CSE211: Compiler Design

Oct. 9, 2023

- **Topic**: Parser Generator Example (PLY)
- **Questions**:
  - *What is a parser generator?*
  - *Do you have any experience with a parser generator?*

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

Announcements

- Homework 1 is planned for release on Today by midnight
    - 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 2 weeks to do the homework

Announcements

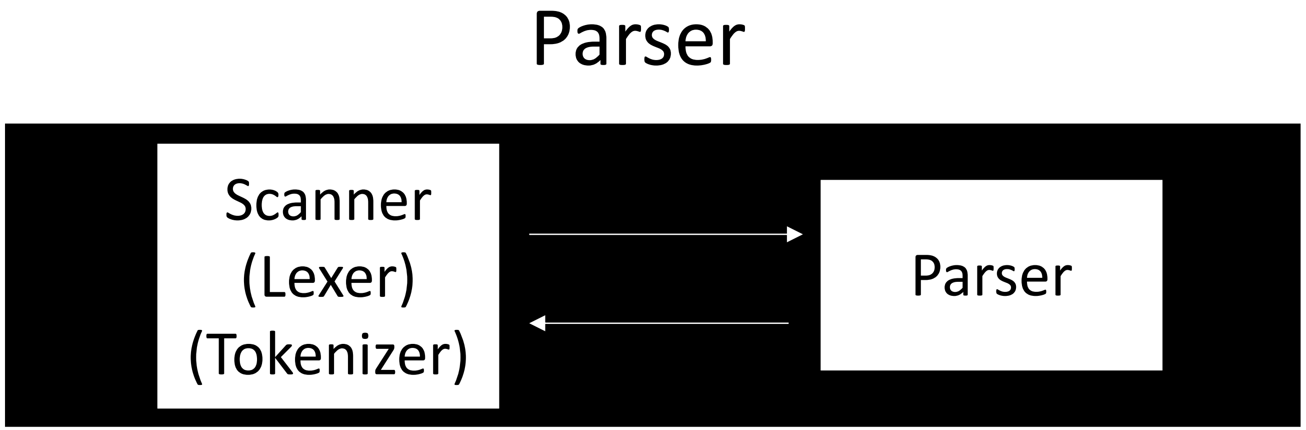
- Think about paper review
  - You will need to approve a paper with me by Oct. 23
  - First review is due Oct. 30
  - You should probably not wait until these due dates because the midterm is also on Oct. 30.
  - I give this time for you to organize, not as a guidance!
  - You can discuss papers on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

Announcements

- I will have office hours this week: Thursday from 3 – 5 PM
- Rithik will update the webpage and hold office hours as well

Review and a few thoughts from last time

Parser architecture



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)

