CSE211: Compiler Design
Oct. 11, 2023

- **Topic**: Parser Generator Example 2

- **Questions**:
  - *Given token definitions, how would you implement a scanner?*

  - *What can you use token actions for?*

from: https://en.wikipedia.org/wiki/Yak

Announcements

- Homework 1 is out!
  - Please partner up if you haven't. If you don't have a partner you can make a private post on Piazza. Please do that in the next few days.
  - Failing to find a partner by the end of the week will be a 20% deduction and you will have to do the homework assignment by yourself.
  - I will make a shared spreadsheet that we can use to record partners
  - Please self organize (use Piazza)
  - You will have (slightly more than) 2 weeks to do the homework

Announcements

- Homework 1 is out!
  - Where we are at now:
  - The homework has you using PLY to parse 2 languages
    - A calculator language
    - A regular expression language
  - You should have been able to write tokens for each
  - At the end of the lecture today you should be able to parse each
  - You might need Friday's lecture for the symbol table needed to finish part 1
  - Early lectures next week will cover parsing by derivatives needed for part 2

Announcements

- Think about paper review
  - You will need to approve a paper with me by Oct. 23
  - First review is due Oct. 30
  - You should probably not wait until these due dates because the midterm is also on Oct. 30.
  - I give this time for you to organize, not as a guidance!
  - You can discuss papers on piazza or ask me for suggestions

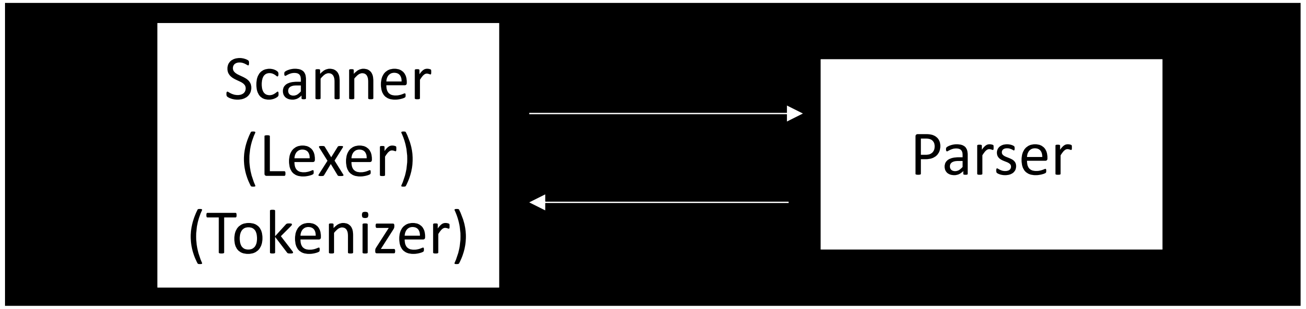
Announcements

- I will have office hours this week: Thursday from 3 – 5 PM
- Rithik has office hours too

Review

Parser architecture

Parser



First level of abstraction. Transforms a string of characters into a string of tokens

Second level: transforms a string of tokens in a tree of tokens.

***Language**:*
Regular Expressions (REs)

***Language**:*
Context-Free Grammars (CFGs)

PLY Parser Generator

- An implementation of Lex and Yacc in Python

- links:
  - source: https://github.com/dabeaz/ply
  - docs: https://ply.readthedocs.io/en/latest/

Review PLY code for Lexer

New material

How to handle keywords and ids

- How to tokenize: `if (x)`

How to handle keywords and ids

```
reserved = {
   'if'         : 'IF',
   'else'       : 'ELSE'
}

tokens = ["ID"] + list(reserved.values())


def t_ID(t):
    "[a-zA-Z]+"
    t.type = reserved.get(t.value, 'ID')
    return t
```

This will work!

