CS 152

Programming Paradigms

Overloading & Name Resolution, Allocations & Lifetimes

Today

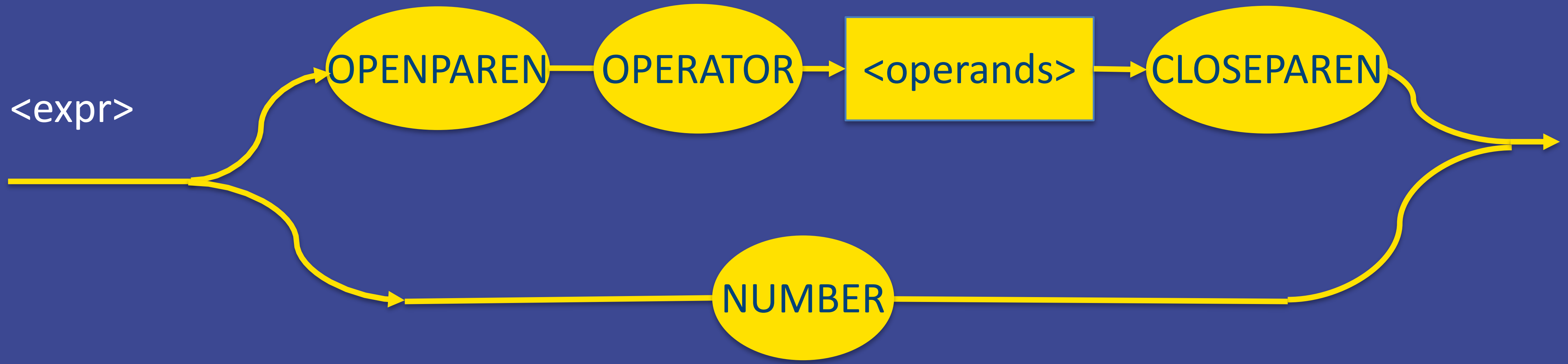
- Homework 6 and more about parsing
- Overloading
- Name Resolution
- Allocations and Lifetimes

Course Learning Outcomes

7. Understand variable scoping and lifetimes.
8. Write interpreters for simple languages that involve arithmetic expressions, bindings of values to names, and function calls.

Homework 6: Syntax Diagram

<expr> -> OPENPAREN OPERATOR <operands> CLOSEPAREN | NUMBER



The *pexpr* function returns a ParseTree:

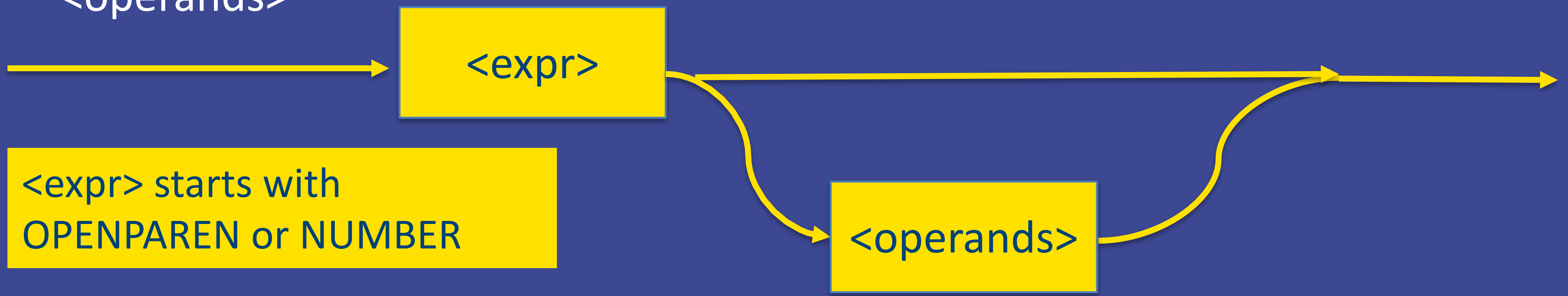
1. OpNode Char [ParseTree]
2. NumNode Float

Homework 6: Syntax Diagram

<operands> -> <expr> | <expr> <operands>

<operands> -> <expr> [<operands>]

