Bottom-Up Parsing – LR(1)

- Previously we have studied top-down or LL(1) parsing.
- The idea here was to start with the start symbol and keep expanding it until the whole input was read and matched.
- In bottom-up or LR(1) parsing we do exactly the opposite, we try to match the input to a rule and then keep *reducing* the input replacing it with the non-terminal of the rule. The last step is to replace the current input with the start-symbol.
- **Observation:** in LR(1) parsing we apply the rules backwards – this is called *reduction*

Bottom-Up Parsing – LR(1)

- In our LL(1) parsing example we replaced non-terminal symbols with functions that did the expansions and the matching for us.
- In LR(1) parsing we use a stack to help us find the correct reductions.
- Given a stack, an LR(1) parser has four available actions:
  - **Shift** – push an input token on the stack
  - **Reduce** – pop elements from the stack and replace by a non-terminal (apply a rule ‘backwards’)
  - **Accept** – accept the current program
  - **Reject** – reject the current program

Bottom-Up Parsing – LR(1)

p + x 1 ;

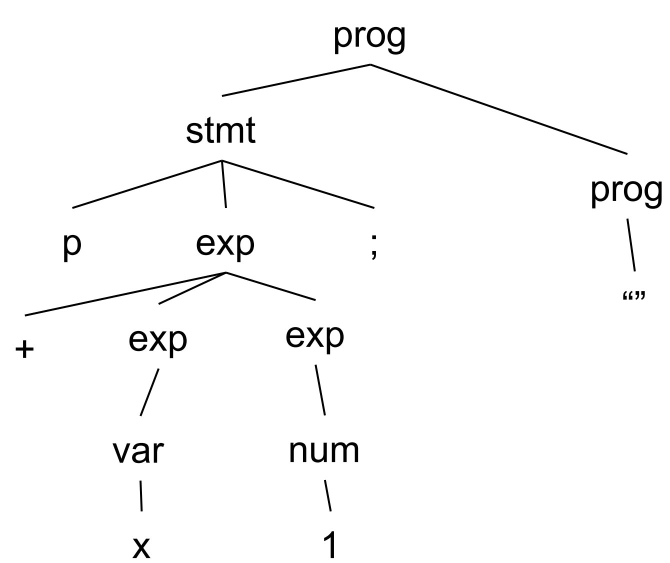
```
grammar exp0;

prog        :        stmt prog
            |        ""
            ;

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

exp         :        '+' exp exp
            |        '-' exp exp
            |        '(' exp ')'
            |        var
            |        num
            ;

var         :        'x' | 'y' | 'z'
            ;

num         :        '0' …'9'
            ;
```

Stack	Input	Action
<empty>	p + x 1 ;	Shift
p	+ x 1 ;	Shift
p +	x 1 ;	Shift
p + x	1 ;	Reduce
p + var	1 ;	Reduce
p + exp	1 ;	Shift
p + exp 1	;	Reduce
p + exp num	;	Reduce
p + exp exp	;	Reduce
p exp	;	Shift
p exp ;	<empty>	Reduce
stmt	<empty>	Shift
stmt <empty>	<empty>	Reduce
stmt prog	<empty>	Reduce
prog	<empty>	Accept

Bottom-Up Parsing – LR(1)

Stack
<empty>
p
p +
p + x
p + var
p + exp
p + exp 1
p + exp num
p + exp exp
p exp
p exp ;
stmt
stmt <empty>
stmt prog
prog

p + x 1 ;



Bottom-Up Parsing – LR(1)

Let’s try an illegal sentence

p + x s ;

```
grammar exp0;

prog    :   stmt prog
        |   ””
        ;

stmt    :   'p' exp ';'
        |   's’ var exp ';'
        ;

exp     :   '+' exp exp
        |   '-' exp exp
        |   '(' exp ')'
        |   var
        |   num
        ;

var     :   'x' | 'y' | 'z'
        ;

num     :   '0’ …'9'
        ;
```

Stack	Input	Action
<empty>	p + x s ;	Shift
p	+ x s ;	Shift
p +	x s ;	Shift
p + x	s ;	Reduce
p + var	s ;	Reduce
p + exp	s ;	Shift
p + exp s	;	Shift
p + exp s ;	<empty>	Reject

Bottom-Up Parsing – LR(1)

Let's try it with the a grammar where left-hand side and right-hand variables are differentiated.

p + x 1 ;

```
prog    :   stmt prog
        |   ""
        ;

stmt    :   'p' exp ';'
        |   's' lhsvar exp ';'
        ;

exp     :   '+' exp exp
        |   '-' exp exp
        |   '(' exp ')'
        |   rhsvar
        |   num
        ;

lhsvar  :   'x' | 'y' | 'z'
        ;

rhsvar  :   'x' | 'y' | 'z'
        ;

num     :   '0' ...'9'
        ;
```

Stack	Input	Action
<empty>	p + x 1 ;	Shift
p	+ x 1 ;	Shift
p +	x 1 ;	Shift
p + x	1 ;	Reject

There is a conflict between the lhsvar rule and rhsvar rule here, we do not have enough information to select one rule over the other. This is called a **reduce/reduce conflict** in bottom-up parsing terminology.

That means, even though our grammar is a perfectly legal context-free grammar, it is not a grammar that can be used by a bottom-up parser, we say that the **grammar is not LR(1)**.

We didn't point this out but there are also grammars which are perfectly legal CFG's that are not LL(1).