Multiline calculator example

- For this, we will use the lexer and parser

- input:
  - 1 or more mathematical expressions separated by a ;
  - mathematical expressions can have non-negative integers as operands
  - mathematical operators are +,-,*,/ and ()

- output:
  - the solution to each expression

Reminder: Production rules vs production actions

- Great to check if a string is grammatically correct

- But can the production rules actually help us with compilation??

Production actions

- Each production *option* is associated with a code block
  - It can use values from its children
  - it returns a value to its parent
  - Executed in a post-order traversal (natural order traversal)

Production actions

input: 1+5*6

Example: executing a mathematical expression during parsing

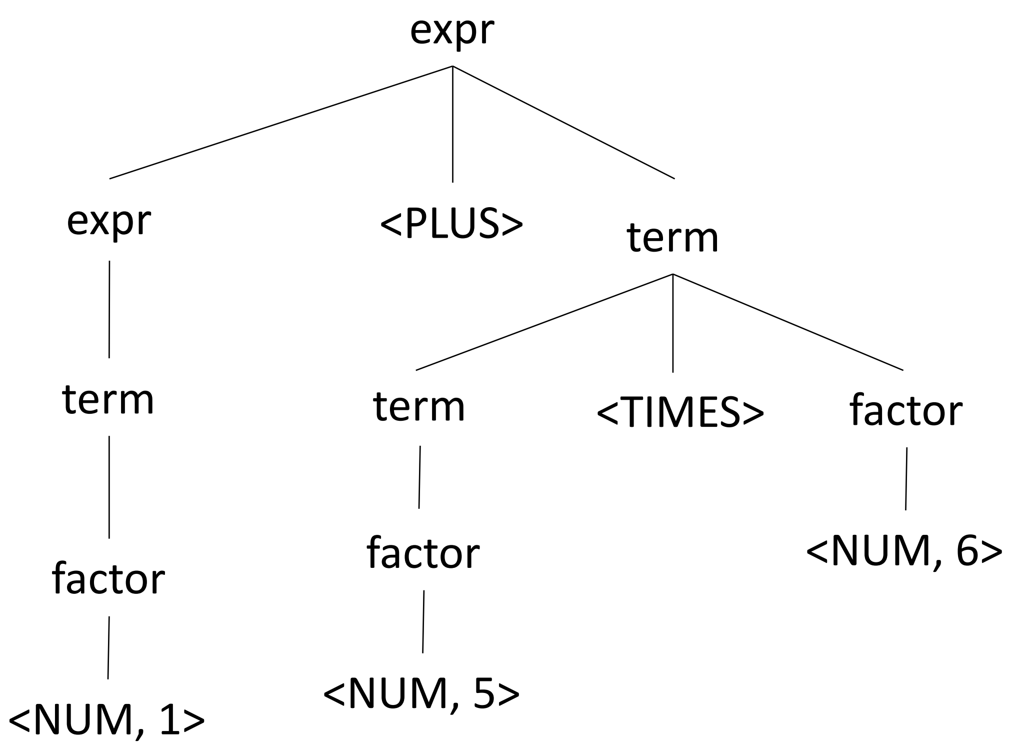
Children values are passed in as an array `C`, indexed from left to right

Operator	Name	Productions	Actions

+,-	expr	: expr PLUS term
		| expr MINUS term
		| term
			`{ret C[0] + C[2]}`
			`{ret C[0] - C[2]}`
			`{ret C[0]}`

*,/	term	: term TIMES factor
		: term DIV factor
		| factor
			`{ret C[0] * C[2]}`
			`{ret C[0] / C[2]}`
			`{ret C[0]}`

()	factor	: LPAR expr RPAR | NUM	`{ret C[1]}` `{ret int(C[0])}`



We have just implemented a simple arithmetic interpreter!

