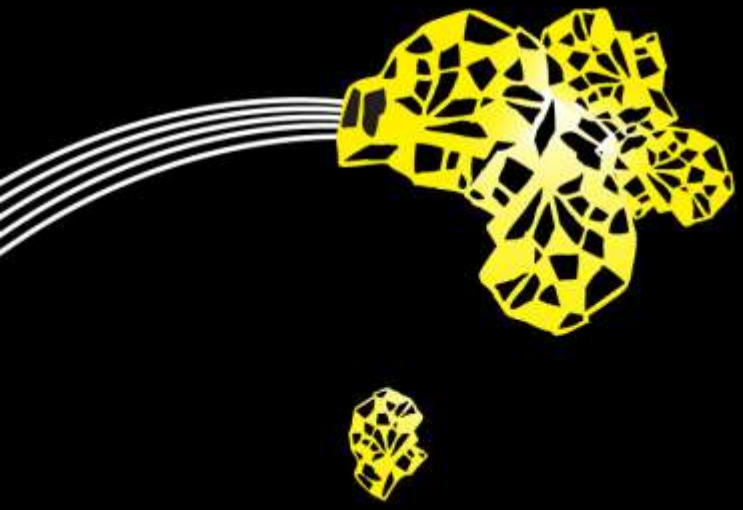
UNIVERSITY OF TWENTE.



COMPILER CONSTRUCTION:

CC 2-1: PREDICTIVE PARSING, LL(1), FIRST

MODULE 8: PROGRAMMING PARADIGMS

30 APRIL 2019

WHERE ARE WE?

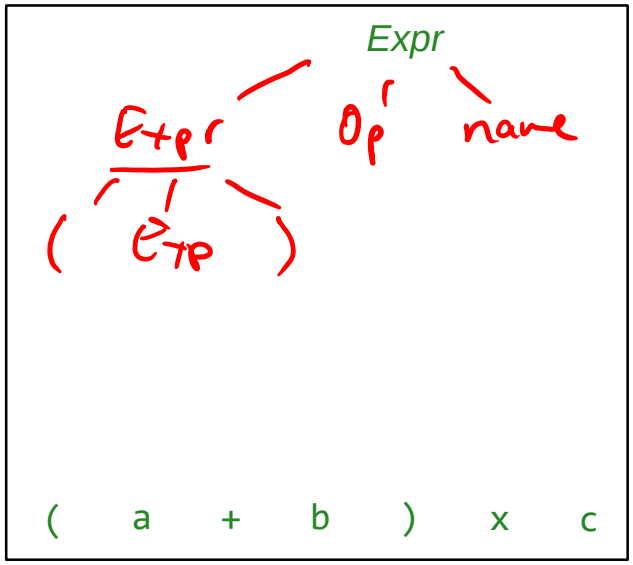
- **Overall structure:** Compiler front end
  - Scanning
  - Parsing ⇦ To be continued today
  - Type checking
- **Seen last time:** Parse trees
  - Correspondence to derivations
  - Grammar ambiguity
  - Operator associativity
- **Today:** Bottom-up versus top-down parsing
  - LL(1) criterion
  - FIRST function for lookahead

BOTTOM-UP VS TOP-DOWN PARSING

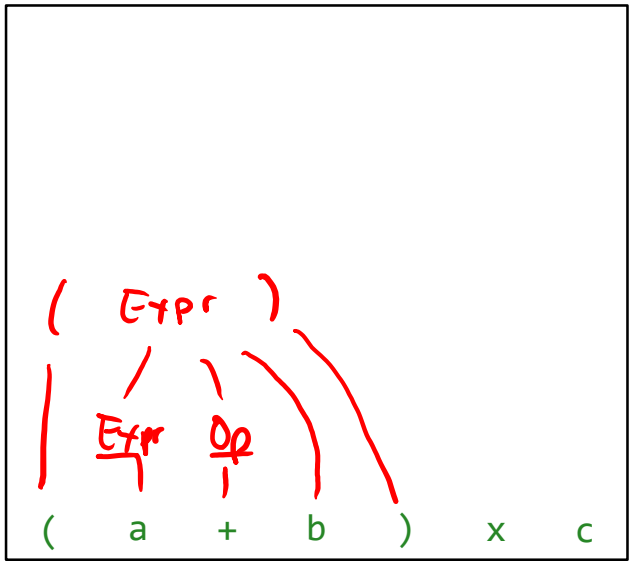
1	*Expr*	→	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

- The task of finding a parse tree starts with
  - Start symbol
  - Text to be parsed
- Two fundamental ways to detect the structure in between

