# 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 straight forward 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 **enddo**

# Grammars

- 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 Exp0 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 Exp0 Program:

s x 1 ; p + x 1 ;

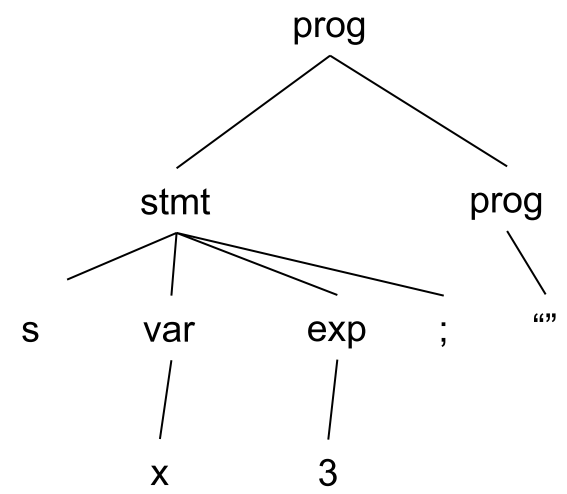
Start Symbol: prog

# 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 x 3 ;




```
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
```

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

prog $\Rightarrow$
stmt prog $\Rightarrow$
s var exp ; prog $\Rightarrow$
s x exp ; prog $\Rightarrow$
s x 3 ; prog $\Rightarrow$
s x 3 ;

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 ;

# Parsers

- The converse is also true:
  - If a sentence has a parse tree, then it belongs to the language.
  - This is precisely what <u>parsers</u> do: to show a program is <u>syntactically correct,</u> parsers construct a <u>parse tree</u>

# 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 ;

```
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 | {3} 3 | {4} 4 | {5} 5 | {6} 6 | {7} 7 | {8} 8 | {9} 9
```

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

```
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

```
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
```

set union operator in Python

# 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
```

# Computing the Lookahead Set

- 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-terminals as appropriate

# Constructing a Parser by Hand

- 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

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

```
def prog():
    sym = I.pointer()
    if sym in ['p','s']:
        stmt()
        prog()
    elif sym == "":
        pass
    else:
        raise SyntaxError('unexpected symbol {} while parsing'.format(sym))
```

**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.match(';')
    elif sym == 's':
        I.next()
        var()
        exp()
        I.match(';')
    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

exp : {‘+’} '+' exp exp
| {‘-’} '-' exp exp
| {‘(‘} '(' exp ')'
| {‘x’,’y’,’z} var
| {‘0’…’9’} num

```
def exp():
    sym = I.pointer()
    if sym == '+':
        I.next()
        exp()
        exp()
    elif sym == '-':
        I.next()
        exp()
        exp()
    elif sym == '(':
        I.next()
        exp()
        I.match(')')
    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))
```

# Constructing a Parser by Hand

var : { ‘x’ } ‘x’ | { ‘y’ } ‘y’ | { ‘z’ } 'z’

```
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))
```

# Constructing a Parser

num : { ‘0’ } ‘0’ | { ‘1’ } ‘1’ | … | { ‘9’ } '9'

```
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():
    sym = I.pointer()
    if sym in ['p','s']:
        stmt()
        prog()
    elif sym == "":
        pass
    else:
        raise SyntaxError(...)
)
```

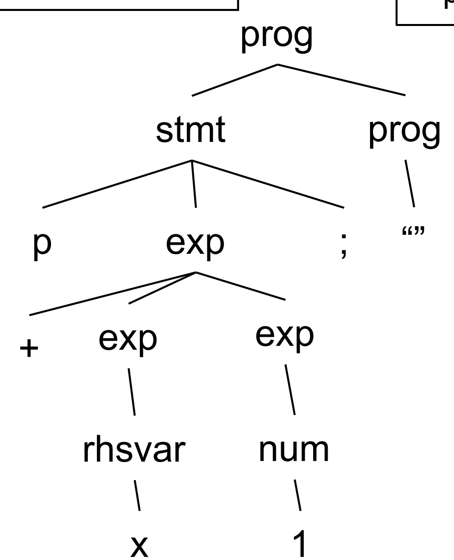
```
def stmt():
    sym = I.pointer()
    if sym == 'p':
        I.next()
        exp()
        I.match(';')
    elif sym == 's':
        I.next()
        var()
        exp()
        I.match(';')
    else:
        raise SyntaxError(...)
```

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

p + x 1 ;

Call Tree:

```
prog()
  stmt()
    I.next() #'p'
    exp()
      I.next() #'+'
      exp()
        var()
          I.next() #'x'
      exp()
        num()
          I.next() #'1'
    I.match(';')
  prog()
```



# 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','1',';','p','x',';']))

In [5]: prog()

In [6]: set_stream(InputStream(['s','x','1',';','q','x',';']))

In [7]: prog()

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

In [ ]:
```

# Assignments

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