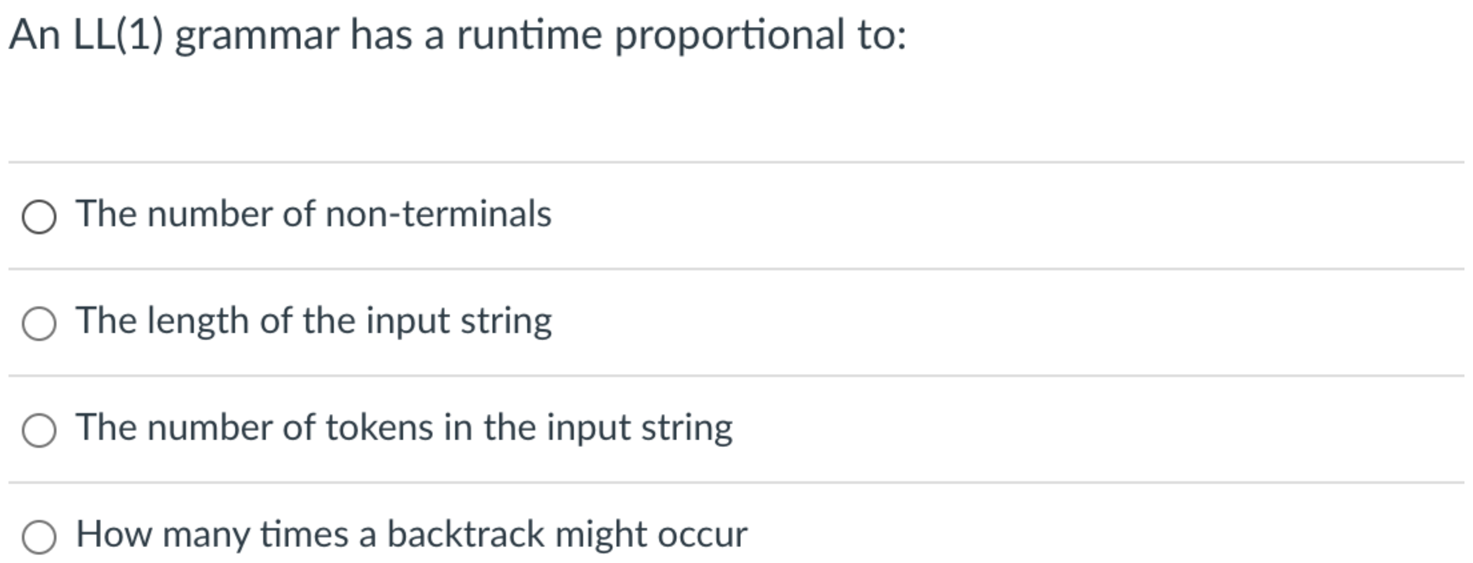
How do we parse a Unit?

```
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: {‘*’}
```

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

New material

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

start a new scope S   remove the scope S

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

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.`

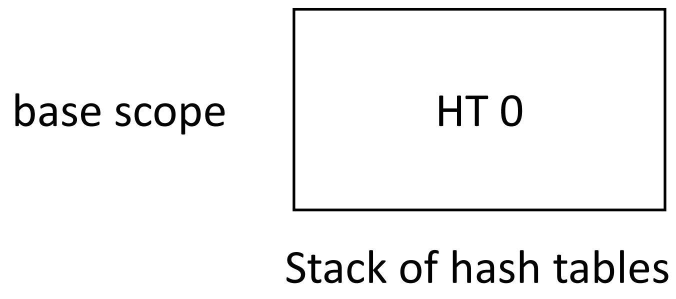
  - **`insert(id,info)`** `: insert a new id into the symbol table along with a set of information about the id.`

  - **`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?

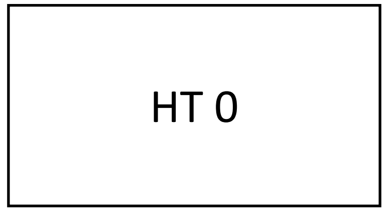
- Many ways to implement:
- A good way is a stack of hash tables:



How to implement a symbol table?

