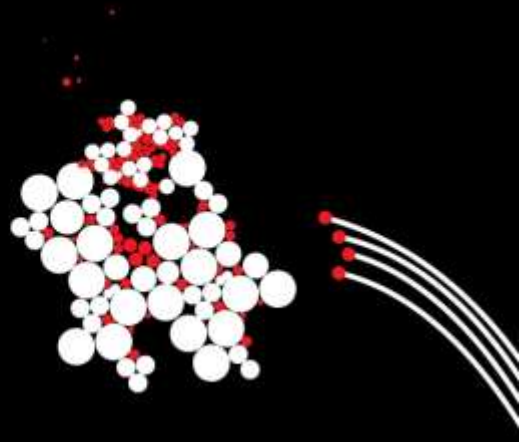
UNIVERSITY OF TWENTE.

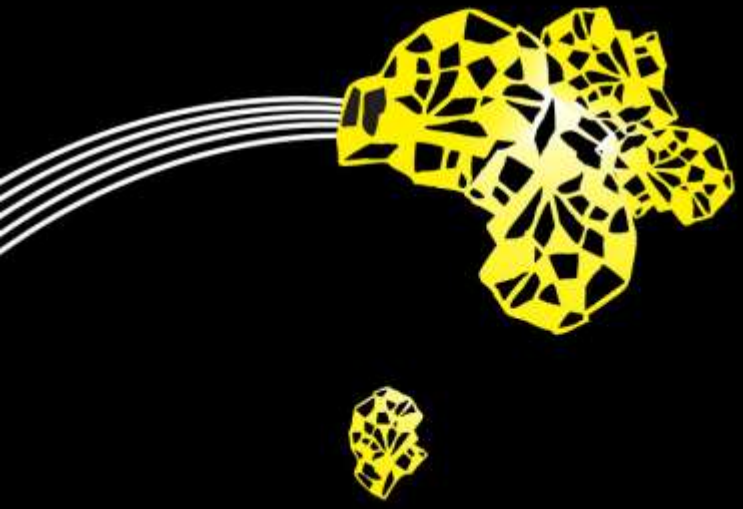


COMPILER CONSTRUCTION:

PARSE TREES & ANTLR (PRIMER)

MODULE 8: PROGRAMMING PARADIGMS

29 APRIL 2019



SEEN BEFORE

- Structure of compiler
- Front end
  - Scanning
  - Parsing
  - Type checking
- Optimizer
- Back end

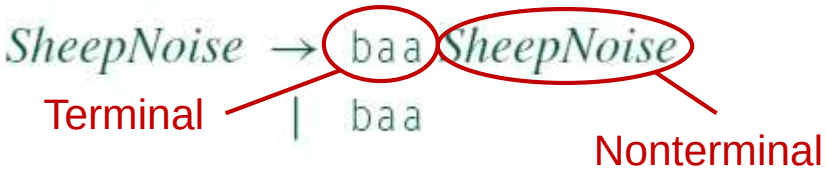
⇐ To be continued today

Parser: turns a sequence of tokens of language-specific token types into a parse tree

(EXTENDED) BACKUS-NAUR FORM ((E)BNF)

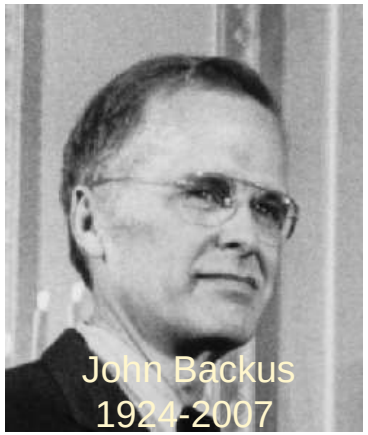
Context-free grammar rules

- *Original*:
  - Left hand side (LHS): Nonterminal
  - Right hand side (RHS): sequence of terminals + nonterminals
  - Multiple rules for one non-terminal, separated by |



- *Extended*: use RE syntax in RHS
  - Can always be rewritten to basic BNF

    SheepNoise → baa +



Example:

- EBNF

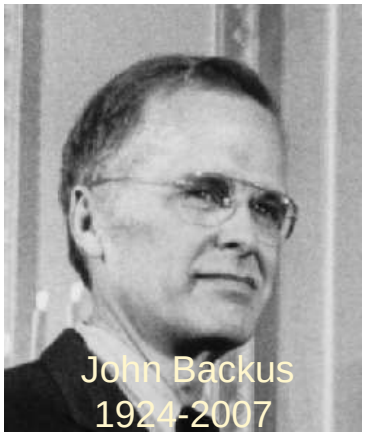
  A → *B*? ((a | b) *C*)*
- Corresponding BNF?

