Languages: Predictive parsing 1

Glenn Bruns

CSUMB

Learning outcomes

After this lecture, you should be able to:

□ write a predictive parser from some BNF rules

Warmup: deriving strings from BNF

```
stmt ::= print ( expr )
       | id = expr
expr ::= id | num
```

num - a sequence of digits

id – an identifier

Can you derive these?

- `print(5)`
- `x = y`
- `x = print(5)`

How to parse?

```
stmt ::= print ( expr )
       | id = expr
expr ::= id | num
```

How to parse a statement?

Idea:

- function stmt() will parse statements
- function expr() will parse expressions
- each function will decide which production to use by looking at the first symbol of each production
- once a production is picked the function will mimic the right-hand side of the production

Parsing example

```
stmt ::= print ( expr )
       | id = expr
expr ::= id | num
```

```
void stmt() {
switch (lookahead) {
      case PRINT:
         match(PRINT); match("("); expr(); match(")");
         break;
      case ID:
         match(ID); match("="); expr()
         break;
      default:
        error("syntax error");
   }
}
```

Predictive parsing

Recursive-descent parsing: "top-down method of syntax analysis in which we execute a set of recursive procedures to process the input" (Aho et al, Dragon book)

Predictive parsing: "a form of recursive-descent parsing in which the lookahead symbol determines the procedure selected for each non-terminal" (Aho et al, Dragon book)

```
void stmt() {
   switch (lookahead) {
      case PRINT:
         match(PRINT); match("("); expr(); match(")"); break;
      case ID:
         match(ID); match("="); expr(); break;
      default:
         error("syntax error");
   }
}
```

Left recursion

Any problems with a predictive parser here?

```
expr ::= expr + term
       | term
```

The BNF can be transformed into this:

```
expr ::= term expr1
expr1 ::= + term expr1 | ""
```

A general rule:

A ::= A α | β

derives the same strings as:

A ::= β R
R ::= α R | ""

(α, β are sequence of terminals and non-terminals that don't start with A)

Exercise

Rewrite to eliminate left recursion

```
expr ::= expr + expr
       | var
```

solution: (use the rule; new non-terminal is 'expr1'l)

```
expr  ::= var expr1
expr1 ::= + expr expr1 |  ""
```

another solution:

```
expr  ::= var expr1
expr1 ::= + var expr1 |  ""
```

$A ::= A\ \alpha \mid \beta$

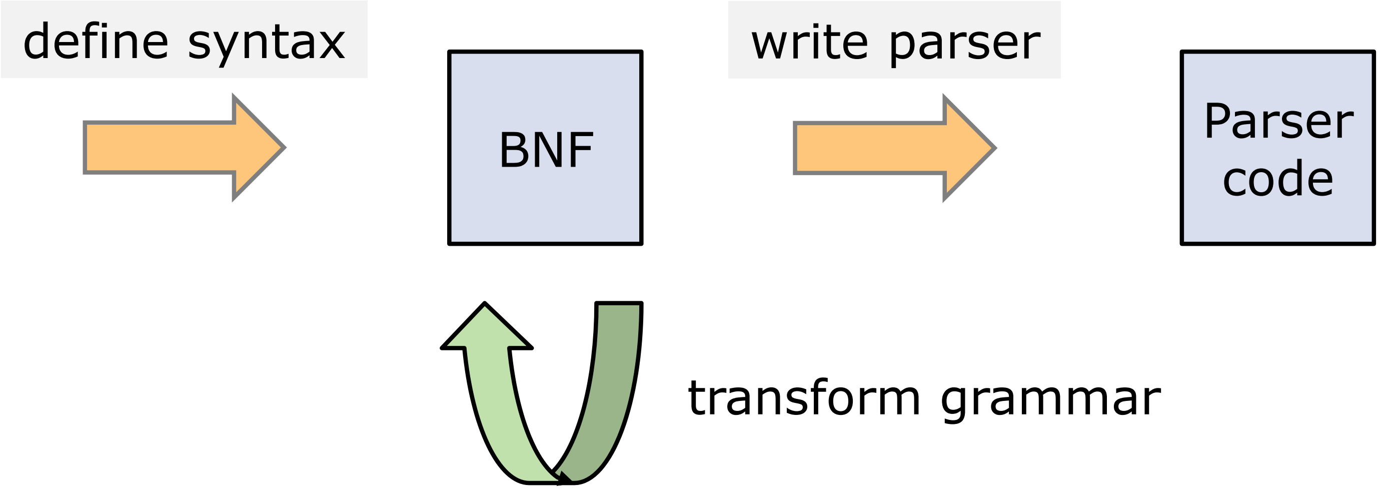
derives the same strings as:

$A ::= \beta\ R$

$R ::= \alpha\ R \mid$ ""

Transforming a grammar

Here's our plan:



define syntax

BNF

write parser

Parser code

transform grammar

We transform the grammar in a way that
that language of the grammar doesn't change.

Dealing with empty productions

How to deal with "" in expr1?

```
expr  ::= var expr
expr1 ::= + expr expr1  |  ""
```

Idea: use the empty production when no other production can be used.

```
void expr1() {
   switch (lookahead) {
      case '+':
         match('+'); expr(); expr1(); break
      default:
         ;    // empty production
}
```

Main program

- initialize lookahead variable
- call function associated with start symbol in grammar
- match on token DONE (end of input)

Example:

```
void parser() {
   lookahead = lexan();
   stmt();        // assuming stmt is start symbol
   match(DONE);
}
```

Summary

- predictive parsing is a simple kind of parser that is based directly on a BNF grammar
- but… you may need to modify a BNF grammar to allow predictive parsing to be used