Introduction to Parsing
Lecture 5

webs

```
import os
CUR_PATH = os.getcwd()
IGNORE_SET = set(['__init__.py', "COUNT_SOURCELINES.PY'])
```

- Yeah, he's a parser-tongue.
- I didn't know Harry spoke Python.
## PA1 Summary

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FINAL AVERAGE</strong></td>
<td><strong>19.445989</strong></td>
</tr>
<tr>
<td>EXT</td>
<td><strong>0.700000</strong></td>
</tr>
<tr>
<td>MAX</td>
<td><strong>25.000000</strong></td>
</tr>
<tr>
<td>C</td>
<td><strong>15.000000</strong></td>
</tr>
<tr>
<td>PY</td>
<td><strong>55.000000</strong></td>
</tr>
<tr>
<td>RB</td>
<td><strong>9.000000</strong></td>
</tr>
<tr>
<td>JS</td>
<td><strong>11.000000</strong></td>
</tr>
<tr>
<td>CL</td>
<td><strong>57.000000</strong></td>
</tr>
<tr>
<td>HS</td>
<td><strong>2.000000</strong></td>
</tr>
<tr>
<td>ML</td>
<td><strong>11.000000</strong></td>
</tr>
</tbody>
</table>
Lexing Review

- Construct patterns using regular expressions that describe the accepted language of each token
- Use a Lexer Generator to convert patterns to a NFA to a table-based DFA
- Consume a single character at a time from input stream
- Emit a token when consumed characters are recognized by a particular token pattern
- You like lexical analysis.
Parsing

- Convert a stream of tokens to a parse tree
- The parse tree captures the syntactic correctness of input programs
  - Represents structure of the program

- You will love parsing.
```java
class Main {
    x : Int;
    y : Int;
    z : Int;
    main() : Object {
        if x < y then
            5
        else
            10
        fi
    }
};
```
class Main {
    x : Int;
    y : Int;
    z : Int;

    main() : Object {
        if x < y then
            5
        else
            10
        fi
    }
};
```java
class Main {
    x : Int;
    y : Int;
    z : Int;
    main() : Object {
        if x < y then
            5
        else
            10
        fi
    }
};
```
Pointed Parsing Preview

```java
class Main {
    x : Int;
    y : Int;
    z : Int;
    main() : Object {
        if x < y then
            5
        else
            10
        fi
    }
}
```
class Main {
    x : Int;
    y : Int;
    z : Int;
    main() : Object {
        if x < y then 5
        else 10
        fi
    }
};
class Main {
  x : Int;
  y : Int;
  z : Int;
  main() : Object {
    if x < y then
      5
    else
      10
    fi
  }
};
class Main {
    x : Int;
    y : Int;
    z : Int;
    main() : Object {
        if x < y then 5
        else 10
    fi
};
class Main {
    x : Int;
    y : Int;
    z : Int;
    main() : Object {
        if x < y then
            5
        else
            10
        fi
    }
}
class Main {
    x : Int ;
    y : Int ;
    z : Int ;
    main() : Object {
        if x < y then
            5
        else
            10
        fi
    }
};
while
Parsing Preview 2

```java
class Main {
    x : Int;
    y : Int;
    z : Int;
    main() : Object {
        while x < y loop
            if z = y + 1 then
                z <- z + y + 1
            else
                z
            pool
        }
    }
}
while condition

lt_expr

body_expr

class Main {
  x : Int;
  y : Int;
  z : Int;
  main() : Object {
    while x < y loop
      if z = y + 1 then
        z <- z + y + 1
      else
        z
    pool
  }
}
Parsing Preview 2

```
class Main {
    x : Int;
    y : Int;
    z : Int;
    main() : Object {
        while x < y loop
            if z = y + 1 then
                z <- z + y + 1
            else
                z
        end
    }
}
```
while

condition

lt_expr

ident:x

ident:y

body_expr

