The Structure of Programming Languages

- With the exception of the Generator we saw that all language processors perform some kind of syntax analysis – an analysis of the structure of the program.

- To make this efficient and effective we need some mechanism to specify the structure of a programming language in a straightforward manner.

- We use grammars for this purpose.
Grammars

- The most convenient way to describe the structure of programming languages is using a context-free grammar (often called CFG or BNF for Backus-Nauer Form).
- Here we will simply refer to grammars with the understanding that we are referring to CFGs. (there are many kind of other grammars: regular grammars, context-sensitive grammars, etc)
Grammars

- Grammars can readily express the structure of phrases in programming languages
  - stmt: function-def | return-stmt | if-stmt | while-stmt
  - function-def: **function** name expr stmt
  - return-stmt: **return** expr
  - if-stmt: **if** expr **then** stmt **else** stmt **endif**
  - while-stmt: **while** expr **do** stmt **endo**
Grammars have 4 parts to them

1. Non-terminal Symbols - these give names to phrase structures - e.g. function-def

2. Terminal Symbols - these give names to the tokens in a language – e.g. `while` (sometimes we don’t use explicit tokens but put the words that make up the tokens of a language in quotes)

3. Rules - these describe that actual structure of phrases in a language – e.g. `return-stmt: return exp`

4. Start Symbol - a special non-terminal that gives a name to the largest possible phrase(s) in the language (often denoted by an asterisk)

   - In our case that would probably be the stmt non-terminal
## Example: The Exp00 Language

| prog : stmt prog |
| @"" |
| stmt : p exp ; |
| @s var exp ; |
| exp : + exp exp |
| @- exp exp |
| @ ( exp ) |
| @ var |
| @ num |
| var : x | y | z |
| num : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Example Exp00 Program:

```
s x 1 ; p + x 1 ;
```
Grammars

- A grammar tells us if a sentence belongs to the language,
  - e.g. Does ‘s x 3 ;’ belong to the language?
- We can show that a sentence belongs to the language by constructing a parse tree starting at the start symbol
Grammars

\[ s \times 3 ; \]

\[
\begin{align*}
\text{prog} & : \text{stmt \ prog} \\
& \mid "" \\
\text{stmt} & : p \ \text{exp} ; \\
& \mid s \ \text{var} \ \text{exp} ; \\
\text{exp} & : + \ \text{exp} \ \text{exp} \\
& \mid - \ \text{exp} \ \text{exp} \\
& \mid ( \ \text{exp} ) \\
& \mid \text{var} \\
& \mid \text{num} \\
\text{var} & : x \mid y \mid z \\
\text{num} & : 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

Note: constructing the parse tree by filling in the leftmost non-terminal at each step we obtain the **left-most derivation**:

\[
\begin{align*}
\text{prog} \Rightarrow \\
\text{stmt \ prog} \Rightarrow \\
\text{stmt} \ \text{var} \ \text{exp} ; \ \text{prog} \Rightarrow \\
s \ \text{var} \ \text{exp} ; \ \text{prog} \Rightarrow \\
s \ x \ \text{exp} ; \ \text{prog} \Rightarrow \\
s \ x \ 3 ; \ \text{prog} \Rightarrow \\
s \ x \ 3 ;
\end{align*}
\]

Constructing the parse tree by filling in the rightmost non-terminal at each step we obtain the **right-most derivation**.
Grammars

- Every **valid** sentence (a sentence that belongs to the language) has a parse tree.
- Test if these sentences are valid:
  - `p x + 1 ;`
  - `s x 1 ; s y x ;`
  - `s x 1 ; p (+ x 1) ;`
  - `s y + 3 x ;`
  - `s + y 3 x ;`
The converse is also true:

- If a sentence has a parse tree, then it belongs to the language.
- This is precisely what parsers do: to show a program is syntactically correct, parsers construct a parse tree.
Top-Down Parsers - LL(1)

- LL(1) parsers start constructing the parse tree at the start symbol as opposed to bottom up parsers, LR
- LL(1) parsers use the current position in the input stream and a single look-ahead token to decide how to construct the next node(s) in the parse tree.
- LL(1)
  - Reads input from Left to right.
  - Constructs the Leftmost derivation
  - Uses 1 look-ahead token.
Top-Down Parsing

Lookahead Set

Consider: p + x 1 ;

For top-down parsing we can think of the grammar extended with the one token look-ahead set.