Source: John Backus: www.columbia.edu
Peter Naur: Eriktj, http://commons.wikimedia.org

(EXTENDED) BACKUS-NAUR FORM ((E)BNF)

$A \rightarrow B\,D \mid D$

$D \rightarrow aC \mid bC \mid \epsilon \mid DD$


John Backus
1924-2007


Peter Naur
1928-2016

Example:

- EBNF

  $A \rightarrow B? ((a \mid b)\, C)^*$

- Corresponding BNF?

  $A \rightarrow B\,D \mid D$

  $D \rightarrow a\,C\,D \mid b\,C\,D \mid \epsilon$

Source: John Backus: www.columbia.edu
Peter Naur: Eriktj, http://commons.wikimedia.org

DERIVATIONS AND PARSE TREES

- Derivation: sequence of rule applications
  - Starting at start symbol (duh)
  - Ending at term to be parsed ("accepted", "generated"): *sentence*
  - Intermediate forms: *sentential form* (mixture of terminals and non-terminals)
- Choice of where to apply next rule
  - Leftmost / rightmost non-terminal
- Parse tree: node for each rule application
  - Children are the symbols of the RHS

1	*Expr*	→	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Rule	Sentential Form
	Expr

Parse tree

Expr

Rightmost Derivation of (a + b) × c

DERIVATIONS AND PARSE TREES

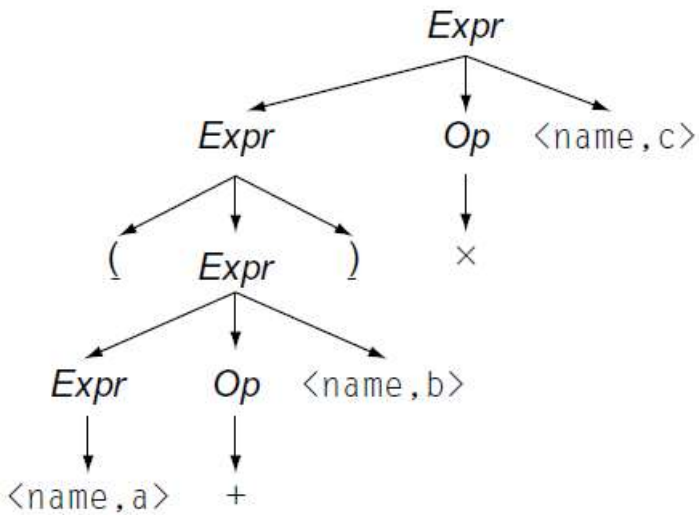
- Derivation: sequence of rule applications
  - Starting at start symbol (duh)
  - Ending at term to be parsed ("accepted", "generated"): *sentence*
  - Intermediate forms: *sentential form* (mixture of terminals and non-terminals)
- Choice of where to apply next rule
  - Leftmost / rightmost non-terminal
- Parse tree: node for each rule application
  - Children are the symbols of the RHS

1	*Expr*	→	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Rule	Sentential Form
	Expr
2	*Expr Op* name
6	*Expr* × name
1	(*Expr*) × name
2	(*Expr Op* name) × name
4	(*Expr* + name) × name
3	(name + name) × name

