Parsing with Lex & Yacc

Stepan Kuznetsov

Discrete Math Bridging Course, HSE University

HW # 1: Practice in Boolean Logic

For the 1st home assignment, choose one of the following tasks:

1. Given a Boolean formula, translate it into Conjunctive Normal Form and into Disjunctive Normal Form.
2. Given a Boolean formula in 2-CNF (in which clauses could also be of the form `(p->q)`), use the resolution method to determine whether it is satisfiable.

HW # 1: Practice in Boolean Logic

- First, one needs to translate the input into a machine-digestable form.

HW # 1: Practice in Boolean Logic

- First, one needs to translate the input into a machine-digestable form.
- Grammar for Boolean formulae:

```
Fm ::= Var | (Fm \/ Fm) | (Fm /\ Fm) | (Fm -> Fm)
```

HW # 1: Practice in Boolean Logic

- First, one needs to translate the input into a machine-digestable form.
- Grammar for Boolean formulae:

```
Fm ::= Var | (Fm \/ Fm) | (Fm /\ Fm) | (Fm -> Fm)
```

- Another grammar:

```
Fm ::= Var | (Fm) | Fm \/ Fm | Fm /\ Fm | Fm -> Fm
```

HW # 1: Practice in Boolean Logic

- First, one needs to translate the input into a machine-digestable form.
- Grammar for Boolean formulae:

```
Fm ::= Var | (Fm \/ Fm) | (Fm /\ Fm) | (Fm -> Fm)
```

- Another grammar:

```
Fm ::= Var | (Fm) | Fm \/ Fm | Fm /\ Fm | Fm -> Fm
```

- The second grammar is **ambiguous:** for example, what does "`p \/ q -> r`" mean?

HW # 1: Practice in Boolean Logic

- First, one needs to translate the input into a machine-digestable form.
- Grammar for Boolean formulae:

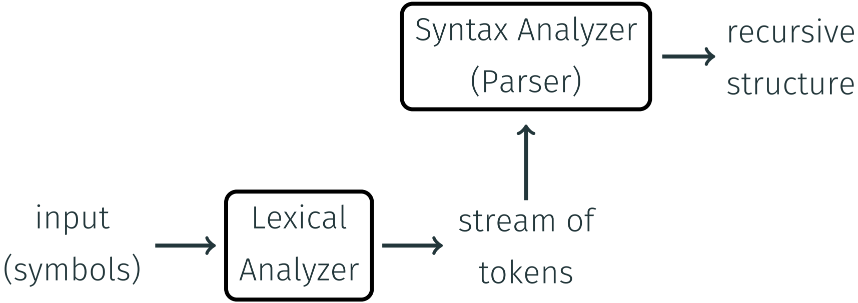
```
Fm ::= Var | (Fm \/ Fm) | (Fm /\ Fm) | (Fm -> Fm)
```

- Another grammar:

```
Fm ::= Var | (Fm) | Fm \/ Fm | Fm /\ Fm | Fm -> Fm
```

- The second grammar is **ambiguous:** for example, what does "`p \/ q -> r`" mean? We have to specify priority and association rules.

The Parsing Workflow



Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

- Output (stream of **tokens**):

Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

- Output (stream of **tokens**):

```
KW_INT
```

Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

- Output (stream of **tokens**):

`KW_INT` `IDENT('main')`

Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

- Output (stream of **tokens**):

KW_INT   IDENT('main')   '('

Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

- Output (stream of **tokens**):

KW_INT   IDENT('main')   '('   KW_VOID

Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

- Output (stream of **tokens**):

`KW_INT` `IDENT(‘main’)` `‘(’` `KW_VOID` …

Lexical Analysis

- Input (stream of **symbols**):

```
int main(void)
{
    printf("Hello, World!\n");
}
```

- Output (stream of **tokens**):

`KW_INT` `IDENT('main')` `'('` `KW_VOID` …

- Tokens are much more convenient to work with (in the grammar).

Running Example: Simplifying Polynomials

- We consider the following task: translating polynomials into normal form.