The look-ahead set uniquely identifies the selection of each rule within a block of rules.
Computing the Lookahead Set

```python
def compute_lookahead_sets(G):
    
    Accepts: G is a context-free grammar viewed as a list of rules
    Returns: GL is a context-free grammar extended with lookahead sets
    
    GL = []
    for R in G:
        (A, rule_body) = R
        S = first_symbol(rule_body)
        if S == ":
            GL.append((A, set(['']), rule_body))
        elif S in terminal_set(G):
            GL.append((A, set(S), rule_body))
        elif S in non_terminal_set(G):
            L = lookahead_set(S,G)
            GL.append((A, L, rule_body))
    return GL
```

Note: a grammar is a list of rules and a rule is the tuple (non-terminal, body)
Note: a grammar extended with lookahead sets is a list of rules where each rule
is the tuple (non-terminal, lookahead-set, body)
Computing the Lookahead Set

```python
def lookahead_set(N, G):
    
    #
    # Accepts: N is a non-terminal in G
    # Accepts: G is a context-free grammar
    # Returns: L is a lookahead set
    
    L = set()
    for R in G:
        (A, rule_body) = R
        if A == N:
            Q = first_symbol(rule_body)
            if Q == "":
                raise ValueError("non-terminal {} is a nullable prefix".format(A))
            elif Q in terminal_set(G):
                L = L | set(Q)
            elif Q in non_terminal_set(G):
                L = L | lookahead_set(Q, G)
    return L
```

Computing the Lookahead Set

grammar G:

prog : stmt prog
    | ""
stmt : p exp ;
    | s var exp ;
exp : + exp exp
    | - exp exp
    | ( exp )
    | var
    | num
var : x | y | z
num : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

grammar GL:

prog : {p,s} stmt prog
    | {""} ""
stmt : {p} p exp ;
    | {s} s var exp ;
exp : {+} + exp exp
    | {-} - exp exp
    | {()} ( exp )
    | {x,y,z} var
    | {0,1,2,3,4,5,6,7,8,9} num
var : {x} x | {y} y | {z} z
num : {0} 0 | {1} 1 | {2} 2 | ... | {8} 8 | {9} 9
Actually, the algorithm we have outlined computes the lookahead set for a simpler parsing technique called sLL(1) – simplified LL (1) parsing.

sLL(1) parsing does not deal with non-terminals that expand into the empty string in the first position of a production – also called nullable prefixes.

All our hand-built parsers will be sLL(1) but when we use Ply and we will have access to a powerful parsing technique called LR(1).
Constructing a Parser

- A sLL(1) parser can be constructed by hand by *converting each non-terminal into a function*

- The body of the function *implements the right sides of the rules for each non-terminal* in order to:
  - Process terminals
  - Call the functions of other non-termsinals as appropriate
A parser for Exp0

We start with the grammar for Exp0 extended with the lookahead sets

```
prog : {p,s} stmt prog
      | ""

stmt : {p} p exp ;
     | {s} s var exp ;

exp : {+} + exp exp
    | {-} - exp exp
    | {} ( exp )
    | {x,y,z} var
    | {0,1,2,3,4,5,6,7,8,9} num

var : {x} x | {y} y | {z} z

num : {0} 0 | {1} 1 | {2} 2 | ... | {8} 8 | {9} 9
```
Constructing a Parser by Hand

We need to set up some sort of character input stream

```python
from grammar_stuff import InputStream

InputStream supports the operations: ‘pointer’, ‘next’, and ‘end_of_file’

set_stream(InputStream([[<input list of characters>]]))
```