```java
class Main {
    x : Int;
    y : Int;
    z : Int;
    main() : Object {
        while x < y loop
            if z = y + 1 then
                z <- z + y + 1
            else
                z
        end loop
    }
    pool
};
```
while condition

lt_expr

ident:x
ident:y

body_expr

if

class Main {
  x : Int;
  y : Int;
  z : Int;
  main() : Object {
    while x < y loop
      if z = y + 1 then
        z <- z + y + 1
      else
        z
    pool
  }
}
class Main {
    x : Int ;
    y : Int ;
    z : Int ;
    main() : Object {
        while x < y loop
            if z = y + 1 then
                z <- z + y + 1
            else
                z
        pool
    then_expr
    else_expr
}

while

condition

lt_expr

ident:x

ident:y

if

if_cond

then_expr

else_expr
while

condition

lt_expr

ident:x
ident:y

if

if_cond

equality

then_expr

else_expr

class Main {
    x : Int;
    y : Int;
    z : Int;
    main() : Object {
        while x < y loop
            if z = y + 1 then
                z <- z + y + 1
            else
                z
        end
    }
}
class Main {
    x : Int ;
    y : Int ;
    z : Int ;
    main() : Object {
        while x < y loop
            if z = y + 1 then
                z <- z + y + 1
            else
                z
        }
    };

while condition

lt_expr

ident:x

ident:y

if cond

equality

ident:z

add

ident:y

1

then_expr

body_expr

if

else_expr
```java
class Main {
    x : Int;
y : Int;
z : Int;