Bottom-Up Parsing – LR(1)

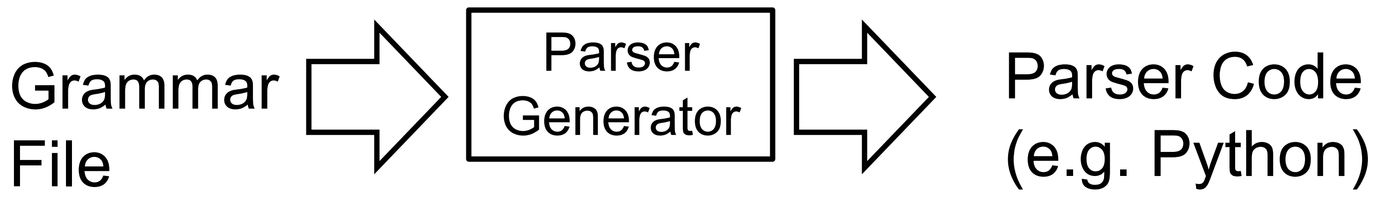
- LR(1) parsers are implemented in such tools as Yacc (Unix) and Bison (Linux)
- The tool we will be using, Ply, also implements LR(1) parsing.
- Other tools such as ANTLR implement LL(1) parsing*

* Actually ANTLR implement LL(k) parsing a slightly more powerful version of LL(1) parsing.

Parser Generators

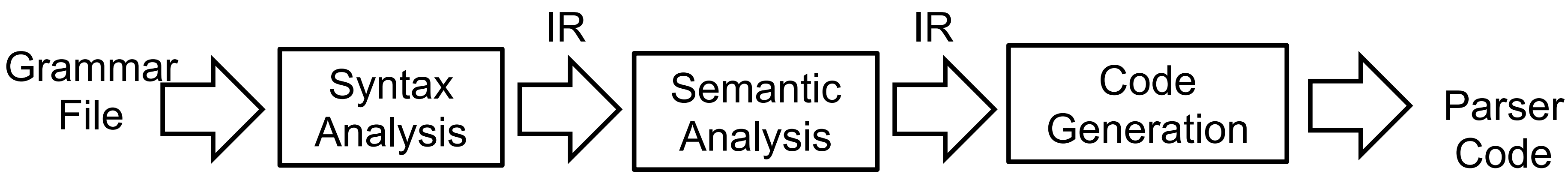
- Writing parsers by hand if difficult and time consuming
- The resulting parsers are difficult to maintain and extend
- Ideally we would like a tool that reads a grammar definition and generates a parser from that description
- Note: This is only true for relatively small languages. Turns out that the parsers for large languages such as Python or Java are written by hand and are typically sLL(1) with many hand coded exceptions built in.

Parser Generators



That looks very much like a translator!

Parser Generators



Parser generators are an example of a domain specific language translator!

Ply is a parser generator, it translates a grammar specification into parser code written in Python.

Using Ply

- Recall:
  - The examples assume that you have cloned or downloaded the Plipy book and have access to the 'code' folder on your local machine
  - For notebook demos it is assumed that you navigated Jupyter to the 'code' folder and started a new notebook
- Documentation on Ply can be found here:
  - http://www.dabeaz.com/ply/ply.html
- Documentation on Ply grammar specifications can be found here:
  - http://www.dabeaz.com/ply/ply.html#ply_nn23

Using Ply

- This is our ‘exp0_gram.py’ file
- In Ply the grammar is specified in the docstring of the grammar functions
- Don’t worry about the lex stuff – it simply sets up a character input stream for the parser to read
- Goal is to generate a parser from this specification

```
from ply import yacc
from exp0_lex import tokens, lexer

def p_grammar(_):
  """
  prog : stmt prog
       | empty

  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'
  """
  pass

def p_empty(p):
    'empty :'
  pass

def p_error(t):
    print("Syntax error at '%s'" % t.value)

parser = yacc.yacc(debug=False,tabmodule='exp0parsetab')
```

Using Ply

```
In [10]: from exp0_gram import parser
         from exp0_lex import lexer

In [11]: parser.parse(input="p + 1 2 ;", lexer=lexer)

In [12]: parser.parse(input="q + 1 2 ;", lexer=lexer)

         Illegal character q
         Syntax error at '+'

In [ ]:
```

Actions

- Making the generated parser do something useful.
- In the hand-coded parser you can add code anywhere in order to make the parser do something useful…like counting ‘p’ statements.
- In parsers generated by parser generators we use something called ‘actions’ we insert into the grammar.
- In Ply actions are inserted into the grammar specification as Python code:

```
def p_exp_var(_):
    '''
    exp : var
    '''
    global count
    count += 1
```

Actions

Actions

- In order to insert actions we need to break the Ply grammar into smaller functions
- Let's try something:
  - The idea of our language processor is to count the number of *right-hand side variables* in a program

```
def p_grammar(_):
  """
  prog : stmt prog
       | empty

...
```

```
def p_prog(_):
  '''
  prog : stmt prog
  '''
  pass

def p_prog_empty(_):
  '''
  prog : empty
  '''
  print("count = {}".format(count))
```

Actions

Actions

```
def p_grammar(_):
  """
  ...
   exp : '+' exp exp
        | '–' exp exp
        | '(' exp ')'
        | var
        | num
  ...
  """
```

Actions

```
def p_exp(_):
  '''
  exp : '+' exp exp
     | '–' exp exp
     | '(' exp ')'
     | num
  '''
  pass

def p_exp_var(_):
  '''
    exp : var
  '''
    global count
    count += 1
```

Actions

```
In [1]: from exp0_count import parser, init_count
        from exp0_lex import lexer

In [2]: init_count()
        parser.parse(input="s x + y 1 ;", lexer=lexer)

        count = 1

In [3]: init_count()
        parser.parse(input="s x + y 1 ; p x ;", lexer=lexer)

        count = 2

In [ ]:
```

Assignment

- Assignment #2 – see website