Rightmost Derivation of (a + b) × c

Parse tree



AMBIGUITY

1) a = 2
2) a = 1 ^ b = 1

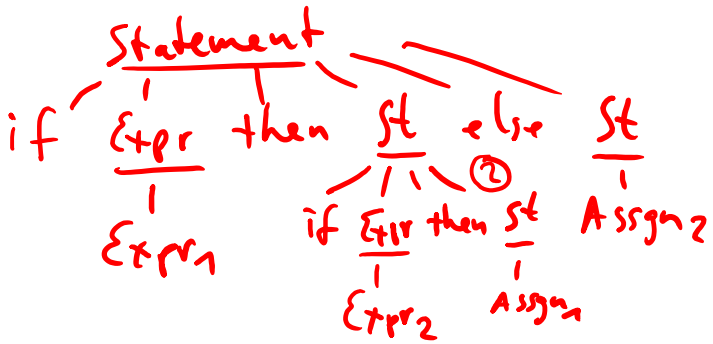
```
if (a == 1)
{ if (b == 2)
    c = 2;
  else c = 3; }
```

Let's parse using grammar

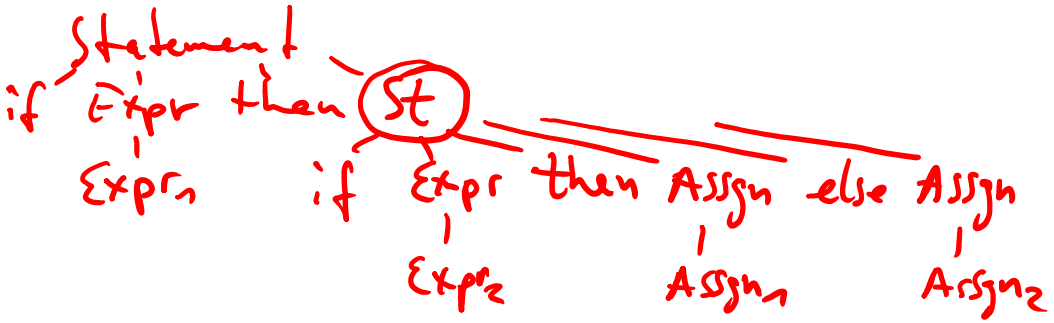
`if` $Expr_1$ `then` `if` $Expr_2$ `then` $Assignment_1$ `else` $Assignment_2$

1	*Statement*	→	`if` *Expr* `then` *Statement* `else` *Statement*	1
2		|	`if` *Expr* `then` *Statement*	2
3		|	*Assignment*	3
4		|	*... other statements ...*	

- Option 1:



- Option 2:



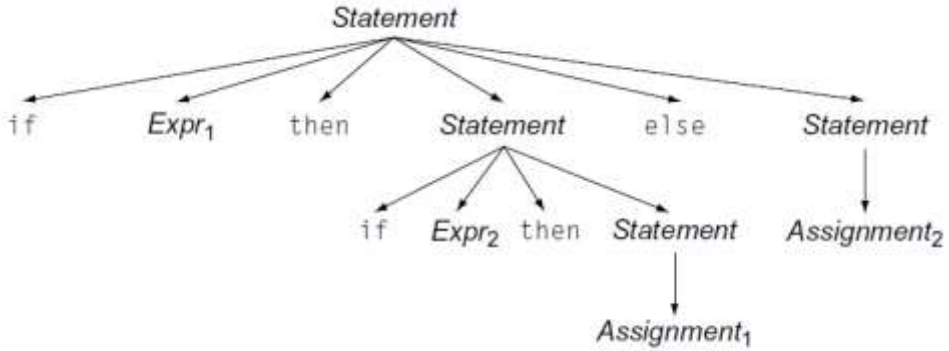
AMBIGUITY

Let's parse using grammar

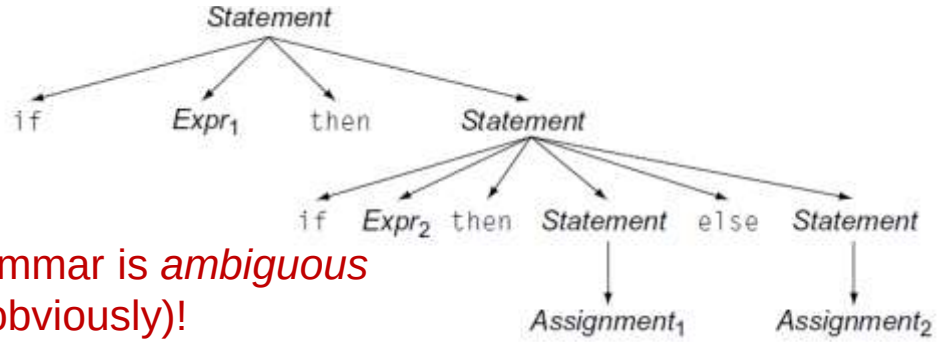
if $Expr_1$ then if $Expr_2$ then $Assignment_1$ else $Assignment_2$