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

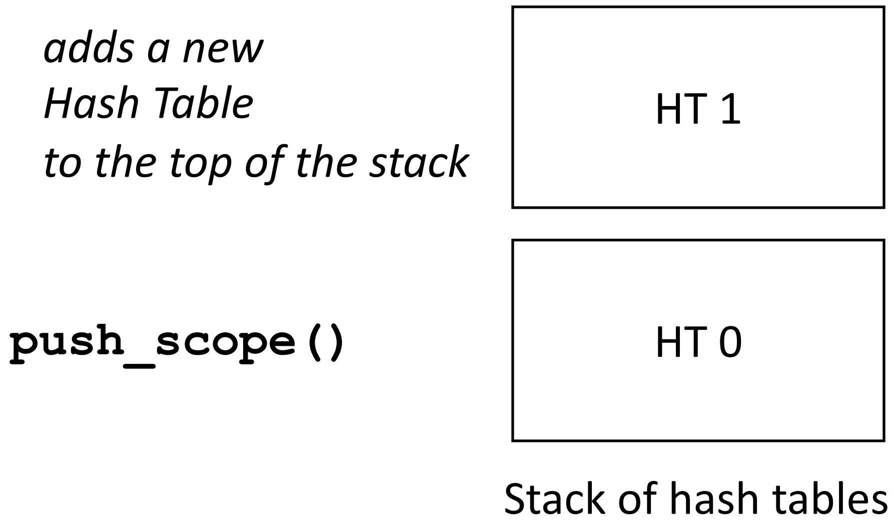
`push_scope()`



Stack of hash tables

How to implement a symbol table?

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

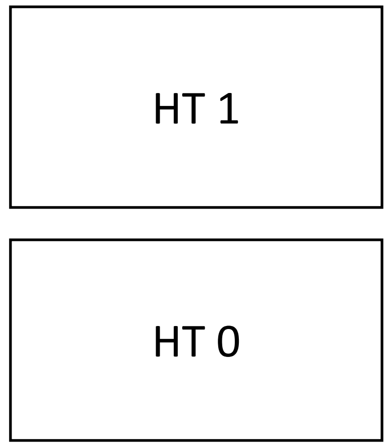


How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`insert(id,data)`



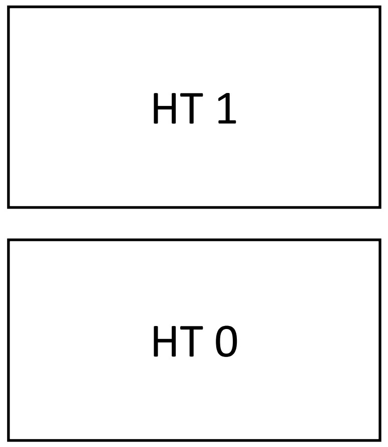
Stack of hash tables

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)`



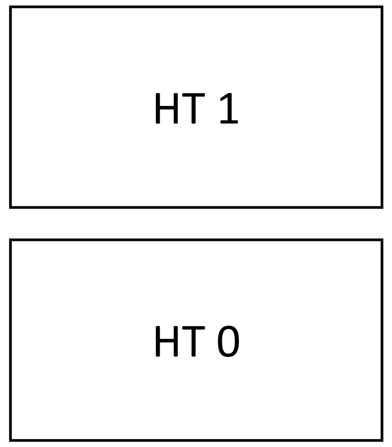
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`lookup(id)`



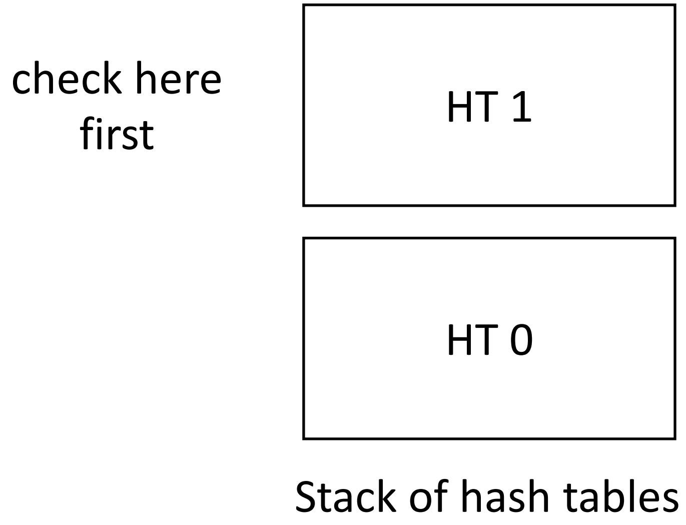
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