Multiline calculator example

```
import ply.lex as lex

tokens = ["NUM", "MULT", "PLUS", "MINUS", "DIV", "LPAR", "RPAR", "SEMI", "NEWLINE"]

t_NUM = '[0-9]+'
t_MULT = '\*'
t_PLUS = '\+'
t_MINUS = '-'
t_DIV = '/'
t_LPAR = '\('
t_RPAR = ‘\)'
t_SEMI = ";"

t_ignore = ' '

def t_NEWLINE(t):
    "\\n"
    t.lexer.lineno += 1

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    exit(1)

lexer = lex.lex()
```

Set up the lexer

Multiline calculator example

- *Import the library*

```
import ply.yacc as yacc
```

- Simple rule

```
def p_expr_num(p):
    "expr : NUM"
    p[0] = int(p[1])
```

functions are given prefixed by `p_`

production rules are the doc string

return values are stored in p[0]
children values are in p[1], p[2], etc.

Multiline calculator example

- *Try it out*

```
parser = yacc.yacc(debug=True)

result = parser.parse("5")
print(result)
```

What about trying to parse “a”?
What about trying to parse “+”?

Multiline calculator example

- *Next rule*

```
def p_expr_plus(p):
    "expr : expr PLUS expr"
    p[0] = p[1] + p[3]
```

- Try it again

```
result = parser.parse("5 + 4")
print(result)
```

What errors are we getting? Can we look into them?

Multiline calculator example

- *Set an error function*

```
def p_error(p):
    print("Syntax error in input!")
```

- Set associativity (and precedence)

```
precedence = (
    ('left', 'PLUS'),
)
```

Multiline calculator example

- *Next rules*

```
def p_expr_minus(p):
    "expr : expr MINUS expr"
    p[0] = p[1] - p[3]


def p_expr_mult(p):
    "expr : expr MULT expr"
    p[0] = p[1] * p[3]


def p_expr_div(p):
    "expr : expr DIV expr"
    p[0] = p[1] / p[3]
```

```
precedence = [
    ('left', 'PLUS', 'MINUS'),
    ('left', 'MULT', 'DIV'),
]
```

Multiline calculator example

- *Last rule for expressions*

```
def p_expr_par(p):
    "expr : LPAR expr RPAR"
    p[0] = p[2]
```

Multiline calculator example

- *An extra we can easily implement*

```
def p_expr_div(p):
    "expr : expr DIV expr"
    if p[3] == 0:
        print("divide by 0 error:")
        print("cannot divide: " + str(p[1]) + " by 0")
        exit(1)
    p[0] = p[1] / p[3]
```

Multiline calculator example

- *Combining rules:*

```
def p_expr_plus(p):
    "expr : expr PLUS expr"
    p[0] = p[1] + p[3]


def p_expr_minus(p):
    "expr : expr MINUS expr"
    p[0] = p[1] - p[3]


def p_expr_mult(p):
    "expr : expr MULT expr"
    p[0] = p[1] * p[3]
```

```
def p_expr_bin(p):
    """
    expr : expr PLUS expr
         | expr MINUS expr
         | expr MULT expr
    """
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    else:
        assert(False)
```

Multiline calculator demo using lambdas

- demo

One consideration: 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
- Global object, accessible (and mutable) by all production actions

- 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

VARIABLE_NAME = “[a-z]+”

INCREMENT = “\+\+”

TYPE = “int”

LB = “{“

RB = “}”

SEMI = “;”

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

statements are either a declaration or an increment

a very simple programming language

VARIABLE_NAME = “[a-z]+”

INCREMENT = “\+\+”

TYPE = “int”

LB = “{“

RB = “}”

SEMI = “;”

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

statements are either a declaration or an increment

a very simple programming language

VARIABLE_NAME = “[a-z]+”

INCREMENT = “\+\+”

TYPE = “int”

LB = “{“

RB = “}”

SEMI = “;”

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

statements are either a declaration or an increment

How to track scope?

- `SymbolTable ST;`

declare_variable: TYPE VARIABLE_NAME SEMI

```
{}
```

Say we have matched string:
`int x;`

`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;`

declare_variable: TYPE VARIABLE_NAME SEMI

```
{ST.insert(C[1],C[0])}
```

Say we are matched string:
`int x;`

In this example we are storing a type

How to track scope?

