Parser Generators

Chap 13

- Up till now we have constructed parsers by hand for our language implementations.
- Given some of the repetitive work involved you probably have asked yourself if some of that can be automated.
- The answer is: Yes!
  - Parser generators will process a grammar specification and generate code that implements a parser.

Lex/YACC

- The most well-known parser generator tool set is Lex/YACC
  - Lex – LEXical analyzer
  - YACC – Yet Another Compiler Compiler
- These tools were developed by the original Unix creators in order to be able to create “little languages” very fast.
- Lex is a regular expression based lexical analyzer (very similar to our lexer)
- YACC creates **bottom-up** parsers.
- We will be using an implementation of Lex/YACC in Python called *PLY.*

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 ;

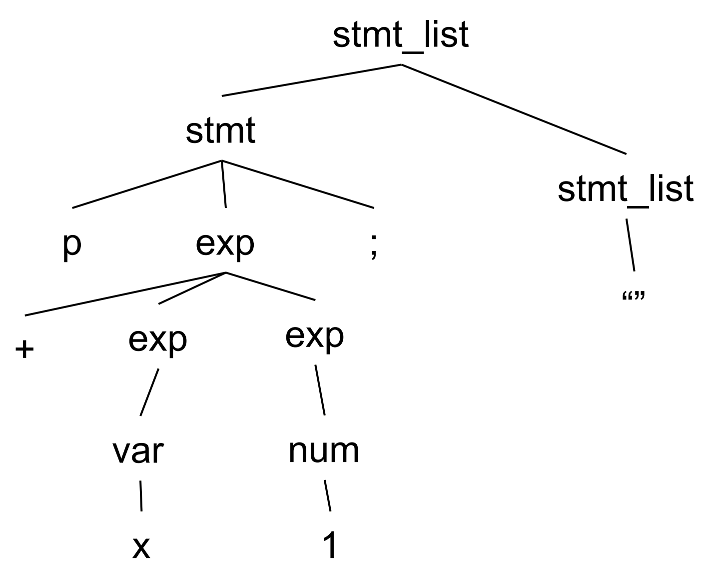
```
stmt_list : stmt stmt_list
          | ""

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
```

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 stmt_list	<empty>	Reduce
stmt_list	<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 stmt_list
stmt_list

p + x 1 ;



Bottom-Up Parsing – LR(1)

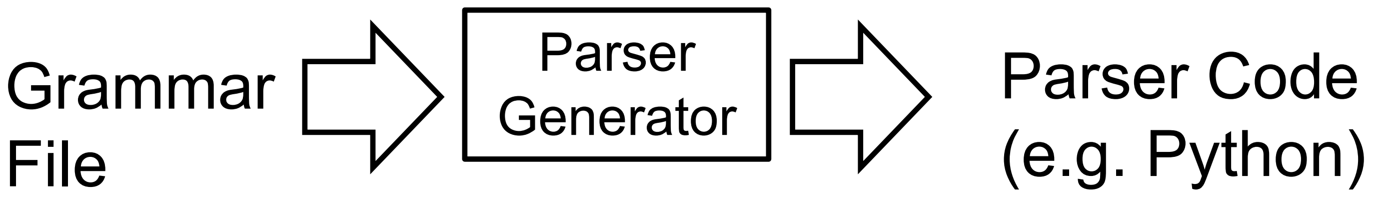
Let's try an illegal sentence

p + x s ;

```
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
```

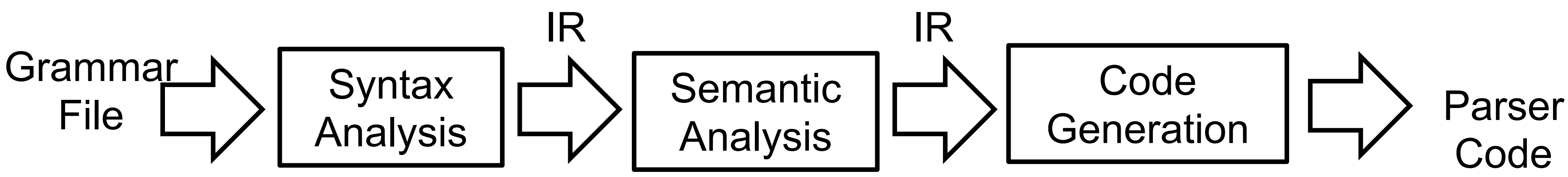
Stack	Input	Action
<empty>	p + x s ;	Shift
p	+ x s ;	Shift
p +	x s ;	Shift
p + x	s ;	Reduce
p + var	s ;	Reduce
p + exp	s ;	Shift
p + exp s	;	Shift
p + exp s ;	<empty>	Reject

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

- 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

YACC Specification of Exp0

- We will use Exp0 as our example language using Ply.