Running Example: Simplifying Polynomials

- We consider the following task: translating polynomials into normal form.

$$(2x + 2)(3x^2 - 1) + 2x = 6x^3 + 6x - 2$$

Running Example: Simplifying Polynomials

- We consider the following task: translating polynomials into normal form.

$$(2x + 2)(3x^2 - 1) + 2x = 6x^3 + 6x - 2$$

- Grammar:

```
Expr    ::= Tm | -Tm | Expr + Tm | Expr - Tm
Tm      ::= Mon | (Expr) | Tm (Expr)
Mon     ::= Int_opt 'x' Pow_opt | INT
Int_opt ::= INT | ε
Pow_opt ::= '^' INT | ε
```

Running Example: Simplifying Polynomials

- We consider the following task: translating polynomials into normal form.

$$(2x+2)(3x^2-1)+2x = 6x^3+6x-2$$

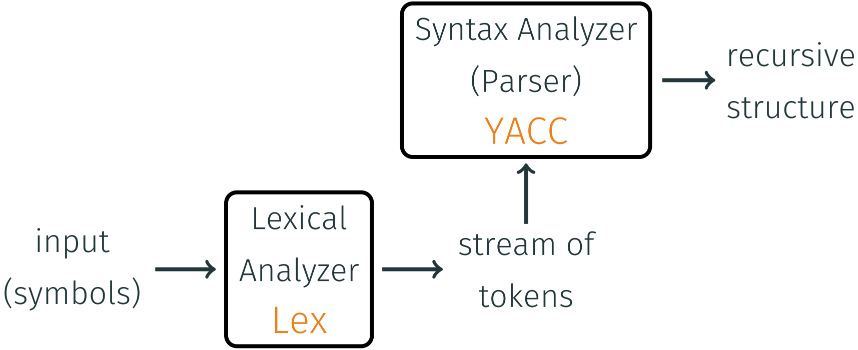
- Grammar:

```
Expr    ::= Tm | -Tm | Expr + Tm | Expr - Tm
Tm      ::= Mon | (Expr) | Tm (Expr)
Mon     ::= Int_opt 'x' Pow_opt | INT
Int_opt ::= INT | ε
Pow_opt ::= '^' INT | ε
```

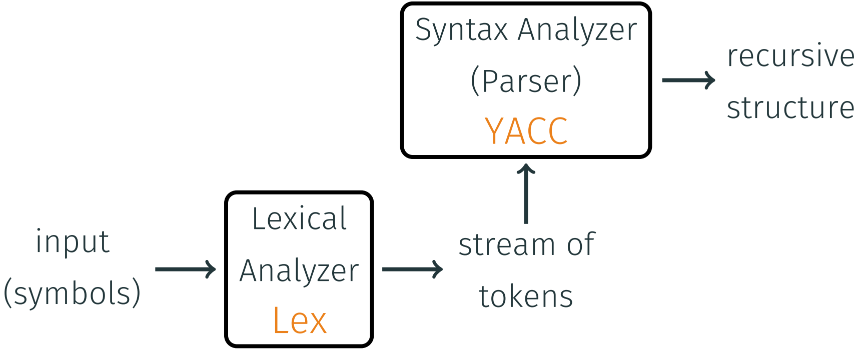
- Input example:

```
(2x+2)(3x^2-1)+2x
```

Implementation: Lex & Yacc

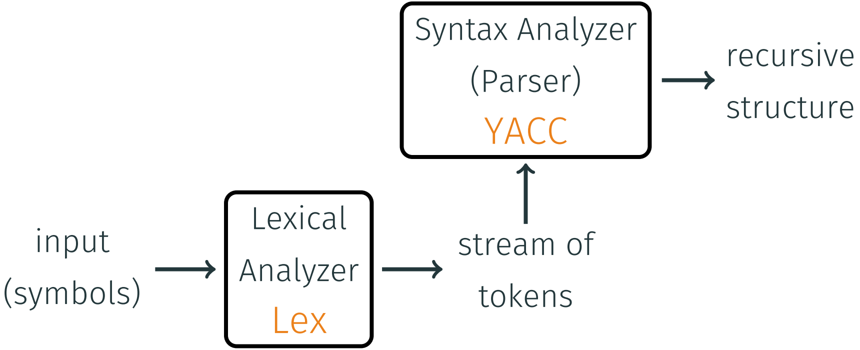


Implementation: Lex & Yacc



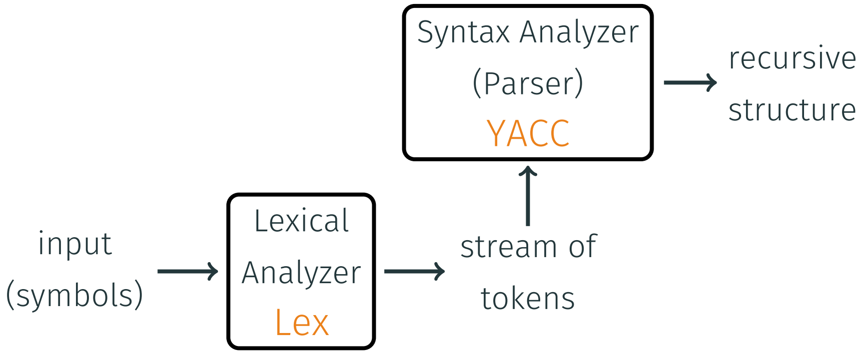
- YACC = Yet Another Compiler Compiler

Implementation: Lex & Yacc



- YACC = Yet Another Compiler Compiler

Implementation: Lex & Yacc



- YACC = Yet Another Compiler Compiler
- In Python, we use PLY (Python Lex & Yacc).