- `SymbolTable ST;`

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

variable_inc: VARIABLE_NAME 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?

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

- `SymbolTable ST;`

variable_inc: VARIABLE_NAME INCREMENT SEMI

```
{if not ST.lookup(x):
    raise SymbolTableException;
 else:
    ... // continue}
```

How to track scope?

- `SymbolTable ST;`

```
statement : variable_inc
          | declare_variable

statement_list : statement_list statement
               | statement
```

How to track scope?

- `SymbolTable ST;`

statement : variable_inc
| declare_variable

statement_list : statement_list statement
| statement

adding in scope

How to track scope?

- `SymbolTable ST;`

```
statement : variable_inc
          | declare_variable
          | LBAR statement_list RBAR

statement_list : statement_list statement
               | statement
```

How to track scope?

- `SymbolTable ST;`

statement : LBAR statement_list RBAR

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 : LBAR statement_list RBAR

start a new scope S

remove the scope S

Think about how to solve with production rules

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.`

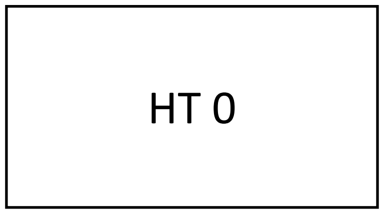
  - **`push_scope()`** : `push a new scope to the symbol table`

  - **`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



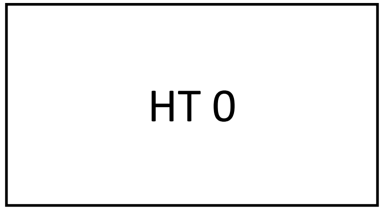
Stack of hash tables

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

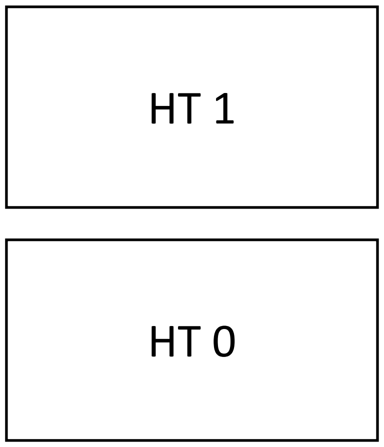
How to implement a symbol table?

- 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()`



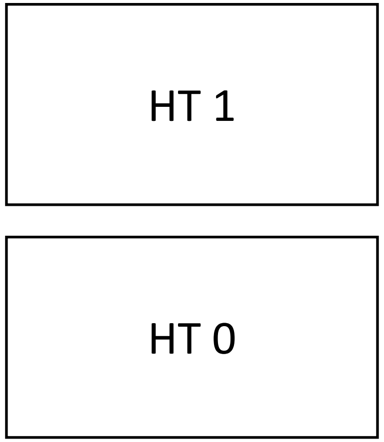
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`insert(id,data)`



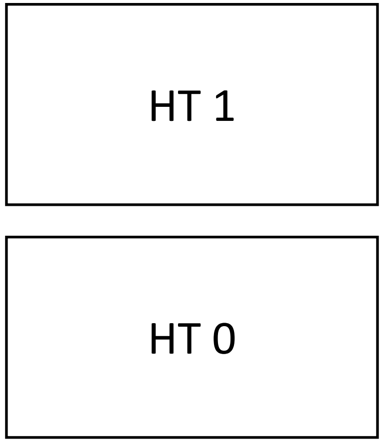
Stack of hash tables

How to implement a symbol table?

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

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

`insert(id,data)`

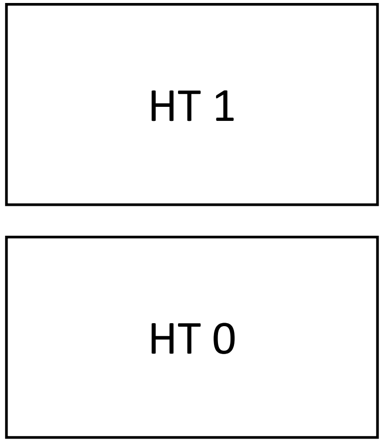


Stack of hash tables

How to implement a symbol table?