Using Ply

- This is the 'exp0_gram.py' file
- In Ply the grammar is specified in the **docstring** of the grammar functions
- Goal is to generate a parser from this specification
- The lex part is specified in a separate file 'exp0_lex.py'

```
from ply import yacc
from exp0_ply_lex import tokens, lexer

def p_grammar(p):
    '''
    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'
    '''
    pass

def p_error(t):
    raise SyntaxError("syntax error at '{}'".format(t.value))

### build the parser
parser = yacc.yacc()
```

```
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
```

Lex

- The ‘exp0_lex.py’ file

```
##### Lexer for Exp0

from ply import lex

# variables that we need to define for the lexical analysis

tokens = ['NEWLINE'] # we have to define at least one token!

literals = [
    ';',
    'p',
    's',
    '+',
    '-',
    '(',
    ')',
    'x',
    'y',
    'z',
    '0',
    '1',
    '2',
    '3',
    '4',
    '5',
    '6',
    '7',
    '8',
    '9'
]

t_ignore = ' \t'

def t_NEWLINE(t):
    r'\n'
    pass

def t_error(t):
    print("Illegal character %s" % t.value[0])
    t.lexer.skip(1)

# build the lexer
lexer = lex.lex(debug=0)
```

Driver

```
# main driver for the exp0 parser
# it reads from stdin

from sys import stdin
from exp0_gram import parser
from exp0_lex import lexer

char_stream = stdin.read()
try:
    parser.parse(char_stream,lexer=lexer)
    print("parse successful.")
except Exception as e:
    print("error: " + str(e))
```

Running the Parser

```
> python3 exp0_ply_parser.py
WARNING: Token 'NEWLINE' defined, but not used
WARNING: There is 1 unused token
Generating LALR tables
s x 1;
parse successful.
>
```

Look at ‘parser.out’

YACC Grammars

- For SMALL languages YACC grammars tend to be very natural
  - e.g. Cuppa1 grammar
- However, bottom-up parsing breaks down for large, production-level languages

Cuppa1 Grammar

```
'''
program : stmt_list

stmt_list : stmt_list stmt
          |

stmt : ID '=' exp opt_semi
     | GET ID opt_semi
     | PUT exp opt_semi
     | WHILE '(' exp ')' stmt
     | IF '(' exp ')' stmt
     | IF '(' exp ')' stmt ELSE stmt
     | '{' stmt_list '}'

opt_semi : ';'
         |

exp : exp PLUS exp
    | exp MINUS exp
    | exp MUL exp
    | exp DIV exp
    | exp EQ exp
    | exp LE exp
    | INTEGER
    | ID
    | '(' exp ')'
    | MINUS exp %prec UMINUS
    | NOT exp
'''
```

```
precedence = (
              ('left', 'EQ', 'LE'),
              ('left', 'PLUS', 'MINUS'),
              ('left', 'MUL', 'DIV'),
              ('right', 'UMINUS', 'NOT')
             )
```

Words in capital letters are tokens!

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…
- 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.
- Details in the PLY documentation and the book.

Conflicts

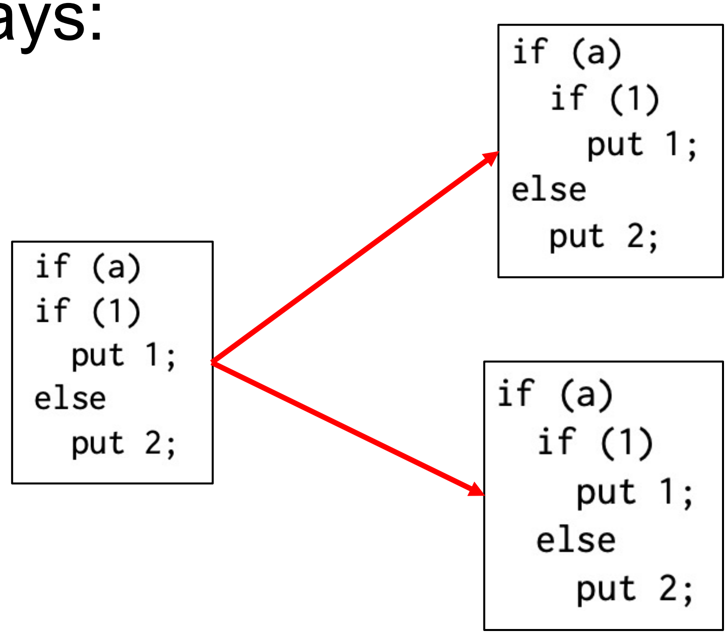
- **Bottom-up parsers** take a **global view** of the grammar – they search the right sides of *all* rules to find a reduction.
- **Top-down parsers** take a **local view** of the grammar – they only search for applicable rules within the appropriate non-terminal.