    main() : Object {
        while x < y loop
            if z = y + 1 then
                z <- z + y + 1
            else
                pool
            fi
            z
        od
    };
}
```
class Main {
  x : Int;
  y : Int;
  z : Int;
  main() : Object {
    while x < y loop
      if z = y + 1 then
        z <- z + y + 1
      else
        z
    end
  }
}
class Main {
    x : Int ;
    y : Int ;
    z : Int ;
    main() : Object {
        while x < y loop
            if z = y + 1 then
                z <- z + y + 1
            else
                z
        end
    };
}

Parser Preview 2

while

condition

lt_expr

ident:x

ident:y

if_cond

equality

ident:z

add

ident:y

1

body_expr

if

then_expr

assign

ident:z

add

ident:z

add

else_expr

z
while

condition

lt_expr

ident:x

ident:y

if

if_cond

equality

ident:z

add

ident:y

1

then_expr

assign

ident:z

add

ident:z

add

ident:y

1

else_expr

pool

body_expr

class Main {
  x : Int ;
  y : Int ;
  z : Int ;
  main() : Object {
    while x < y loop
      if z = y + 1 then
        z <- z + y + 1
      else
        z
    pool
  }
}
class Main {
    x : Int;
    y : Int;
    z : Int;

    main() : Object {
      while x < y loop
        if z = y + 1 then
          z <- z + y + 1
        else
          z
      end
    };
  }

  body_expr

  while

  condition

  lt_expr

  ident:x

  ident:y

  if_cond

  equality

  ident:z

  add

  ident:y

  1

  then_expr

  assign

  ident:z

  add

  ident:z

  add

  ident:y

  1

  else_expr

  ident:z
Parsing overview

- Figure out what the **parse tree** (aka **abstract syntax tree**) looks like for a given input and language
- If a string is in the language (e.g., if it is a **syntactically valid program**), a parse tree **must** exist
How do we parse?

- Lexing uses regular languages to specify tokens
  - Then you use a Lexer Generator (flex, ply, ocamllex, ...)
- Parsing uses Context-free languages to specify grammars
  - Then you use a Parser Generator (bison, yacc, more ply, ...)

Compiler Construction
How do we parse?

- Lexing uses regular languages to specify tokens
  - Then you use a Lexer Generator (flex, ply, ocamllex, ...)
- Parsing uses Context-free languages to specify grammars
  - Then you use a Parser Generator (bison, yacc, more ply, ...)

- You are determining whether the stream of tokens represents a structurally valid input program.
The Role of the Parser

We need

1. A language to describe valid sequences of tokens (i.e., syntactically valid tokens)
2. A method (algorithm) for distinguishing valid from invalid sequences of tokens
Program Structure

- Programming languages have **recursive** structure.

- Consider the language of arithmetic expressions with integers, +, *, and ()

- An **expression** is either
  - an integer
  - another **expression** followed by “+” followed by another **expression**
  - another **expression** followed by “*” followed by another **expression**
  - a “(” followed by an **expression** followed by “)”
In our pretend programming language, an expression is either
- an integer
- another expression followed by “+” followed by another expression
- another expression followed by “*” followed by another expression
- a “(” followed by an expression followed by “)”

int, int+int, (int+int)*int are all expressions in this language
Notation for Programming Languages

- Alternatively:

- $E \rightarrow \text{int}$
- $E \rightarrow E + E$
- $E \rightarrow E \ast E$
- $E \rightarrow (E)$

- You can view these as **rewrite rules**

- You start with $E$ and replace occurrences of $E$ with some right-hand side

- $E \rightarrow E \ast E \rightarrow (E) \ast E \rightarrow (E + E) \ast E \rightarrow ... \rightarrow (\text{int} + \text{int}) \ast \text{int}$
Observation

- All arithmetic expressions can be obtained by a sequence of replacements
- Any sequence of replacements forms a valid arithmetic expression
- We cannot obtain \(( \text{int} )\) by any sequence of replacements! Why?
- This is a Context-free Grammar
Revenge of CS Theory

Thus, for any nondeterministic Turing machine $M$ that runs in some polynomial time $p(n)$, we can devise an algorithm that takes an input $\omega$ of length $n$ and produces $E_{M,\omega}$. The running time is $O(p^2(n))$ on a multitape deterministic Turing machine and...