Scanner

(5 + 4) * 8

Scanner

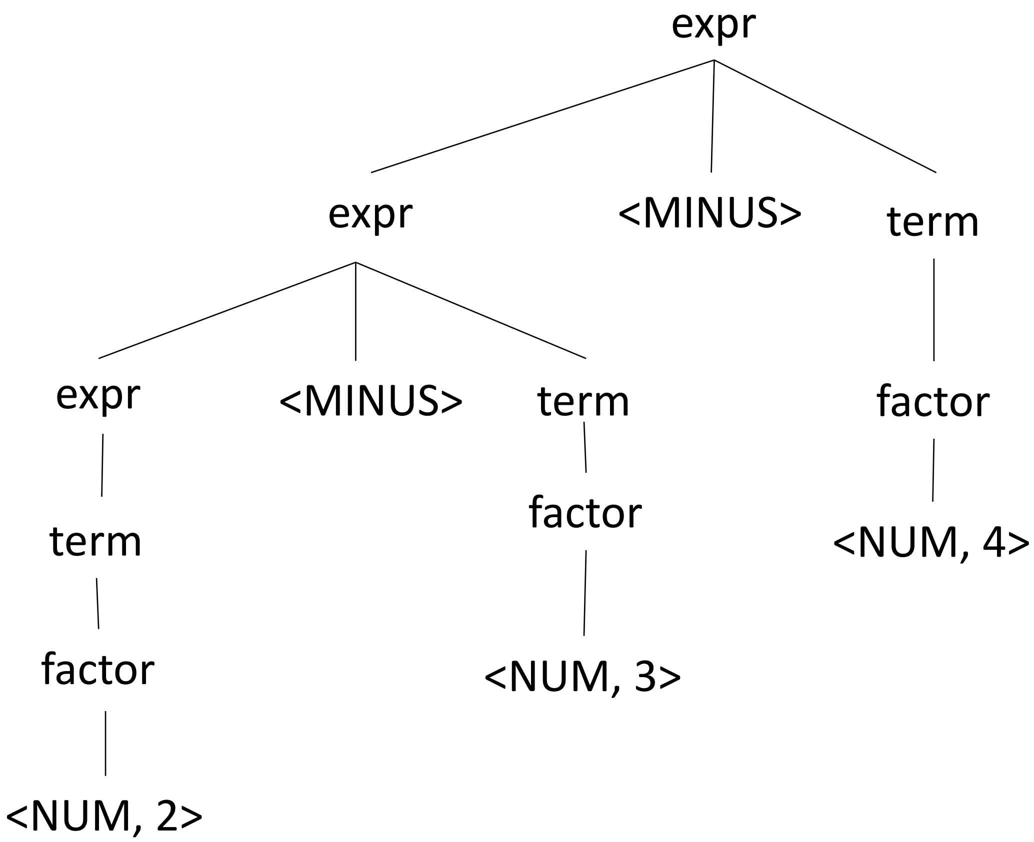
```
[[(LPAR, “(”) (NUM, “5”) (PLUS, “+”) (NUM, “4”) (RPAR, “)”)
 (TIMES, “*”) (NUM, “8”)]
```

Splits an input sentence it into lexemes

Parsing

Operator	Name	Productions
+,-	expr	: expr PLUS term | expr MINUS term | term
*,/	term	: term TIMES pow | term DIV pow | pow
^	pow	: factor CARROT pow | factor
()	factor	: LPAR expr RPAR | NUM

`input: 2-3-4`



Let’s make a richer grammar

Let’s add minus, division and power to our grammar

Operator	Name	Productions

Tokens:

NUM = [0-9]+
PLUS = ‘\+’
TIMES = ’*’
LP = ‘\(’
RP = \)’
MINUS = ‘-’
DIV = ‘/’
CARROT = ‘\^’

Let’s make a richer grammar

Let’s add minus, division and power to our grammar

Operator	Name	Productions
+,-	expr	: expr PLUS term | expr MINUS term | term
*,/	term	: term TIMES pow | term DIV pow | pow
^	pow	: factor CARROT pow | factor
()	factor	: LPAR expr RPAR | NUM

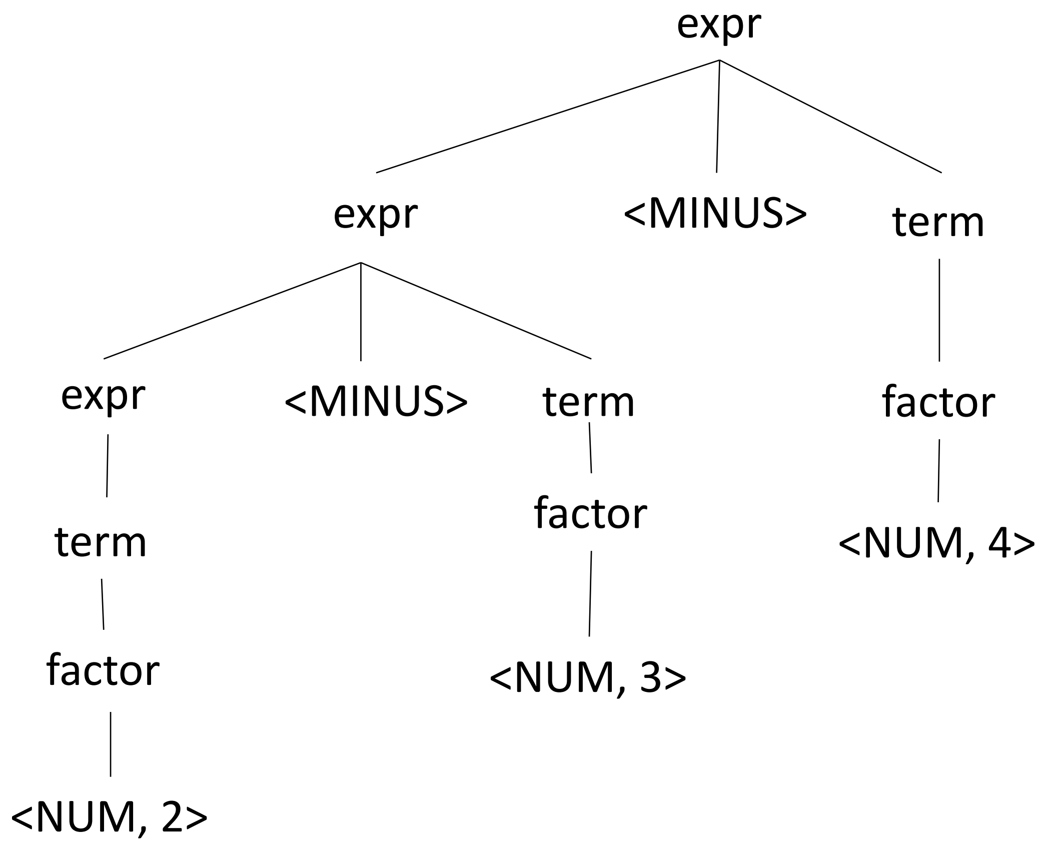
Tokens:

NUM = [0-9]+
PLUS = ‘\+’
TIMES = ’*’
LP = ‘\(’
RP = \)’
MINUS = ‘-’
DIV = ‘/’
CARROT =’ \^’

Let’s make a richer grammar

`input: 2-3-4`

Operator	Name	Productions
+,-	expr	: expr PLUS term | expr MINUS term | term
*,/	term	: term TIMES pow | term DIV pow | pow
^	pow	: factor CARROT pow | factor
()	factor	: LPAR expr RPAR | NUM



What do these look like in real-world languages?

- C++ : https://en.cppreference.com/w/cpp/language/operator_precedence

- Python: https://docs.python.org/3/reference/expressions.html#operator-precedence

Godbolt examples

New material

Production rules in a compiler

- 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)

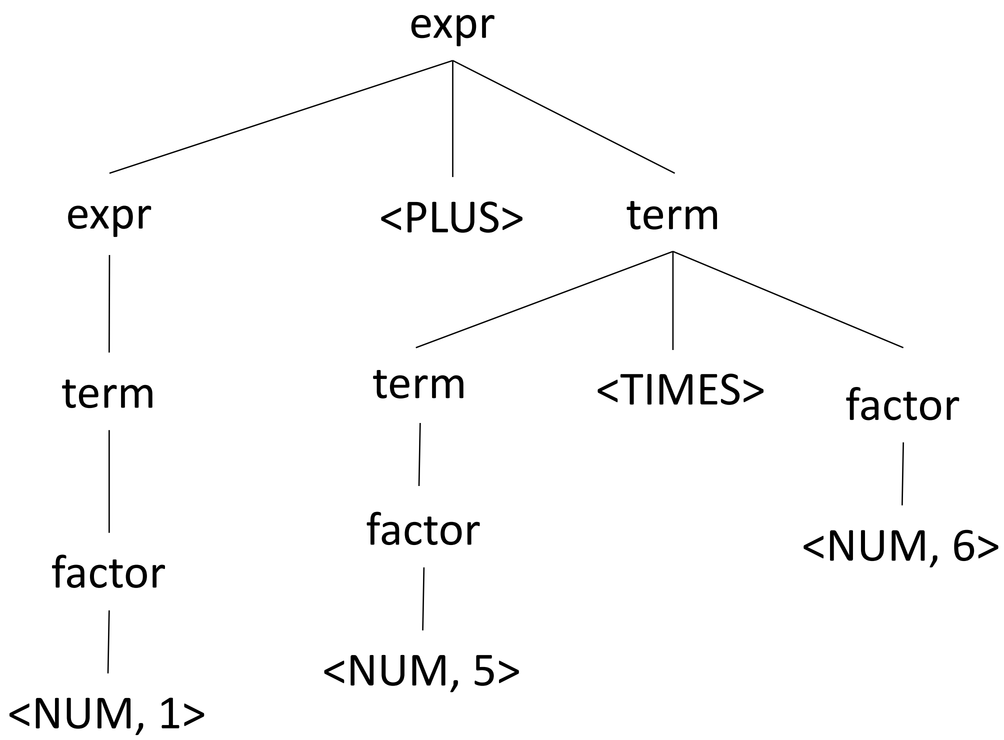
Production actions

`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	{ } { } { }
*,/	term	: term TIMES factor : term DIV factor | factor	{ } { } { }
()	factor	: LPAR expr RPAR | NUM	{ } { }



Production actions

input: 1+5*6