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

`lookup(id)`

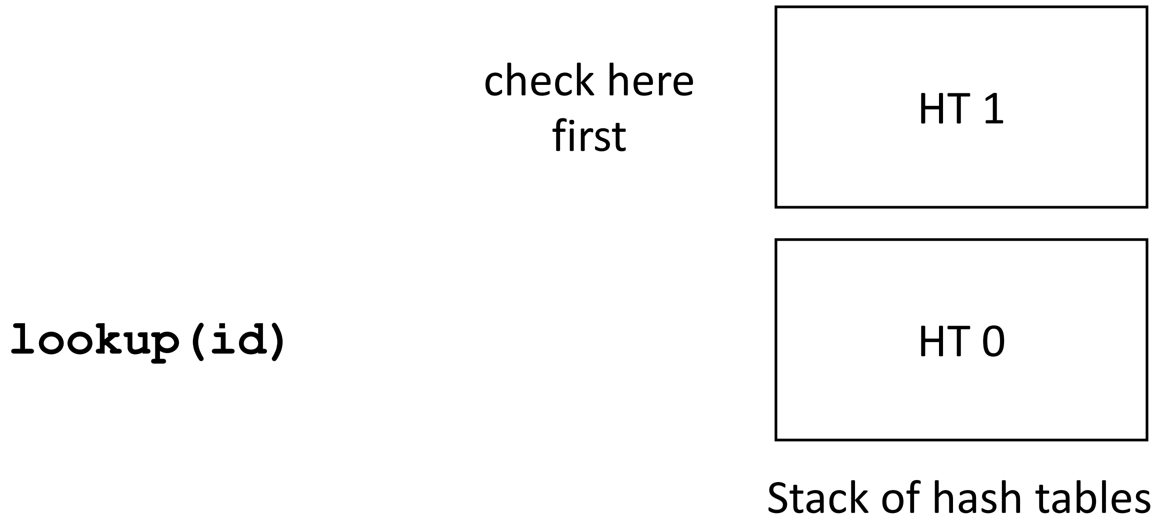


Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:



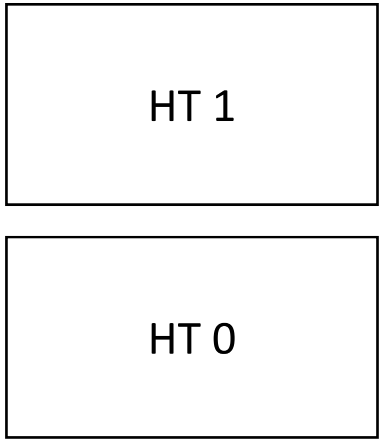
How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`lookup(id)`

then check here



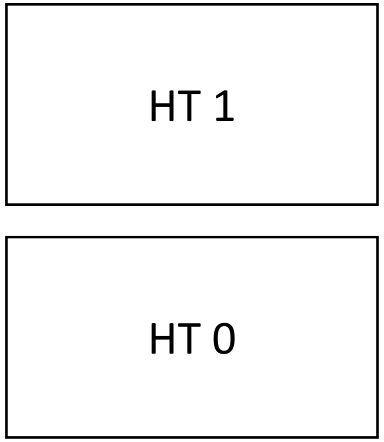
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`pop_scope()`

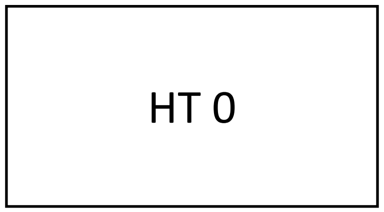


Stack of hash tables

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++;
```

HT 0

Stack of hash tables