PLY Code for Lexical Analysis

- Declare tokens and literals (one-symbol tokens):

```
tokens = [ 'INT' ]
literals = ['+','-','(',')','^','x']
```

PLY Code for Lexical Analysis

- Declare tokens and literals (one-symbol tokens):

```
tokens = [ 'INT' ]
literals = ['+','-','(',')','^','x']
```

- For each token, declare a “`t_`”-function:

```
def t_INT(t):
    r'\d+'
    try:
        t.value = int(t.value)
    except ValueError:
        print "Too large!", t.value
        t.value = 0
    return t
```

PLY Code for Lexical Analysis

- `r'\d+'` is a regular expression for sequences of decimal numbers.

PLY Code for Lexical Analysis

- `r'\d+'` is a **regular expression** for sequences of decimal numbers.
- Another example: regular expression for **names** (identifiers)

```
t_NAME     = r'[a-zA-Z_][a-zA-Z0-9_]*'
```

PLY Code for Lexical Analysis

- `r'\d+'` is a **regular expression** for sequences of decimal numbers.
- Another example: regular expression for **names** (identifiers)

```
t_NAME     = r'[a-zA-Z_][a-zA-Z0-9_]*'
```

- Finally, build the lexer:

```
import ply.lex as lex
lex.lex()
```

PLY Code for Parsing

- Each rule of the grammar is implemented as a "`p_`"-function:

```
def polymult(p,q) :
    r = []
    for i in xrange(len(p)) :
        for j in xrange(len(q)) :
            safeadd(r,i+j,p[i]*q[j])
    return r
```

...

```
def p_tm_mult(p):
    "tm : tm '(' expr ')'"
    p[0] = polymult(p[1],p[3])
```

PLY Code for Parsing

```
def p_tm_mult(p):
    "tm : tm '(' expr ')'"
    p[0] = polymult(p[1],p[3])
```

PLY Code for Parsing

```
def p_tm_mult(p):
    "tm : tm '(' expr ')'"
    p[0] = polymult(p[1],p[3])
```

- A "`p_`"-function generates an object `p[0]`, using `p[1]`, `p[2]`, ..., which are obtained from the lexer or recursively from parsing.

PLY Code for Parsing

- Finally, build the parser:

```
import ply.yacc as yacc
yacc.yacc()
```

PLY Code for Parsing

- Finally, build the parser:

```
import ply.yacc as yacc
yacc.yacc()
```

- The code of PLY examples is available on the course’s webpage:

  http://www.mi-ras.ru/~sk/lehre/dm_hse2019/

PLY Code for Parsing

- Finally, build the parser:

```
import ply.yacc as yacc
yacc.yacc()
```

- The code of PLY examples is available on the course's webpage:

  `http://www.mi-ras.ru/~sk/lehre/dm_hse2019/`

- For priorities, see another example available on the webpage: calculator.

HW # 1: Practice in Boolean Logic

Choose one:

1. Given a Boolean formula, constructed from variables using `/\` (conjunction), `\/` (disjunction), `->` (implication), and `~` (negation), translate it into Conjunctive Normal Form and into Disjunctive Normal Form.

2. Given a Boolean formula in 2-CNF, use the resolution method to determine whether it is satisfiable. Clauses of the 2-CNF can be of one of the two forms: α `\/` β or α `->` β, where α and β are literals (`p` or `~p`, where `p` is a variable). The CNF is presented in the usual notation, for example:
   `(p -> q) /\ (~r \/ s) /\ (~q -> p)`

HW # 1: Practice in Boolean Logic

Choose one:

1. Given a Boolean formula, constructed from variables using `/\` (conjunction), `\/` (disjunction), `->` (implication), and `~` (negation), translate it into Conjunctive Normal Form and into Disjunctive Normal Form.

2. Given a Boolean formula in 2-CNF, use the resolution method to determine whether it is satisfiable. Clauses of the 2-CNF can be of one of the two forms: α `\/` β or α `->` β, where α and β are literals (`p` or `~p`, where `p` is a variable). The CNF is presented in the usual notation, for example:
   `(p -> q) /\ (~r \/ s) /\ (~q -> p)`

Tasks available at the course's webpage.

Good luck!