1	*Statement*	→	if *Expr* then *Statement* else *Statement*
2		|	if *Expr* then *Statement*
3		|	*Assignment*
4		|	*... other statements ...*

- Option 1:



- Option 2:



No unique answer: grammar is *ambiguous*
This is to be avoided (obviously)!
(for instance: give priority to one of the cases)

LEFT- AND RIGHT-ASSOCIATIVITY

- What is the difference between

```
Expr → Expr + Operand
     | Expr - Operand
     | Operand
```

```
Expr → Operand + Expr
     | Operand - Expr
     | Operand
```

  - They accept the same language
- Consider the text: `1 - 3 + 4 - 2` = 0 — − + : left-associative
  - What should the outcome be?

LEFT- AND RIGHT-ASSOCIATIVITY

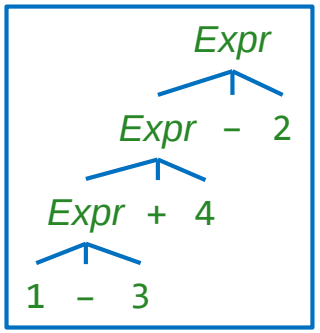
- What is the difference between

```
Expr → Expr + Operand
     | Expr - Operand
     | Operand
```

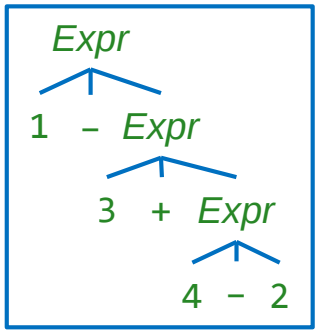
```
Expr → Operand + Expr
     | Operand - Expr
     | Operand
```

  - They accept the same language
- Consider the text: `1 – 3 + 4 – 2`
  - What should the outcome be?

Parse tree left hand grammar:



Parse tree right hand grammar:



- The shape of the parse tree matters for the meaning
  - For – and + (*left associative* operators) the left hand tree is correct
  - For ^ (exponentiation, *right associative*) the right hand tree would be correct

SEEN BEFORE

- Structure of compiler
- Front end
  - Scanning
  - Parsing
  - Type checking

  } Primer on ANTLR
- Optimizer
- Back end

ANTLR LEXER GRAMMARS

MyFirstLanguage.g4

= scanner

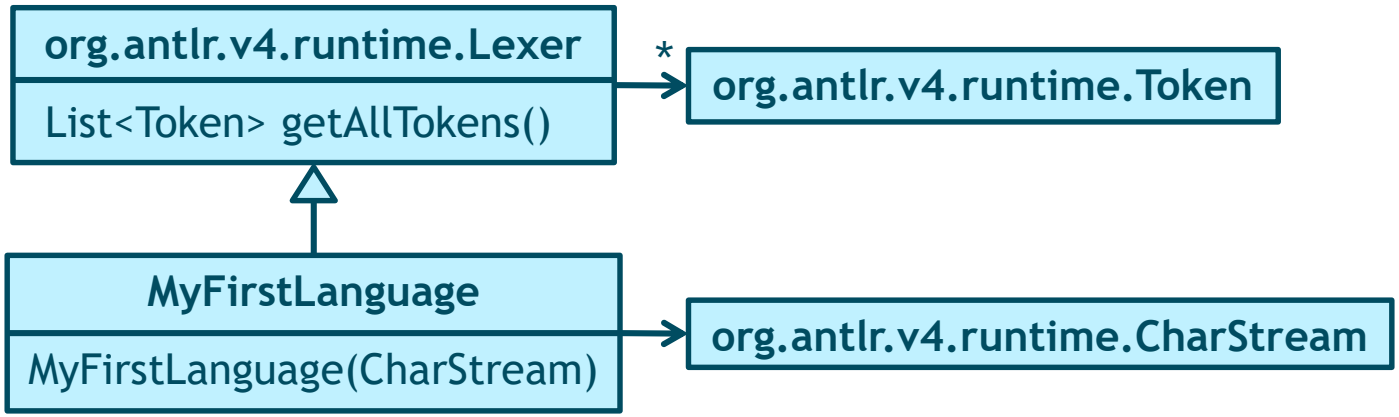
```
lexer grammar MyFirstLanguage;

KEYWORD : 'key';
```