WTF, MAN. I JUST WANTED TO LEARN HOW TO PROGRAM VIDEO GAMES.
Context-free Grammars

A context-free grammar consists of

- A set of non-terminals \( N \)
  - Written in uppercase throughout these notes
- A set of terminals \( T \) comprised of tokens
  - Lowercase or punctuation throughout these notes
- A start symbol \( S \) (a non-terminal)
- A set of productions (rewrite rules)

Assuming \( E \in N \)

\[
E \rightarrow \epsilon \quad \text{or} \quad E \rightarrow Y_1Y_2...Y_n \quad \text{where } Y_i \in N \cup T
\]
Example Context-free Grammar

- $E \rightarrow \text{int}$
- $E \rightarrow E + E$
- $E \rightarrow E \ast E$
- $E \rightarrow (E)$

Several terminals: int, *, +, (, )
Terminals are never replaced
One Non-terminal: $E$

Convention: First non-terminal replacement rule is the start rule
Context-free Grammars

- Context-free Grammars are more expressive than regular languages
- Every regular language can be described with a context-free grammar!
- Rules have the form $A \rightarrow aB, C \rightarrow \epsilon$ where $a \in T; A, B, C \in N$
NFA to Context-free Grammar

Can you convert the regular language $ab^*a$ to a CFG?

What are $T$, $N$, $S$, and relevant production rules?
Context-free Languages

- What if I wanted a language $A$ that counted balanced parentheses?

  $L(A) = (n)^n$
What if I wanted a language $A$ that counted balanced parentheses?

$L(A) = (n^n)

This language is not regular

- DFAs do not have a means of tracking occurrences in each state!
Context-free Languages

Read productions as replacement rules:

\[ X \rightarrow Y_1 \ldots Y_n \]

means that \( X \) can be replaced by \( Y_1 \ldots Y_n \)

\[ X \rightarrow \epsilon \]

means that \( X \) can be erased/eaten (replaced with empty string)
Context-free Grammars

- Grammars are generative
- Start with a string consisting of the start symbol, $S$
- Replace any non-terminal $X$ in the string by a right-hand side of some production $X \rightarrow Y_1 \ldots Y_n$
- Continue replacing until there are only terminals in the string
Context-free Grammars

More formally:

\[ X_1 \ldots X_{i-1} X_i X_{i+1} \ldots X_n \]

\[ \rightarrow \]

\[ X_1 \ldots X_{i-1} Y_1 \ldots Y_m X_{i+1} \ldots X_n \]

if there is a production

\[ X_i \rightarrow Y_i \ldots Y_m \]
Context-free Languages: $\rightarrow^*$

Write

$$X_1\ldots X_n \rightarrow^* Y_1\ldots Y_m$$

if

$$X_1\ldots X_n \rightarrow \ldots \rightarrow \ldots \rightarrow Y_1\ldots Y_m$$

in 0 or more steps
Context-free Languages

If $G$ is a context-free grammar with start symbol $S$, then the language of $G$ is

$$L(G) = \{ a_1...a_n \mid S \rightarrow^* a_1...a_n \}$$

where every $a_i$ is a terminal.

$L(G)$ is a set of strings over the alphabet of terminals.
Writing Grammars

When writing a grammar (or, indeed, a regular expression):

1. All strings generated are in the language
2. The grammar can generate all strings in the language
Implementation notes

- Context-free grammars are a big concept
- However
- Membership in a language is “yes”/“no”
  - But we need the parse tree (and eventually, the abstract syntax tree)
  - How do we decide which production rules to use?
- We must handle errors
- Need an implementation of grammars!
  - Bison, yacc, ply, etc.
Implementation notes (2)

- Form of the grammar is important
  - Many grammars generate the same language!

- Automatic tools are sensitive to specific grammars
  - Recall: Lexer generators sensitive to regexes!
A derivation is a sequence of productions $S \rightarrow ... \rightarrow ...$

A derivation can be drawn as a tree
- Start symbol is the tree’s root
- For a production $X \rightarrow Y_1...Y_n$, add children $Y_1,...,Y_n$ to node $X$
Derivation Example

Grammar:  \( E \rightarrow E + E | E \ast E | (E) | id \)
String:  id \* id + id
Can we build the derivation?
Derivation Example

1. $E$
2. $E + E$
3. $E * E + E$
4. id $* E + E$
5. id $* id + E$
6. id $* id + id$

Therefore, $E \rightarrow^* id * id + id$
Derivation in Detail

1. $E$
Derivation in Detail

1. $E$

2. $E + E$
Derivation in Detail

1. $E$
2. $E + E$
3. $E * E + E$

Diagram:

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```