<operands>



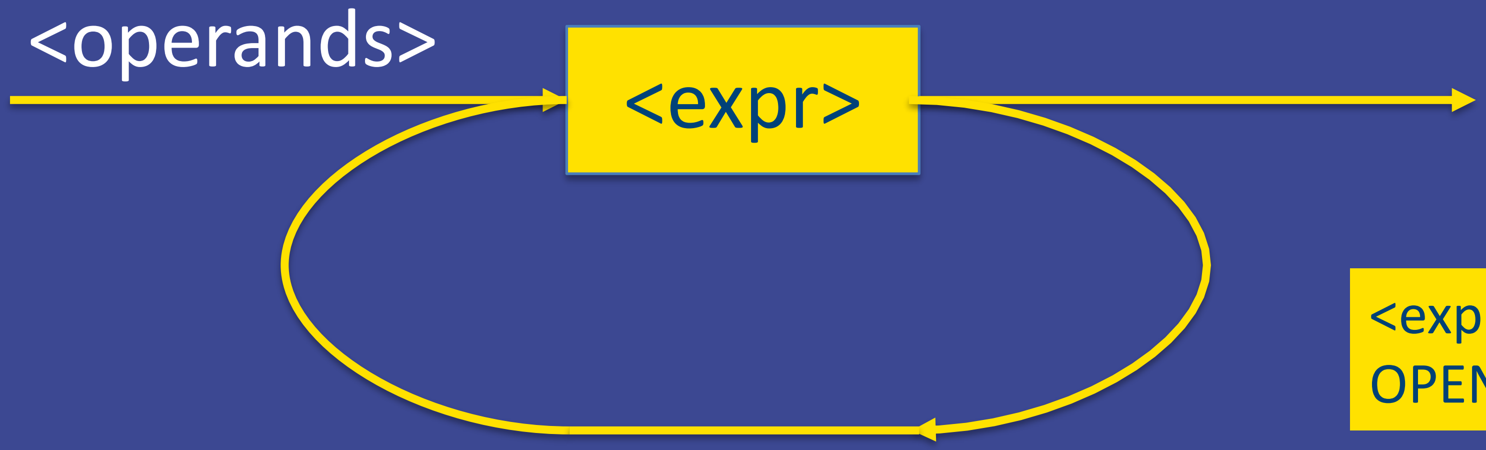
<expr> starts with
OPENPAREN or NUMBER

The *poperand* function returns a list of ParaseTrees

Alternate Syntax Diagram

<operands> -> <expr> | <operands> <expr>

<operands> -> <expr> {<expr>}



<expr> starts with
OPENPAREN or NUMBER

Predictive Parsers

- A predictive parser is a recursive descent parser that decides what production to use based only on the next (k) tokens.
- A recursive descent parser that decides what production to use based only on a single token is called a single-symbol lookahead parser

Predictive Parsers

Grammar must satisfy two conditions for predictive parsers to work. To formulate these conditions, we'll define:

First(α): set of tokens that can begin the string α

<expr> -> OPENPAREN OPERATOR <operands> CLOSEPAREN

| NUMBER

First(<expr>) = {OPENPAREN , NUMBER}

Follow(α): set of tokens that can follow the string α

Follow(<operands>) = {CLOSEPAREN}

Predictive Parsers

Grammar must satisfy two conditions for predictive parsers

1. Parser must be able to distinguish between choices in a rule

   $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \dots \mid \alpha_n$

   $First(\alpha_i) \cap First(\alpha_j) = \emptyset$ for all $i \neq j$

2. For an optional part, no token beginning the optional part can also come after the optional part

   $A \rightarrow B\ [\alpha]\ C$

   $First(\alpha) \cap Follow(\alpha) = \emptyset$

Top-down Parsing Limitations

- Top-down parsers must decide which production to use, based only on the next (k) tokens.
- That places restrictions on the grammars they support.

Bottom-up Parsers

- A bottom-up parser is able to postpone the decision until it has seen:
  - input tokens corresponding to the entire right side of the production
  - and some lookahead tokens beyond (to avoid backtracking)

Bottom-up Parser

- Bottom-up parsers are also called shift-reduce parsers
  - They shift tokens onto a stack prior to reducing strings to non-terminals
- Build derivations and parse trees from the leaves to the roots
- Match an input with right side of a rule and reduces it to the non-terminal on the left

What is Overloading?

