The Structure of Programming Languages

- 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
- Grammars allow us to derive valid sentences or programs that are part of the language by applying the rules of the grammar repeatedly until no further rule application is possible.

Listing 2.1: A grammar that specifies the syntactic structure of arithmetic expressions.

```
program : expression

expression : expression + expression
           | expression - expression
           | expression \*  expression
           | expression / expression
           | \( expression \)
           | x
           | y
           | z
```

```
program                    # apply program : expression
⇒ expression               # apply expression : expression + expression
⇒ expression + expression  # apply expression : x
⇒ x + expression           # apply expression : y
⇒ x + y
```

Listing 2.1: A grammar that specifies the syntactic structure of arithmetic expressions.

```
program : expression

expression : expression + expression
           | expression - expression
           | expression \*  expression
           | expression / expression
           | \( expression \)
           | x
           | y
           | z
```

x(y) + z

program =>
Expression =>
Expression + expression =>
(expression) + expression =>
(y) + expression => syntax error

Grammars

- Grammars have 4 parts to them
  1. Non-terminal Symbols - these give names to phrase structures - e.g. program
  2. Terminal Symbols - these give names to the tokens in a language – e.g. x
  3. Rules - these describe that actual structure of phrases in a language – e.g. expression : expression + expression
  4. Start Symbol - a special non-terminal that gives a name to the largest possible phrase(s) in the language
     - By convention it is usually the non-terminal defined by the first rule.
     - In our case that would be the program non-terminal

Derivations

A derivation is a sequence of steps that begins with the start symbol and at each derivation step replaces a single non-terminal with the right side of a production that has that non-terminal on the left side. A valid sentence in the language of a grammar is a sequence of symbols arrived at through a derivation that contains only terminals.

Let’s try this with: `x + y * z`

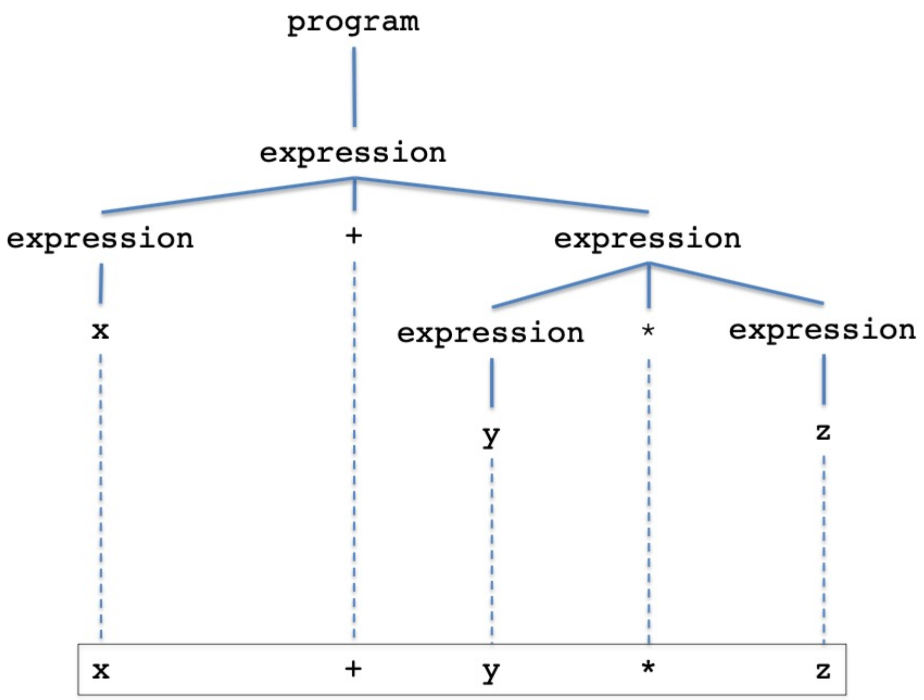
```
program
⇒ expression
⇒ expression + expression
⇒ x + expression
⇒ x + expression * expression
⇒ x + y * expression
⇒ x + y * z
```

Since we were able to derive our sentence from the start symbol our sentence is valid!