Example: executing a mathematical expression during parsing

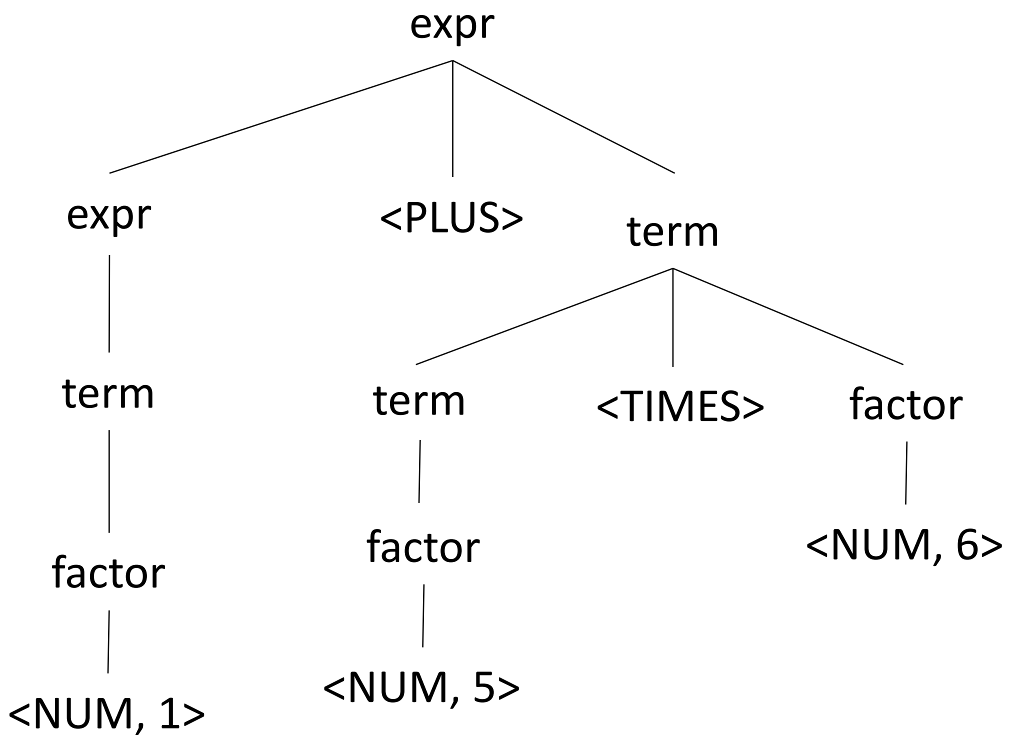
Children values are passed in as an array `C`, indexed from left to right

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!
Could this be in a compiler?

Parser generators

- Specify:
    - Tokens
    - Production Rules
    - Production Actions

- Parser generator gives you a function in which you can pass strings
    - Executes production actions
    - Error reporting

Historically

- Lex
  - lexer (scanner)
  - released in 1975
  - co-developed by Eric Schmidt
  - ”Flex” is a common open-source implementation
  - historically outputs a .c file

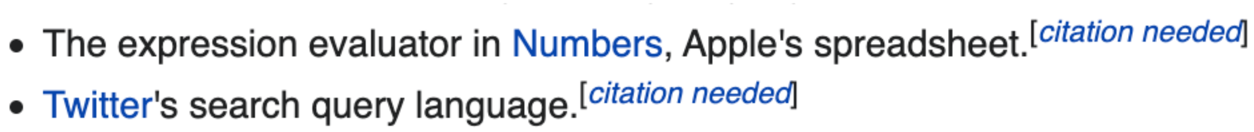
- Yacc (Yet Another Compiler Compiler)
  - parser
  - released in 1975
  - originally written in B, but soon rewritten in C
  - interface is widely supported, but newer implementations are more used now
  - historically outputs a .c file

