Creating a grammar

Marcin Kuta

Theory of Compilation
Laboratory 2

Types of parsers

- LL parsers
  - ANTLR (Python, Java,...)
  - JavaCC
- LR parsers
  - yacc, bison (C, C++, ...)
  - SLY (Python)
  - LARK (Python)

Grammar transformations

Necessary transformation for LL and LR grammars

- LL parsers
  - left recursion removal
  - left factorization
- LR parsers
  - Grammar augmentation with a new start symbol and production $\langle new_start_symbol \rangle \rightarrow \langle old_start_symbol \rangle$

Example

Grammar of arithmetic expressions

```
<expr> -> <expr> '+' <term>
        |  <expr> '-' <term>
        |  <term>

<term> -> <term> '*' <factor>
        |  <term> '/' <factor>
        |  <factor>

<factor> -> '(' <expr> ')'
          |  id
```

Grammar transformations for ANTLR

- In practice, left recursion removal is not performed
- Grammar is specified in EBNF (extended BNF) instead of BNF

Example

```
<expr> -> <term> (('+'|'-') <term>)*

<term> -> <factor> (('*'|'/') <factor>)*

<factor> -> '(' <expr> ')'
          |  id
```

Grammar transformations

- Explicit left recursion removal

Example

- Grammar of arithmetic expressions after left recursion removal

```
<expr> -> <term> <exprp>

<exprp> -> '+' <term> <exprp>
         |  '-' <term> <exprp>

<term> -> <factor> <termp>

<termp> -> '*' <factor> <termp>
         |  '/' <factor> <termp>

<factor> -> '(' <expr> ')'
          |  id
```

Grammar transformations for SLY

Example

- Original unambiguous grammar of arithmetic expressions can be used
- But it makes parser table and parse tree larger than necesssary
- It also slows down parsing

```
<expr> -> <expr> '+' <term>
        |  <expr> '-' <term>
        |  <term>

<term> -> <term> '*' <factor>
        |  <term> '/' <factor>
        |  <factor>

<factor> -> '(' <expr> ')'
          |  id
```

Grammar transformations for SLY

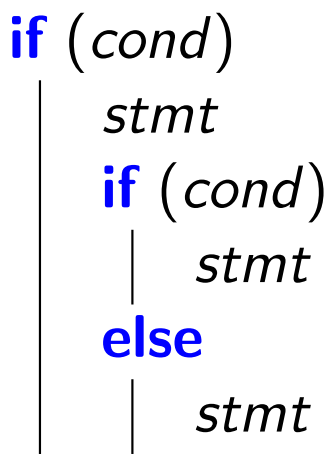
Example

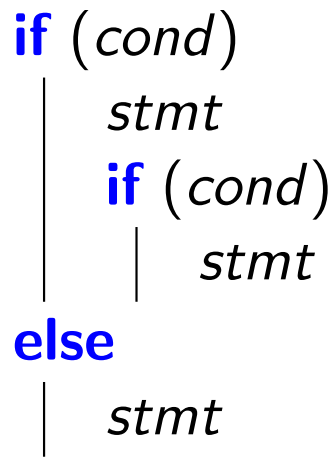
- A better way: use simple, ambiguous grammar of arithmetic expressions and resolve conflicts explicitly by specifying precedence and associativity of operators
- Operators with lower priority come first in the precedence list

```
precedence = (
    ...
    ("left", '+', '-'),
    ("left", '*', '/'),
    ...
)
...
<expr> -> <expr> '+' <expr>
        | <expr> '-' <expr>
        | <expr> '*' <expr>
        | <expr> '/' <expr>
        | '(' <expr> ')'
        | id
```

Dangling else ambiguity

```
if_stmt : IF '(' expr ')' stmt
        | IF '(' expr ')' stmt ELSE stmt
```





Dangling else ambiguity

Ways of resolving the conflict

- Transforming into equivalent unambiguous grammar
- Correct but not practical (grammar becomes more complex than necessary)

```
stmt : matched
     | unmatched

matched : IF '(' expr ')' matched ELSE matched
        | other_stmt

unmatched : IF '(' expr ')' stmt
          | IF '(' expr ')' matched ELSE unmatched
```

Dangling else ambiguity

Resolving the shift-reduce conflict

- Shifting preferred over reduce
- Solution - we resolve conflict by choosing shift over reduce:

```
precedence = (
    ("nonassoc", 'IFX'),
    ("nonassoc", 'ELSE')
)

...

if_stmt : IF '(' expr ')' stmt %prec IFX
        | IF '(' expr ')' stmt ELSE stmt
```

Dangling else ambiguity

Resolving the shift-reduce conflict

- Shifting preferred over reduce
- Solution - we resolve conflict by choosing shift over reduce:

```
precedence = (
    ("right", 'IFX', 'ELSE')
)

...

if_stmt : IF '(' expr ')' stmt %prec IFX
        | IF '(' expr ')' stmt ELSE stmt
```

Ambuguity - source of conflicts

- Amiguous grammars cannot be LR and cause conflicts in parser tables

Example - matrix specification

```
outerlist -> outerlist innerlist
outerlist -> innerlist
innerlist -> innerlist elem
innerlist -> elem
```

- Language modification by introduction of separators between vectors (;) and between vector elements (,)
- Language modification enables grammar modification into unambiguous grammar

```
outerlist -> outerlist ; innerlist
outerlist -> innerlist
innerlist -> innerlist , elem
innerlist -> elem
```

LR grammar

LR grammar that is not LALR

```
def : param_spec return_spec ','

param_spec : type
           | name_list ':' type

return_spec : type
            | name ':' type

type : ID

name : ID

name_list : name
          | name ',' name_list
```

LR grammar

LR grammar that is LALR

```
def : param_spec return_spec ','

param_spec : type
           | name_list ':' type

return_spec : type
            | ID ':' type

type : ID

name : ID

name_list : name
          | name ',' name_list
```

```
bison spec.y -Wcex/-Wcounterexamples
```

Recursion in LR grammar

Left recursion

```
exprseq : expr
        | exprseq ',' expr
```

- parsing a sequence of any number of elements requires only bounded stack space
- a list should be reversed

Right recursion

```
exprseq : expr
        | expr ',' exprseq
```

- parsing a sequence of any number of elements requires linear stack space
- adding element to a list is straightforward

Grammar specification

```
@_('expr "+" expr',
   'expr "-" expr',
   'expr "*" expr',
   'expr "/" expr')
def expr(self, p):
    pass

@_('"(" expr ")"')
def expr(self, p):
    pass
```

Error handling

```
@_('"(" expr ")"')
def expr(self, p):
    pass

@_('IF "(" expr ")" instr ELSE instr')
def instr(self, p):
    pass
```

```
@_('"(" error ")"')
def expr(self, p):
    pass

@_('IF "(" error ")" instr ELSE instr')
def instr(self, p):
    pass
```

References

1. https://sly.readthedocs.io/en/latest/sly.html, Sect. Writing a Parser
2. https://www.gnu.org/software/bison/manual/bison.pdf