- Associating more than one meaning with the same name/identifier

Operator Overloading

Some languages (such as Python and C++, but not Java) allow built-in operators to be overloaded

>>> 4 + 3

7

>>> "Go " + "Spartans!"

'Go Spartans!'

Operator Overloading

We can also use Python magic methods to specify the behavior of the + operator (or any other operator) on any user defined object.

Operator Overloading

```
class Account:
    """
    Represent a bank account.

    Argument:
    account_holder (string): account holder's name.
    balance (float): account balance in dollars.

    Attributes:
    holder (string): account holder's name.
    balance (float): account balance in dollars.
    """
```

Operator Overloading

```
def __init__(self, account_holder, balance):
    self.holder = account_holder
    self.balance = balance

def __add__(self, other):
    new_name = f'{self.holder} and {other.holder}'
    new_balance = self.balance + other.balance
    new_account = Account(new_name, new_balance)
    return new_account
```

Operator Overloading

```
>>> alex_acc = Account('Alex', 60)
>>> zoe_acc = Account('Zoe', 100)
>>> joint_account = alex_acc + zoe_acc
>>> print(joint_account.holder)
Alex and Zoe
>>> print(joint_account.balance)
160
```

Operator Overloading

```
>>> 4 + 3
7
>>> "Go " + "Spartans!"
'Go Spartans!'
>>> joint_account = alex_acc + zoe_acc
```

How can the translator determine what the + means?

Translator must look at the data type of each operand to determine which operation to carry out.

Potential Ambiguity?

Python:

```
>>> "Go " + 3
TypeError: Can't convert 'int' object to str implicitly
```

JavaScript:

```
>"Go " + 3
"Go 3"
```

=> Implicit conversion

Function Overloading

- Some programming languages allow function overloading.
- The same name is used for two or more functions that take a different number of parameters or a different type of parameters.
- C++, Java and Haskell (with type classes) allow function overloading.

Function Overloading in C++

```
int max(int x, int y) {      // max #1
    return x > y ? x : y;
}

double max(double x, double y) {    // max #2
    return x > y ? x : y;
}

int max(int x, int y, int z) {  // max # 3
    return x > y ?  (x > z ? x : z) : (y > z ? y : z) ;
}
```

max(6.2, 9.8)

A. max # 1 is called
B. max # 2 is called
C. max # 3 is called

Function Overloading in C++

```
int max(int x, int y) {     // max #1
    return x > y ? x : y;
}

double max(double x, double y) {    // max #2
    return x > y ? x : y;
}
int max(int x, int y, int z) {  // max # 3
    return x > y ?  (x > z ? x : z) : (y > z ? y : z) ;
}
```

max(6 , 80, 10)

A. max # 1 is called
B. max # 2 is called
C. max # 3 is called

Name Resolution

- Overload resolution: the process of choosing a unique function among many with the same name
- We can determine the appropriate function based on the calling context
- Calling context: the information contained in each call
- Here the calling context consists of number and type of parameters
- Lookup operation of the symbol table must search on name plus number and type of parameters

Ambiguity?

```
int max(int x, int y) {     // max #1
    return x > y ? x : y;
}

double max(double x, double y) {    // max #2
    return x > y ? x : y;
}
int max(int x, int y, int z) {  // max # 3
    return x > y ?  (x > z ? x : z) : (y > z ? y : z) ;
}
```

max(5.1, 10)

A. max # 1 is called
B. max # 2 is called
C. max # 3 is called
D. IDK

Implicit Conversions

- Implicit conversions in C++:
  - integer -> double (widening conversion)
  - double -> integer (narrowing conversion)

max(5.1, 10)
is ambiguous

- Implicit conversions in Java:
  - integer -> double
  - ~~double -> integer~~

max(5.1, 10)
max(5.1, 10.0)
max # 2 is called

- Implicit conversions complicate name resolution

More Overloading to solve Ambiguity

```
double max(double x, int y) {     // max #4
    return x > y ? x : y;
}

double max(int x, double y) {     // max #5
    return x > y ? x : y;
}
```

max(5.1, 10)

A. max # 1 is called
B. max # 2 is called
C. max # 4 is called
D. max # 5 is called
E. IDK

Function Overloading in Python?

