

# Um analisador recursivo para TINY

---

- Vamos construir um analisador recursivo para TINY de maneira sistemática, gerando uma árvore sintática
- O vetor de tokens vai ser gerado a partir de um analisador léxico escrito com o JFlex

```
S      -> CMDS
CMDS   -> CMD (; CMD)*
CMD    -> if EXP then CMDS (else CMDS)? end
        | repeat CMDS until EXP
        | id := EXP
        | read id
        | write EXP
EXP    -> SEXP (< SEXP | = SEXP)?
SEXP   -> TERMO (+ TERMO | - TERMO)*
TERMO  -> FATOR (* FATOR | / FATOR)*
FATOR  -> "(" EXP ")" | num | id
```

# Detecção de erros

---

- Um analisador recursivo com retrocesso tem um comportamento ruim na presença de erros sintáticos
- Ele não consegue distinguir *falhas* (um sinal de que ele tem que tentar outra possibilidade) de *erros* (o programa está sintaticamente incorreto)
- Uma heurística é manter em uma variável global uma marca d'água que indica o quão longe fomos na sequência de tokens

# Retrocesso local x global

---

- O retrocesso em caso de falha do nosso analisador é *local*. Isso quer dizer que se eu tiver  $(A \mid B) C$  e A não falha mas depois C falha, ele não tenta B depois C novamente
- Da mesma forma, se eu tenho  $A \mid A B$  a segunda alternativa nunca vai ser bem sucedida
- As alternativas precisam ser *exclusivas*
- Retrocesso local também faz a repetição ser *gulosa*  $A^* A \equiv \text{falha}$
- Uma implementação com retrocesso *global* é possível, mas mais complicada

# Recursão à esquerda

$$(P^*)^* \equiv \text{loop infinito}$$

- Outra grande limitação dos analisadores recursivos é que as suas gramáticas não podem ter *recursão à esquerda*
- A presença de recursão à esquerda faz o analisador entrar em um laço infinito!
- Precisamos transformar recursão à esquerda em repetição
- Fácil quando a recursão é direta:

$$A \rightarrow A x_1 \mid \dots \mid A x_n \mid y_1 \mid \dots \mid y_n$$



$$A \rightarrow (y_1 \mid \dots \mid y_n) (x_1 \mid \dots \mid x_n)^*$$

$$\begin{aligned} S_{EXP} &\rightarrow S_{EXP} + TERMO \\ &\mid S_{EXP} * TERMO \\ &\mid TERMO \\ &\Downarrow \\ S_{EXP} &\rightarrow TERMO ( + TERMO \\ &\mid - TERMO )^* \end{aligned}$$

# Parsing Expression Grammars

---

- As *parsing expression grammars* (PEGs) são uma generalização do parser com retrocesso local
- A sintaxe das gramáticas adota algumas características de expressões regulares: \* e + para repetição ao invés de {}, ? para opcional ao invés de []
- Usa-se / para alternativas ao invés de |, para enfatizar que esse é um operador bem diferente do das gramáticas livres de contexto
- Acrescentam-se dois operadores de *lookahead*: *&p* e *!p*
- Finalmente, uma PEG pode misturar a tokenização com a análise sintática, então os terminais são *caracteres* (com sintaxe para strings e classes)

# Uma PEG para TINY

```
S      <- CMDS
CMDS   <- CMD (";" CMD)*
CMD    <- if EXP then CMDS ("else" CMDS)? end
        / repeat CMDS until EXP
        / ID "==" EXP
        / read ID
        / write EXP
EXP    <- SEXP ("<" SEXP / "=" SEXP)*
SEXP   <- TERMO ("+" TERMO / "-" TERMO)*
TERMO  <- FATOR ("*" FATOR / "/" FATOR)*
FATOR  <- "(" EXP ")" / ID / NUM
```

simpático

kw's → if /  
repeat /  
then /  
until /  
end /  
read /  
write

```
! ID      <- SP [a-zA-Z_][a-zA-Z0-9]*
! NUM     <- SP [0-9]+
! SP      <- [ \r\n\t]*
! "lit"   <- SP 'lit'
! kw      <- SP 'kw' ! [a-zA-Z0-9_]
```

! kw's

Lexico

! [a-zA-Z0-9\_] → não q usamos de PEGMP