**Note:** all the Python code given in the slides is available in the ‘code’ section of the Plipy Notebooks.

**Note:** the hand-built parser for Exp0 is in 'exp0_recdesc.py'
Constructing a Parser by Hand

Consider the following rule:

```
prog : stmt prog
| ""
```

```
def prog():
    while not I.end_of_file():
        stmt()
```

Note: a lookahead set is not necessary here – only one rule to choose from besides the empty rule.
Constructing a Parser by Hand

stmt : {'p'} 'p' exp ';' |
      | {'s'} 's' var exp ';

def stmt():
    sym = I.pointer()
    if sym == 'p':
        I.next()
        exp()
        I.next() # match the ';
    elif sym == 's':
        I.next()
        var()
        exp()
        I.next() # match the ';
    else:
        raise SyntaxError('unexpected symbol {} while parsing'.format(sym))

Notice that we are using the look-ahead set to decide which rule to call!
Constructing a Parser by Hand

```python
def exp():
    sym = I(pointer())
    if sym == '+':
        I.next()
        exp()
        exp()
    elif sym == '-':
        I.next()
        exp()
        exp()
    elif sym in ['x', 'y', 'z']:
        var()
    elif sym in ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']:
        num()
    else:
        raise SyntaxError('unexpected symbol {} while parsing'.format(sym))
```

```
exp : {'+'} 'exp exp
| {'-'} '-' exp exp
| {'('} '(' exp ')' 
| {'x','y','z'} var 
| {'0'...'9'} num
```
Constructing a Parser by Hand

```python
def var():
    sym = I.pointer()
    if sym == 'x':
        I.next()
    elif sym == 'y':
        I.next()
    elif sym == 'z':
        I.next()
    else:
        raise SyntaxError('unexpected symbol {} while parsing'.format(sym))
```

\[
\text{var} : \{ \text{'x'} \} \text{'x'} | \{ \text{'y'} \} \text{'y'} | \{ \text{'z'} \} \text{'z'}
\]
Constructing a Parser

```
def num():
    sym = I.pointer()
    if sym in ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']:
        I.next()
    else:
        raise SyntaxError('unexpected symbol {} while parsing'.format(sym))
```
Constructing a Parser: An Example

def prog():
    while not I.end_of_file():
        stmt()

def stmt():
    sym = I.pointer()
    if sym == 'p':
        I.next()
        expr()
        I.next() # ';'
    elif sym == 's':
        I.next()
        var()
        expr()
        I.next() # ';'
    else:
        raise SyntaxError(…)

def exp():
    sym = I.pointer()
    if sym == '+':
        I.next()
        expr()
        expr()
    elif sym == '-':
        I.next()
        expr()
        expr()
    elif sym in ['x', 'y', 'z']:
        var()
    elif sym in ['0', ... , '9']:
        num()
    else:
        raise SyntaxError(…)

Call Tree:

```
prog()  
   /   
stmt()  
   /     
exp()   
   /     
   +     
   /     
exp     exp
   /     
exp     exp
   /     
exp     exp
   /     
rhsvar num
   /     
x     1
```

```p + x 1;```
Constructing a Parser: An Example

- Observations:
  - Our parser is an LL(1) parser (why?)
  - The parse tree is implicit in the function call activation record stack
  - Building a parser by hand is a lot of work and the parser is difficult to maintain.
  - We would like a tool that reads our grammar file and converts it automatically into a parser – that is what Ply does!
Running the Parser

- The examples assume that you have cloned/downloaded the Plipy book and have access to the ‘code’ folder.
- For notebook demos it is assumed that you navigated Jupyter to the ‘code’ folder and started a new notebook.
- This works for all OS’s that Anaconda supports.
Running the Parser

In [1]: from exp0_recdesc import prog

In [2]: from exp0_recdesc import set_stream

In [3]: from grammar_stuff import InputStream

In [4]: set_stream(InputStream(['s','x','l','p','x','x']))

In [5]: prog()

In [6]: set_stream(InputStream(['s','x','l','q','x','x']))

In [7]: prog()

File "<string>", line unknown
SyntaxError: unexpected symbol q while parsing
Assignments

- Read Chapter 2
- Assignment #1 -- see the website