Historically

- Bison
  - Parser only, often coupled with flex
  - Released in 1985: actively maintained
  - better error tracking and debugging
  - compatible with yacc rules
  - outputs C/++, Java

More modern

- Antlr
  - Lexer and Parser
  - Released 1992, actively maintained
  - BSD License
  - From Wikipedia, used in:



- The expression evaluator in Numbers, Apple's spreadsheet.[citation needed]
- Twitter's search query language.[citation needed]

-
  - Outputs: Python, Javascript, C#, Swift

- Others: https://en.wikipedia.org/wiki/Comparison_of_parser_generators

PLY

- An implementation of Lex and Yacc in Python

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

- Your homework augments this example in several ways:
  - *Variables, Scope, Precedence, Associativity*

Demo

- *Lots of thanks to the excellent PLY documentation! Some functions are copied from there*

- *Setup:*
  - *clone the ply repo*
  - *make a new directory*
  - *copy the ply/ directory into the directory*

A Simple Language

- ARTICLE = {The, A, My, Your}
- NOUN = {Dog, Car, Computer}
- VERB = {Ran, Crashed, Accelerated}
- ADJECTIVE = {Purple, Spotted, Old}

Lexer Demo

- *Library import*

```
import ply.lex as lex
```

- *Token list*

```
tokens = ["ADJECTIVE", "NOUN", "VERB", "ARTICLE"]
```

- *Token specification*

```
t_ADJECTIVE = "old|purple|spotted"
t_NOUN = "dog|computer|car"
t_ARTICLE = "the|my|a|your"
t_VERB = "ran|crashed|accelerated"
```

Lexer Demo

- *Build the lexer*

```
lexer = lex.lex()
```

- *Need an error function*

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

Lexer Demo

- *Now give the lexer some input*

```
lexer.input("dog")
print(lexer.token())
```

Lexer Demo

- *output:*

line number (1 indexed)

```
LexToken(NOUN, 'dog’, 1, 0)
```

number of characters streamed
(0 indexed)

- *try a longer string:*

```
lexer.input("dog computer")
```

What happens?

Lexer Demo

- *The lexer streams the input, we need to stream the tokens:*

```
# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break      # No more input
    print(tok)
```

Lexer Demo

- *Need to add a token for whitespace!*

```
tokens = ["ADJECTIVE", "NOUN", "VERB", "ARTICLE", "WHITESPACE"]
 …
t_WHITESPACE = '\ '
```

- *Now we can lex:*

```
LexToken(NOUN,'dog',1,0)
LexToken(WHITESPACE,' ',1,3)
LexToken(NOUN,'computer',1,4)
```

Lexer Demo

- *Now we can do a sentence*

```
lexer.input("my spotted dog ran")
```

```
LexToken(ARTICLE,'my',1,0)
LexToken(WHITESPACE,' ',1,2)
LexToken(ADJECTIVE,'spotted',1,3)
LexToken(WHITESPACE,' ',1,10)
LexToken(NOUN,'dog',1,11)
LexToken(WHITESPACE,' ',1,14)
LexToken(VERB,'ran',1,15)
```

Can we clean this up?

Lexer Demo

- *We can ignore whitespace*

```
#t_WHITESPACE = '\
t_ignore = ' '
```

No need for the \ because ignore is just characters, not a regex

```
LexToken(ARTICLE,'my',1,0)
LexToken(WHITESPACE,' ',1,2)
LexToken(ADJECTIVE,'spotted',1,3)
LexToken(WHITESPACE,' ',1,10)
LexToken(NOUN,'dog',1,11)
LexToken(WHITESPACE,' ',1,14)
LexToken(VERB,'ran',1,15)
```

gets simplified to:

```
LexToken(ARTICLE,'my',1,0)
LexToken(ADJECTIVE,'spotted',1,3)
LexToken(NOUN,'dog',1,11)
LexToken(VERB,'ran',1,15)
```