```
  E
 /\  \\
E / \ E
```
Derivation in Detail

1. $E$
2. $E + E$
3. $E * E + E$
4. id * $E + E$
Derivation in Detail

1. $E$
2. $E + E$
3. $E * E + E$
4. $id * E + E$
5. $id * id + E$
Derivation in Detail

1. $E$
2. $E + E$
3. $E * E + E$
4. $id * E + E$
5. $id * id + E$
6. $id * id + id$
Notes on Derivation

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes

- A left-right traversal of the leaves is the original input

- The parse tree shows the association of operations, whereas the input string alone does not!
Left- and Right-most Derivations

- The example is a **left-most derivation**
  - At each step, replace the left-most non-terminal!

- The $\rightarrow^*$ operation does not specify which non-terminal is replaced!

- You can have **right-most derivations** too!

1. $E$
2. $E + E$
3. $E * E + E$
4. id * $E + E$
5. id * id + $E$
6. id * id + id
Right-most Derivation

1. $E$

$E + E$

$E + \text{id}$

$E * E + \text{id}$

$E * \text{id} + \text{id}$

$id * id + \text{id}$
Right-most Derivation

1. $E$
2. $E + E$
Right-most Derivation

1. $E$
2. $E + E$
3. $E + \text{id}$
Right-most Derivation

1. $E$
2. $E + E$
3. $E + \text{id}$
4. $E * E + \text{id}$
Right-most Derivation

1. $E$
2. $E + E$
3. $E + \text{id}$
4. $E \times E + \text{id}$
5. $E \times \text{id} + \text{id}$
Right-most Derivation

1. $E$
2. $E + E$
3. $E + \text{id}$
4. $E * E + \text{id}$
5. $E * \text{id} + \text{id}$
6. $\text{id} * \text{id} + \text{id}$
Summary of Derivations

- We don’t just want to know if \( s \in L(G) \)
  We also need the parse tree!
  - And eventually an abstract syntax tree
- A derivation defines a parse tree
  - but one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation
Review so far

A parser consumes a sequence of tokens, $s$, and produces a parse tree

- How do we **recognize** whether $s \in L(G)$?

- A **parse tree** of $s$ describes **how** $s \in L(G)$

- **Ambiguity** arises when more than one parse tree (interpretation) for some string $s$

- An **error** occurs when no parse tree exists for some string $s$

- How do we actually **construct** the parse tree?
Ambiguity

Some grammars admit ambiguity
Consider:
$$E \rightarrow E + E | E \ast E | (E) | \text{int}$$

Some strings in $L(G)$
\begin{align*}
\text{int} + \text{int} + \text{int} \\
\text{int} \ast \text{int} + \text{int}
\end{align*}
Ambiguity Example

The string \texttt{int + int + int} has two parse trees

![Parse Trees]

Consider: + is left-associative!
Ambiguity Example (2)

The string \texttt{int * int + int} has two parse trees

Consider: PEMDAS! * before +
Ambiguity

- A grammar is **ambiguous** if it has more than one parse tree for some string
  - Equivalently, there is more than one right-most or left-most derivation for some string

- Ambiguity is **bad**
  - Leaves the meaning of some programs ill-defined

- Ambiguity is **common** in programming languages
  - Arithmetic expressions
  - If-then-else expressions
Dealing with Ambiguity

Most direct method: rewrite the grammar unambiguously

\[
E \rightarrow E + T | T \\
T \rightarrow T \ast \text{int} | \text{int}(E)
\]

- Enforces precedence of \(\ast\) over +
- Enforces left-associativity of + and *
Dealing with Ambiguity

The `int * int + int` string has only one parse tree now.

Diagram: 
- The left tree represents a correct parse tree.
- The right tree represents an incorrect parse tree, marked with an X.
The Dangling Else

Consider this new grammar

\[ E \rightarrow \text{if } E \text{ then } E \]
\[ \quad | \text{if } E \text{ then } E \text{ else } E \]
\[ \quad | \text{other...} \]

This grammar is ambiguous
Dangling Else Example

The string \( \text{if } E_1 \text{ then if } E_2 \text{ then } E_3 \text{ else } E_4 \) has two parse trees.

Usually, we want the second one.
else matches the closest unmatches then. Can we describe that in the grammar?

Distinguish between matched and unmatched “then”

\[ \begin{align*}
E & \rightarrow MIF \\
& \mid UIF \\
MIF & \rightarrow \text{if } E \text{ then } MIF \text{ else } MIF \\
& \mid \text{other...} \\
UIF & \rightarrow \text{if } E \text{ then } E \\
& \mid \text{if } E \text{ then } MIF \text{ else } UIF
\end{align*} \]
Dangling Else Example

The string \( \text{if } E_1 \text{ then if } E_2 \text{ then } E_3 \text{ else } E_4 \) is no longer parsed ambiguously!
Ambiguity

- No general approach exists for handling ambiguity
- Impossible to automatically convert every ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
  - MIF/UIF is not necessarily elegant or natural
  - We would like disambiguation mechanisms
Precedence and Disambiguation

- Instead of rewriting the grammar...
  - Use the ambiguous one
  - But also include disambiguating declarations

Most tools allow precedence and associativity declarations to disambiguate grammars
Associativity Declaration

The `int * int + int` string has only one parse tree now

Left-associativity declaration: `%left +`  
 `%left *`
Review

- We specify the **syntax** of the language using context-free grammars
- We use a **parser generator** to construct a parser from our grammar
- A parser will tell us whether a **string of tokens** $\in L(G)$
  - Also builds a **parse tree**
  - Passes it on to the rest of the compiler

- Next time: How do we actually build the parse tree?
Midterm 1 in Two Weeks

Things to know

- Cool Syntax
- Compiler/language processor steps
- Regular Languages
- NFA/DFAs
- Regular Expressions
- Parsing
- Context-Free Grammars