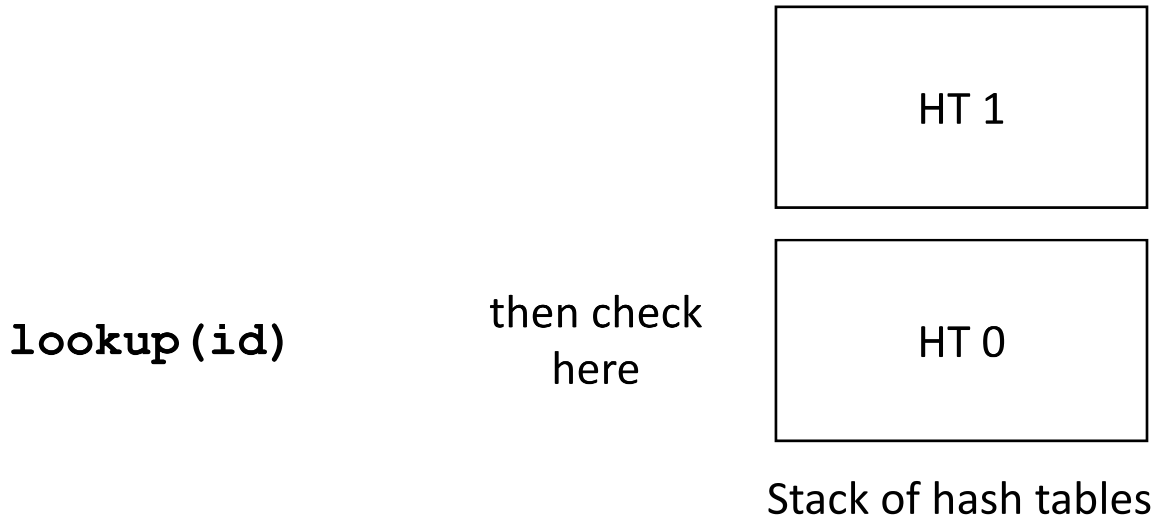
`lookup(id)`



How to implement a symbol table?

- Many ways to implement:

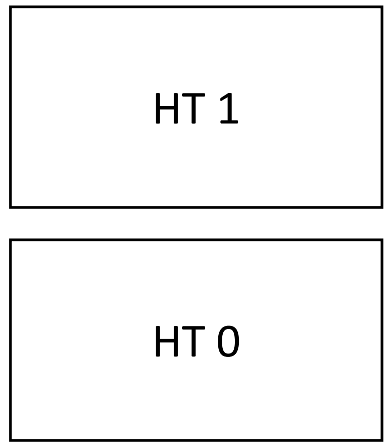
- A good way is a stack of hash tables:



How to implement a symbol table?

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

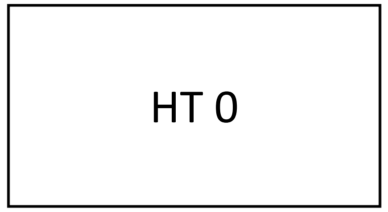
`pop_scope()`



Stack of hash tables

How to implement a symbol table?

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

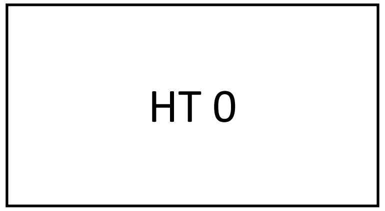


Stack of hash tables

How to implement a symbol table?

- Example

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



Stack of hash tables

Parser actions

Parser actions

- Like token actions: perform an action each time a production option is matched.

- Typically performed after the entire production action is matched

- Useful for:
  - tracking state

Example

- `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)
}
```

If we wrote our own recursive descent parser we can implement our own actions inlined

Example

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

- `SymbolTable ST;`

Parser actions would be written like this

```
                       $1   $2   $3
declare_statement ::= TYPE ID SEMI
{
  ST.insert($2, None);
}
```

result of each symbol. For a terminal it will be the value

always some way to refer to symbol value, e.g. an array

What values get returned from non-terminals?

```
1: Expr   ::= Expr '+' Unit     {print $1}
2:          |   Expr '-' Unit
3:          |   Unit
4: Unit   ::= '(' Expr ')'
5:          |    NUM
```

What does this print?

What values get returned from non-terminals?

```
1: Expr   ::= Expr '+' Unit     {print $1; return "expr"}
2:          |   Expr '-' Unit     {return "expr"}
3:          |   Unit              {...}
4: Unit   ::= '(' Expr ')'
5:          |    NUM
```

Each production rule needs to return something

What values get returned from non-terminals?

building a calculator

```
1: Expr  ::= Expr '+' Unit    {}
2:        |   Expr '-' Unit    {}
3:        |   Unit             {}
4: Unit  ::= '(' Expr ')'     {}
5:        |    NUM             {}
```

What values get returned from non-terminals?

building a calculator

```
1: Expr  ::= Expr ‘+’ Unit    {return $1 + $3}
2:        |   Expr ‘-’ Unit    {return $1 - $3}
3:        |   Unit             {return $1}
4: Unit  ::= ‘(‘ Expr ‘)’     {return $2}
5:        |    NUM             {return $1}
```

Shortcomings of parser actions

Difficult to perform actions in the middle of a production

- `SymbolTable ST;`

statement : LBRAC statement_list RBRAC

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

start a new scope S

remove the scope S

Example