Python does not support function overloading because there is no need for that in the language:

- We can define one function in Python that accepts an arbitrary number of parameters.
- The function parameters in Python do not have a type associated with them.

Default Values for Parameters

```
def average(first, second, third=None):
    if third is None:
        return (first + second) / 2
    else:
        return (first + second + third) / 3
>>> average(90, 100)
95.0
>>> average(90, 100, 80)
90.0
```

Arbitrary Number of Parameters

```
def average(*args):
    if args:
        return sum(args) / len(args)
    else:
        return 0
>>> average(90, 100)
95.0
>>> average(90, 100, 80)
90.0
>>> average(90, 100, 80, 70, 90)
86.0
```

Arbitrary Type of Parameters

The function parameters in Python do not have a type associated with them.

```
def median(a, b, c):
    sorted_list = sorted([a, b, c])
    return sorted_list[1]

print(median(5, 7, 2))
5
print(median("A", "C", "B"))
B
```

Name Overloading

- Some programming languages (such as Java) allow the use of the same name for entities of different types: a variable, a function, a type
- Separate symbol tables:
  - One symbol table for variables
  - One symbol table for functions
  - One symbol table for types

Environment

- Symbol table: a mapping from names to attributes (translation)

Symbol Table

Names $\longrightarrow$ Attributes

- Environment: a mapping from names to locations (runtime)

Environment

Names $\longrightarrow$ Locations

Environment

- Environment may be constructed statically (at load time), dynamically (at execution time), or with a mixture of both
- Not all names in a program are bound to locations
  - const int MAX = 90;
  - The compiler can simply replace all occurrences of MAX by 90

Allocations

- Typically, in a block-structured language:
  - Global variables are allocated statically
  - Local variables are allocated dynamically when the block is entered
- When a block is entered, memory for variables declared in that block is allocated
- When a block is exited, this memory is deallocated

Activation

- Memory for local variables within a function will not be allocated until the function is called
- Activation: a call to a function
- Activation record: the corresponding region of allocated memory

Name, Locations & Objects

- In a block-structured language with lexical scope, the same name can be associated with different locations, but only one of these locations can be accessed at any one time
- object: allocated location

Lifetime

- Lifetime (or extent) of an object is the duration of its allocation in the environment
- Lifetime of an object can extend beyond the region of a program in which it can be accessed

Lifetime

```
import random
passing = 70
def easy_grader(grade):
    …
def grader(grade):
    …
def main():
    …
```

Lifetime

```
def main():
    user_grade = float(input("Please enter a grade: "))
   # pick a random number between 0 and 9
    pick = random.randint(0,9)
    if pick == 0:
        letter_grade = easy_grader(user_grade)
    else:
        letter_grade = grader(user_grade)
    print ("Your final grade is ", letter_grade)
```

Lifetime

```
def easy_grader(grade):
    passing = 68
    if grade >= passing:
        return "P"
    else:
        return "F"

def grader(grade):
    if grade >= passing:
        return "P"
    else:
        return "F"
```

Lifetime

```
import random
passing = 70
def easy_grader(grade):
    …
def grader(grade):
    …
def main():
    …
```

Allocations
passing (70)

Lifetime

```
def main():
    user_grade = float(input("Please enter a grade: "))
   # pick a random number between 0 and 9
    pick = random.randint(0,9)
    if pick == 0:
        letter_grade = easy_grader(user_grade)
    else:
        letter_grade = grader(user_grade)
    print ("Your final grade is ", letter_grade)

if __name__ == '__main__':
   main()
```

Activation

activation record of main

Allocations
passing (70)
user_grade
pick
letter_grade

Lifetime

```
def main():
    user_grade = float(input("Please enter a grade: "))
   # pick a random number between 0 and 9
    pick = random.randint(0,9)
    if pick == 0:
        letter_grade = easy_grader(user_grade)
    else:
        letter_grade = grader(user_grade)
    print ("Your final grade is ", letter_grade)

if __name__ == '__main__':
    main()
```

Activation

activation record of main

Allocations
passing (70)
user_grade
pick
letter_grade

Lifetime

```
def easy_grader(grade):
    passing = 68
    if grade >= passing:
        return "P"
    else:
        return "F"
```

Allocations
passing (70)
user_grade
pick
letter_grade
passing (68)

activation record of main: user_grade, pick, letter_grade

activation record of easy_grader: passing (68)