Lexer Demo

- *What about newlines?*

```
lexer.input("""
my spotted dog ran
the old computer crashed
""")
```

- *Need to add a newline token!*

Lexer Demo

- *What about newlines?*

```
lexer.input("""
my spotted dog ran
the old computer crashed
""")
```

- *Need to add a newline token!*

```
tokens = ["ADJECTIVE", "NOUN", "VERB", "ARTICLE", "NEWLINE"]

t_NEWLINE = "\\n"
```

Lexer Demo

```
LexToken(NEWLINE,'\n',1,0)
LexToken(ARTICLE,'my',1,1)
LexToken(ADJECTIVE,'spotted',1,4)
LexToken(NOUN,'dog',1,12)
LexToken(VERB,'ran',1,16)
LexToken(NEWLINE,'\n',1,19)
LexToken(ARTICLE,'the',1,20)
```

Line numbers are not updating

Lexer Demo

- *Token actions*

```
t_NEWLINE = "\\n"
```

Changes into:

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

docstring is the regex, lexer object which has a linenumber attribute.

If we don’t return anything, then it is ignored.

Lexer Demo

- *Example: changing a sentence into gender neutral*

```
tokens = ["ADJECTIVE", "NOUN", "VERB", "ARTICLE", "NEWLINE", "PRONOUN"]
t_PRONOUN = "her|his|their"
```

```
lexer.input("""
his spotted dog ran
her old computer crashed
""")
```

Lexer Demo

- *Add a token action:*

```
def t_PRONOUN(t):
    "her|his|their"
    if t.value in ["his", "her"]:
        t.value = "their"
    return t
```

Now output will have a gender neutral sentence!

How to handle keywords and ids

```
tokens = ["IF", "ELSE", "ID"]


t_ID = "[a-zA-Z]+"
t_IF = "if"
t_ELSE = "else"
t_ignore = ' '

def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    print("line number: %d" % t.lexer.lineno)
    exit(1)

lexer = lex.lex()

lexer.input("if")
```

parses ”if” as an ID!

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 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

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

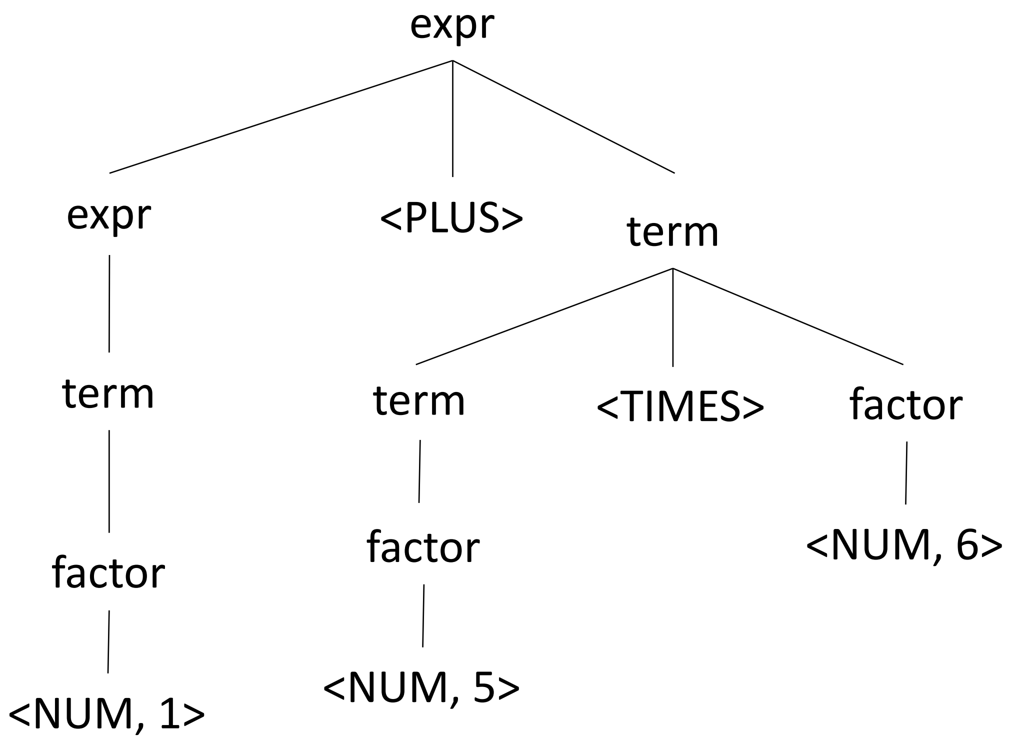
Children values are passed in as an array `C`, indexed from left to right