Conflicts

- The global view of grammars in bottom-up parsers leads to a phenomenon called *conflicts.*
- There are two type of conflicts:
  - Shift/reduce conflicts
  - Reduce/reduce conflicts

Shift/Reduce Conflicts

- The classical example of a shift/reduce conflict is the if-then-else statement.
- In most programming languages the if-then-else statement is inherently ambiguous. Consider the two nested if-statements which can be interpreted in two distinct ways:



Here we use indentation to illustrated association

Shift/Reduce Conflicts

- This ambiguity shows up as a shift/reduce conflict in YACC
- YACC has a default mechanism to deal with this conflict: always shift
  - In this case, that means that the ‘else’ part with always be associated with the closest ‘if’ statement:

```
if (a)
  if (1)
    put 1;
  else
    put 2;
```

Cuppa1

- The shift/reduce conflict in Cuppa1 is due to the if-then-else.
- Here is the YACC grammar snippet of the Cuppa1 statements:

```
stmt : ID '=' exp opt_semi
     | GET ID opt_semi
     | PUT exp opt_semi
     | WHILE '(' exp ')' stmt
     | IF '(' exp ')' stmt
     | IF '(' exp ')' stmt ELSE stmt
     | '{' stmt_list '}'
```

Cuppa1

- We can look at the generated 'parser.out' file to see what YACC has to say about this conflict:

```
state 48

    (8) stmt -> IF ( exp ) stmt .
    (9) stmt -> IF ( exp ) stmt . ELSE stmt

  ! shift/reduce conflict for ELSE resolved as shift
```

Reduce/Reduce Conflicts

- Reduce/reduce conflicts are dreaded in the language implementation community
- Usually that means that you have two syntactic entities that look very similar but appear in different contexts
- Because YACC takes a global view of the rules it cannot detect the context and therefore it cannot decide which rule to use to provide a reduce action.

Reduce/Reduce Conflict Example

- Consider the grammar snippet of a very simple language that does pattern matching in nested parentheses
- Notice that expressions and patterns look exactly the same
  - the difference is that patterns appear on the left side of an assignment and expressions on the right side.

```
stmtlist : stmtlist stmt
         |

stmt : pattern '=' exp
     | exp

exp : ID
    | '(' ')'
    | '(' exp ')'

pattern : ID
        | '(' ')'
        | '(' pattern ')'
...
```

Right hand sides look Identical!
-> ID
-> '(' ')'

Reduce/Reduce Conflict Example

- We would expect that YACC will get confused by the fact that ID and '(' ')' are right sides for two sets of rules.

```
stmtlist : stmtlist stmt
         |

stmt : pattern '=' exp
     | exp

exp : ID
    | '(' ')'
    | '(' exp ')'

pattern : ID
        | '(' ')'
        | '(' pattern ')'
```

```
state 5

    (8) pattern -> ID .
    (5) exp -> ID .

  ! reduce/reduce conflict for ) resolved using rule 5 (exp -> ID .)
```

```
state 8

    (9) pattern -> ( ) .
    (6) exp -> ( ) .

  ! reduce/reduce conflict for ) resolved using rule 6 (exp -> ( ) .)
```

```
WARNING: Conflicts:
WARNING:
WARNING: reduce/reduce conflict in state 5 resolved using rule (exp -> ID)
WARNING: rejected rule (pattern -> ID) in state 5
WARNING: reduce/reduce conflict in state 8 resolved using rule (exp -> ( ))
WARNING: rejected rule (pattern -> ( )) in state 8
```

Reduce/Reduce Conflict Example

- The fact that YACC outright rejected a set of rules means that the generated parser will not work correctly
- One way to fix this is to acknowledge that these two syntactic entities look the same and therefore we make them the same syntactic entity and deal with differences between them at the semantic level.

```
'''
stmtlist : stmtlist stmt
         |

stmt : pattern '=' exp
     | exp

pattern : exp

exp : ID
    | '(' ')'
    | '(' exp ')'
'''
```