Lifetime

```
def main():
    user_grade = float(input("Please enter a grade: "))
   # pick a random number between 0 and 9
    pick = random.randint(0,9)
    if pick == 0:
        letter_grade = easy_grader(user_grade)
    else:
        letter_grade = grader(user_grade)
    print ("Your final grade is ", letter_grade)
```

Allocations
passing (70)
user_grade
pick
letter_grade

Lexical Scope Revisited

```
passing = 70
def grader(grade):
    if grade >= passing:
        return "P"
    else:
        return "F"
def main():
    user_grade = float(input("Please enter a grade: "))
   # pick a random number between 0 and 9
    pick = random.randint(0,9)
    if pick == 0:
        passing = 68
    letter_grade = grader(user_grade)
    print ("Your final grade is ", letter_grade)
```

Will this implementation achieve the same result?

A. Yes
B. No

Lexical Scope Revisited

```
passing = 70
def grader(grade):
    if grade >= passing:
        return "P"
    else:
        return "F"
def main():
    user_grade = float(input("Please enter a grade: "))
    passing = 68
    letter_grade = grader(user_grade)
    print ("Your final grade is ", letter_grade)
```

What is the letter grade associated with 69?

A. P
B. F

Lexical Scope Revisited

```
passing = 70
def grader(grade):
    if grade >= passing:
        return "P"
    else:
        return "F"
def main():
    user_grade = float(input("Please enter a grade: "))
    passing = 68
    letter_grade = grader(user_grade)
    print ("Your final grade is ", letter_grade)
if __name__ == '__main__':
    main()
```

Allocations
passing (70)

Lexical Scope Revisited

```
passing = 70
def grader(grade):
    if grade >= passing:
        return "P"
    else:
        return "F"
def main():
    user_grade = float(input("Please enter a grade: "))
    passing = 68
    letter_grade = grader(user_grade)
    print ("Your final grade is ", letter_grade)
if __name__ == '__main__':
    main()
```

activation record of main

Allocations
passing (70)
user_grade
passing (68)
letter_grade

Lexical Scope Revisited

```
passing = 70
def grader(grade):
    if grade >= passing:
        return "P"
    else:
        return "F"
def main():
    user_grade = float(input("Please enter a grade: "))
    passing = 68
    letter_grade = grader(user_grade)
    print ("Your final grade is ", letter_grade)
if __name__ == '__main__':
    main()
```

Allocations	
passing (70)	
user_grade	activation record of main
passing (68)	activation record of main
letter_grade	activation record of main
grade(param)	activation record of grader

Lexical Scope Revisited

```
passing = 70
def grader(grade):
    if grade >= passing:
        return "P"
    else:
        return "F"
def main():
    user_grade = float(input("Please enter a grade: "))
    passing = 68
    letter_grade = grader(user_grade)
    print ("Your final grade is ", letter_grade)
if __name__ == '__main__':
    main()
```

Allocations
passing (70)
user_grade
passing (68)
letter_grade
grade(param)

activation record of main

activation record of grader

Dynamic Scope

- Dynamic scope: the bindings are determined during execution and they depend on the runtime context

Dynamic Scope?

```
passing = 70
def grader(grade):
    if grade >= passing:
        return "P"
    else:
        return "F"
def main():
    user_grade = float(input("Please enter a grade: "))
    passing = 68
    letter_grade = grader(user_grade)
    print ("Your final grade is ", letter_grade)
if __name__ == '__main__':
    main()
```

With dynamic scope, what would the letter grade corresponding to 69 be?

A. P
B. F

Dynamic Scope?

```
passing = 70
def grader(grade):
    if grade >= passing:
        return "P"
    else:
        return "F"
def main():
    user_grade = float(input("Please enter a grade: "))
    passing = 68
    letter_grade = grader(user_grade)
    print ("Your final grade is ", letter_grade)
if __name__ == '__main__':
    main()
```

Allocations
passing (70)
user_grade
passing (68)
letter_grade
grade(param)

activation record of main

activation record of grader

Dynamic vs Lexical Scope

- Which one is easier to implement?
  A. dynamic scoping
  B. lexical scoping

Dynamic vs Lexical Scope

- Which one is easier to debug?
  A. dynamic scoping
  B. lexical scoping