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)
```

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 are 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?

- `SymbolTable ST;`

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

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.`

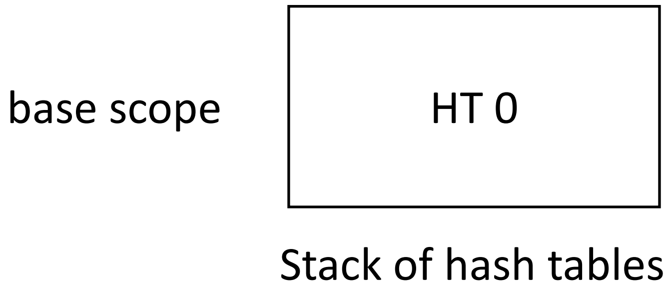
  - **`push_scope()`** `: push a new scope to the symbol table`

  - **`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:

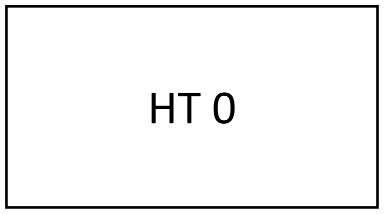


Stack of hash tables

How to implement a symbol table?

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

`push_scope()`

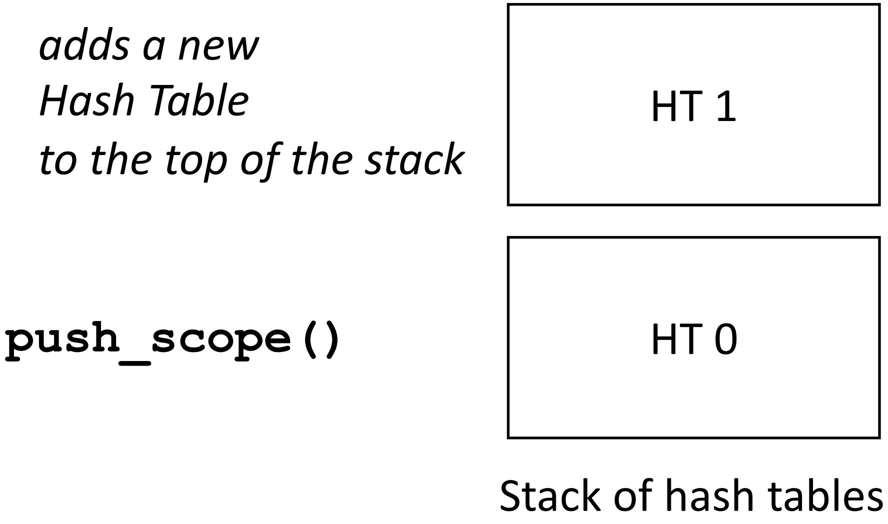


Stack of hash tables

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

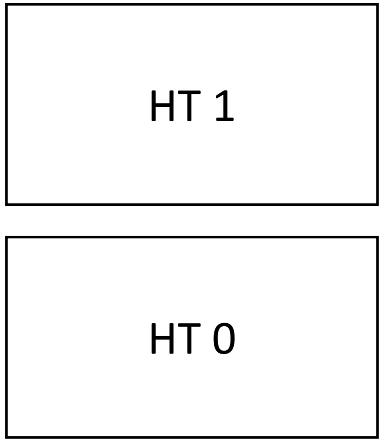
`push_scope()`

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