Antlr

MyFirstLanguage.tokens

MyFirstLanguage.java

Structure:



Usage:

```
CharStream stream = CharStreams.fromString(text);
Lexer lexer = new MyFirstLanguage(stream);
for (Token token : lexer.getAllTokens()) {
    System.out.print(token.toString() + " ");
}
```

PRAGMATICS OF ANTLR

- Generated Java files have to be in some package
  - Declare in Antlr grammar file:

```
lexer grammar MyFirstLanguage;

@header{package pp.slides.cc1;}

KEYWORD : 'key';
```

Added to header of `MyFirstLanguage.java`

- Automatic generation in Eclipse plugin sometimes hard to convince
  - Context menu: “Run as -> Generate ANTLR Recognizer” *usually* works
  - Console window:

```
ANTLR Tool v4.7 (…)
MyFirstLanguage.g4 -o . -listener -no-visitor -encoding UTF-8
```

This is the output directory option

- However, IntelliJ does *not* need the **`package`** declaration in `@header`!
  - There is no way to get compatible behaviour; strip the .g4-file!

ANTLR REGULAR EXPRESSION SYNTAX

Lexer rules always start with uppercase

- Single quotes for literal characters or character sequences

```
KEYWORD : 'key' ;
LBRACE : '{' ;
NEWLINE : '\n' ;
```

- *Choice* specified in multiple ways:

```
LETDIG : '0'..'9'|'a'..'z' ;
HEX : [0123456789ABCDEF] ;
NO_QUOTES : ~('\'' | '\"') ;
```

Operator or range (char ordering!)
No quotes! Rule order is important
Anything but the single or double quote

- Sequencing, repetition (zero-or-more, at least once), optional

```
NUMBER : '0' | '1'..'9'('0'..'9')* ;
WORD : ('a'..'z' | 'A'..'Z')+ ;
OPT : 'end' ' please'? ;
```

- Special features

```
fragment LETTER : ('a'..'z'|'A'..'Z');
WORD : LETTER+ ;
WS : [ \t\r\n]+ -> skip ;
STRING : '\"' .*? '\"' ;
```

Auxiliary rule, *not* a token type

Skip: not emitted as token
Turn non-greedy at `*?` (= any)

JAVA BLOCK COMMENTS

- Correct solution:

  /* (^* | *+ ^/)* *+ /

- In Antlr:

```
COMMENT : '/*' (~'*' | '*'+ ~'/')* '*'+'/' ;
```

  - Or, using “non-greedy” special feature of Antlr:

```
COMMENT : '/*' .*? '*/' ;
```

NEXT STEP: ANTLR PARSER GENERATION

parse rules: start with lower case

```
parser grammar MyLanguageParser1;
options { tokenVocab = MyLanguage; }
@header{package pp.slides.cc1;}
main : BEGIN stat* END ;
stat : RETURN SEMI ;
```

specify tokens

```
lexer grammar MyLanguage;
@header{package pp.slides.cc1;}
RETURN : 'return' ;
BEGIN : '{' ;
END : '}' ;
SEMI : ';' ;
WS : [ \r\n\t]+ -> skip ;
```

Alternative: combined grammar

```
grammar MyLanguageParser2;
import MyLanguage;
main : BEGIN stat* END ;
stat : RETURN SEMI ;
```

Or everything in single file:

```
grammar MyLanguageCombined;
@header{package pp.slides.cc1;}
main : BEGIN stat* END ;
stat : 'return' ';' ;
BEGIN : '{' ;
END : '}' ;
WS : [ \r\n\t]+ -> skip ;
```

unnamed terminal