Parse Trees

- Derivations can also be expressed as parse trees.

```
program
⇒ expression
⇒ expression + expression
⇒ x + expression
⇒ x + expression * expression
⇒ x + y * expression
⇒ x + y * z
```



Example: The Exp0 Language

```
stmt_list : stmt stmt_list
          | ""

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 ;

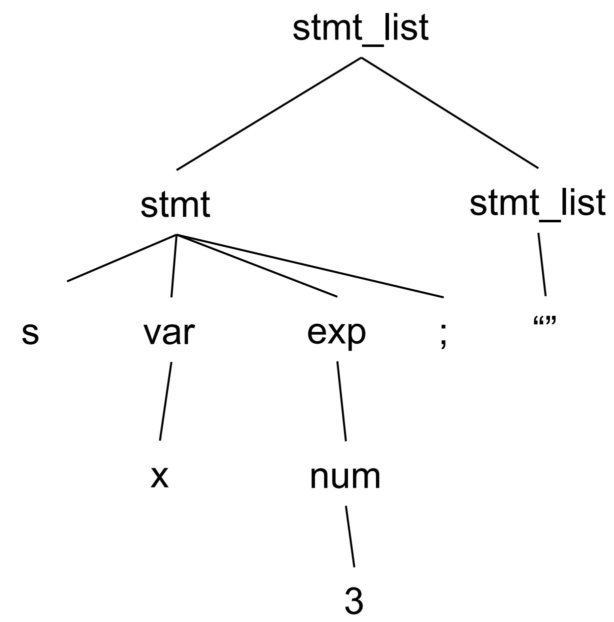
Start Symbol: stmt_list

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 ;




```
stmt_list : stmt stmt_list
          | ""

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**:

stmt_list ⇒
stmt stmt_list ⇒
s var exp ; stmt_list ⇒
s x exp ; stmt_list ⇒
s x num ; stmt_list ⇒
s x 3 ; stmt_list ⇒
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 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 ;

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

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:

```
stmt_list : stmt stmt_list
        | ""

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:

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

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 parsers will be sLL(1)
  - Later in the course we will discuss a tool called Ply and we will have access to another parsing technique called LR(1) – which is bottom-up parsing

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 LL(1) Parsers

- A parser for Exp0
  - We start with the grammar for Exp0 extended with the lookahead sets

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

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 LL(1) Parsers

We need to set up some sort of character input stream. In our case we use the ‘InputStream’ class

Note: all the Python code given in the slides is available in the repl.it VM.

Note: the parser for Exp0 is in ’exp0’

The Stream Class

It is convenient to map the input string into a stream structure.

```
class InputStream:
    def __init__(self, char_stream=None):
        # if no stream given read it from the terminal
        if not char_stream:
            char_stream = stdin.read()
        # turn char stream into a list of characters
        # ignoring any kind of white space
        clean_stream = char_stream.replace(' ','') \
                                  .replace('\t','') \
                                  .replace('\n','')
        self.stream = [c for c in clean_stream]
        self.stream.append('\eof')
        self.stream_ix = 0

    def pointer(self):
        return self.stream[self.stream_ix]

    def next(self):
        if not self.end_of_file():
            self.stream_ix += 1
        return self.pointer()

    def match(self, sym):
        if sym == self.pointer():
            s = self.pointer()
            self.next()
            return s
        else:
            raise SyntaxError('unexpected symbol {} while parsing, expected {}'
                              .format(self.stream[self.stream_ix], sym))

    def end_of_file(self):
        if self.pointer() == '\eof':
            return True
        else:
            return False
```

Constructing LL(1) Parsers

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

```
def stmt(stream):
    sym = stream.pointer()
    if sym in ['p']:
        stream.match('p')
        exp(stream)
        stream.match(';')
        return
    elif sym in ['s']:
        stream.match('s')
        var(stream)
        exp(stream)
        stream.match(';')
        return
    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 LL(1) Parsers

Consider the following rule:

stmt_list : {p,s} stmt stmt_list
| {""} ""

```
def stmtlist(stream):
    sym = stream.pointer()
    if sym in ['p','s']:
        stmt(stream)
        stmtlist(stream)
        return
    else:
        return
```