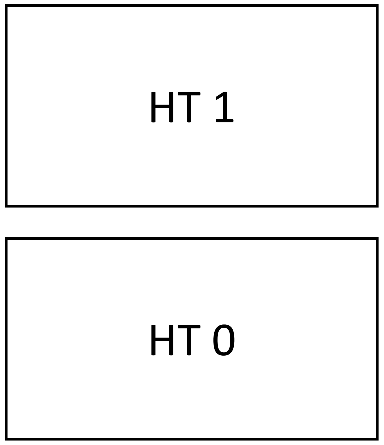
How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

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

`insert(id,data)`



HT 1

HT 0

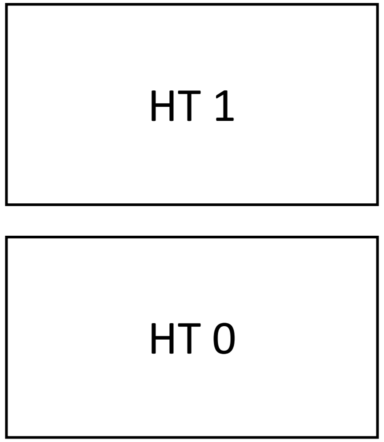
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`lookup(id)`



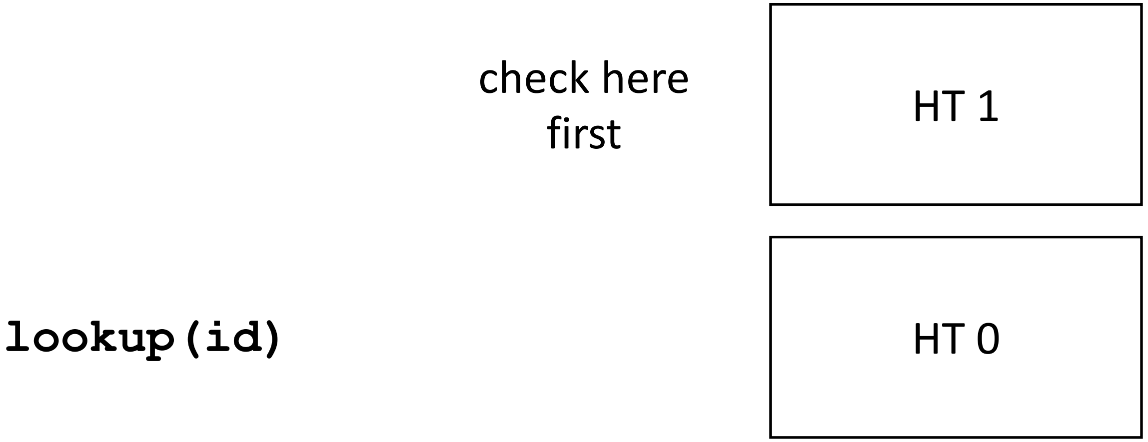
HT 1

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:



Stack of hash tables

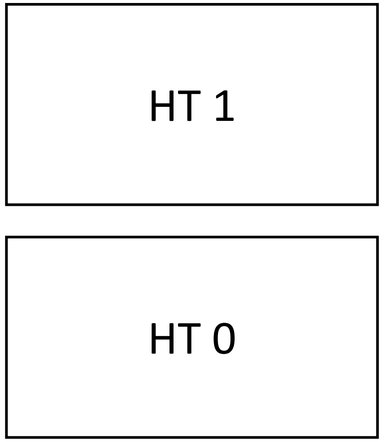
How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`lookup(id)`

then check here

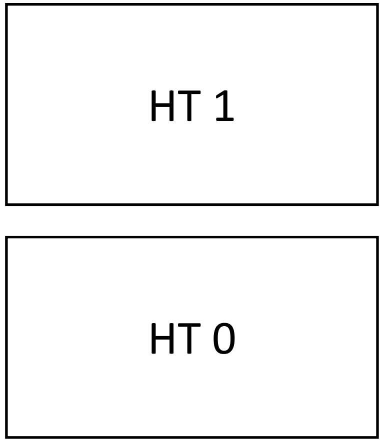


Stack of hash tables

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:

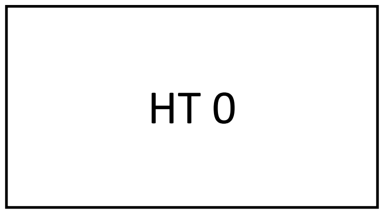
HT 0

Stack of hash tables

How to implement a symbol table?

- Example

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



Stack of hash tables