Constructing LL(1) Parsers

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

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

Constructing LL(1) Parsers

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

```
def var(stream):
    sym = stream.pointer()
    if sym in ['x']:
        stream.match('x')
        return
    elif sym in ['y']:
        stream.match('y')
        return
    elif sym in ['z']:
        stream.match('z')
        return
    else:
        raise SyntaxError('unexpected symbol {} while parsing'.format(sym))
```

Constructing LL(1) Parsers

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

```
def num(stream):
    sym = stream.pointer()
    if sym in ['0']:
        stream.match('0')
        return
    elif sym in ['1']:
        stream.match('1')
        return
    elif sym in ['2']:
        stream.match('2')
        return
    elif sym in ['3']:
        stream.match('3')
        return
    elif sym in ['4']:
        stream.match('4')
        return
    elif sym in ['5']:
        stream.match('5')
        return
    elif sym in ['6']:
        stream.match('6')
        return
    elif sym in ['7']:
        stream.match('7')
        return
    elif sym in ['8']:
        stream.match('8')
        return
    elif sym in ['9']:
        stream.match('9')
        return
    else:
        raise SyntaxError('unexpected symbol {} while parsing'.format(sym))
```

Constructing LL(1) Parsers

- To pull this all together we add a high-level parsing function

```python
def parse():
    from inputstream import InputStream
    stream = InputStream() # reads from stdin
    try:
        stmtlist(stream) # call the parser function for start symbol
        if stream.end_of_file():
            print("parse successful")
        else:
            raise SyntaxError("bad syntax at {}".format(stream.pointer()))
    except Exception as e:
        print("error: " + str(e))

if __name__ == "__main__":
    parse()
```

Running the Parser

- Run the parser in a command shell, in our case we use the cloud based Linux VM

```
$ python3 exp0_parser.py
s x 1; p (+ x 1);
^D
parse successful
$
```

End of input

Parsers build Parse Trees

- In order to see that parsers build parse trees in order to prove that a sentence belongs to a language consider the expression: + x y

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

```
def var(stream):
    sym = stream.pointer()
    if sym in ['x']:
        stream.match('x')
        return
    elif sym in ['y']:
        stream.match('y')
        return
    elif sym in ['z']:
        stream.match('z')
        return
    else:
        raise SyntaxError('unexpected symbol {} while parsing'.format(sym))
```

Parsing + x y will result in the following tree:

```
exp
  match(+)
  exp
    var
      match(x)
  exp
    var
      match(y)
```

Parsing function call tree == parse tree

Our First Language Processor

- Parsers are good because they can tell us if a program is valid or not
- But in order to have to extend it with “actions”, code that does something useful.
- Idea: Our first language processor parses Exp0 programs and counts the number of times the value of a variable is accessed
  - Example: s x 1; s x (+ x 1);
  - In this program we only access the value of a variable once!
- Note: Scanning for variable names and counting the number of times a variable name occurs does NOT work, we need to use a parser that understands the difference between a variable value reference and a variable storage reference (rvalues and lvalues).

Extended Parser

```
# counter for variable expression references
count = None

def parse():
    global count
    count = 0
    from inputstream import InputStream
    stream = InputStream() # reads from stdin
    try:
        stmtlist(stream) # call the parser function for start symbol
        if stream.end_of_file():
            print("found {} variable reference{}"
                  .format(count,"" if count==1 else "s"))
        else:
            raise SyntaxError("bad syntax at {}".format(stream.pointer()))
    except Exception as e:
        print("error: " + str(e))
```

```
def exp(stream):
    sym = stream.pointer()
    if sym in ['+']:
...
    elif sym in ['x', 'y', 'z']:
        global count # make sure we reference the global count
        var(stream)  # recognize an rvalue var
        count += 1   # bump up the counter
        return
...
    else:
        raise SyntaxError('unexpected symbol {} while parsing'
                 .format(sym))
```

Running the Processor

```
$ python3 exp0count.py
s x 1; p (+ x 1);
^D
found 1 variable reference
```

Assignments

- Read Chapter 2
- Assignment #1 -- see BrightSpace