Top-down



Bottom-up

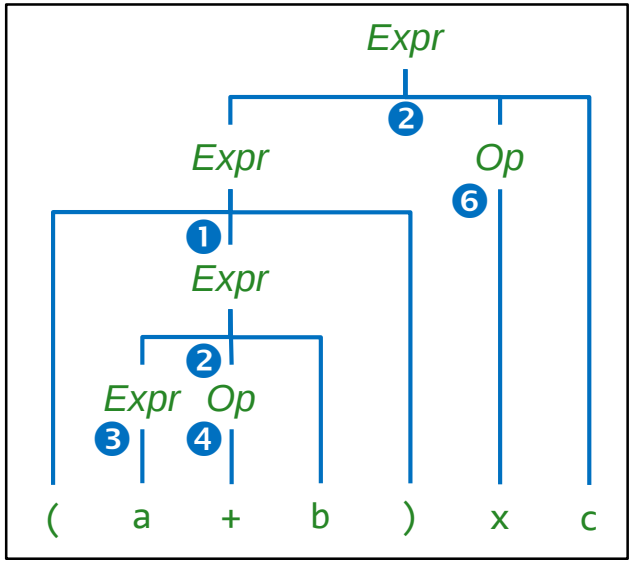


BOTTOM-UP VS TOP-DOWN PARSING

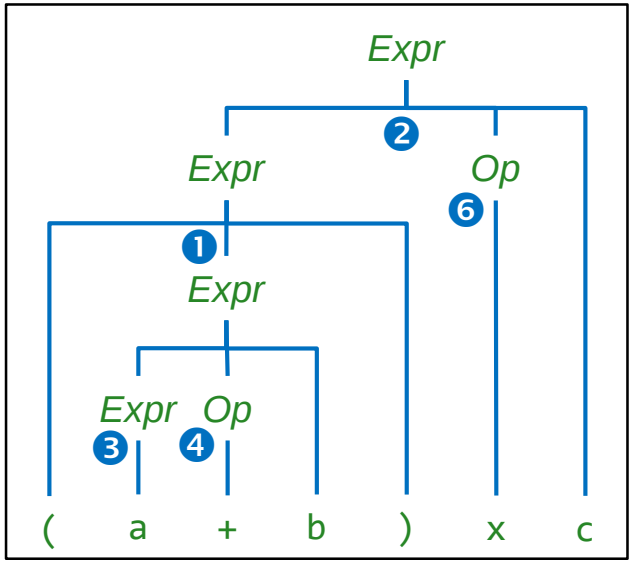
1	*Expr*	→	(*Expr*)
2		|	*Expr Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

- The task of finding a parse tree starts with
  - Start symbol
  - Text to be parsed
- Two fundamental ways to detect the structure in between

Top-down



Bottom-up



PREDICTIVE PARSING: LOOK-AHEAD

- To be avoided: backtracking
  - Introduces complexity and overhead
- Wanted: the ability to “guess” correctly which rule to apply
  - Called *predictive parsing*
  - Achieved by looking at first symbol(s) in token stream: *look-ahead*
- Consider top-down parsing of `1 - 3 + 4 - 2`
  - Right-recursive grammar

    Expr → `Operand` + *Expr* | `Operand` – *Expr* | `Operand`
    - No good: all rules apply on first symbol
    - Refactor to

      Expr → `Operand` *Next* ❶
      Next → + *Expr* ❷ | – *Expr* ❸ | ϵ ❹
    - This works: apply rules ❶ ❸ ❶ ❷ ❶ ❸ ❶ ❹

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

PREDICTIVE PARSING: LOOK-AHEAD

- To be avoided: backtracking
  - Introduces complexity and overhead
- Wanted: the ability to "guess" correctly which rule to apply
  - Called *predictive parsing*
  - Achieved by looking at first symbol(s) in token stream: *look-ahead*
- Consider top-down parsing of `1 - 3 + 4 - 2`
  - Left-recursive grammar

    Expr → *Expr* `+ Operand` | *Expr* `– Operand` | `Operand`
    - No good: how deep is the tree?

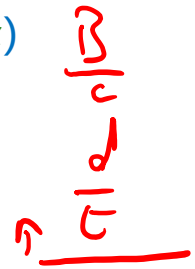
Expr → *Expr* `+ Operand` | *Expr* `- Operand` | `Operand`

Expr → Operand Expr'

Expr' → + Expr | – Expr | ε

LL(1): LEFT-TO-RIGHT, LEFTMOST, LOOKAHEAD 1

- At every state, the parser has
  - A stack of instantiated rules
    - Each element of the stack is the remaining part of a RHS
    - Initially, the stack consists only of the start symbol
  - A partially built parse tree
- Parser algorithm
  1. Remove the first symbol α from the top element of the stack
     - If $\alpha \in T$, it should be the next token on the input, otherwise report error
     - If $\alpha \in NT$, we need to know which RHS of α to choose
       - Based on the next token in the input (lookahead!)
       - Let's say $\alpha \rightarrow \beta$ with $\beta = \beta_1 \beta_2 \cdots \beta_n$
       - Put $\beta_1 \beta_2 \cdots \beta_n$ on the stack and in the parse tree (as children of α)
  2. Pop the stack as long as the top element is the empty string ϵ
  3. If the stack is empty, we're done



LL(1): LEFT-TO-RIGHT, LEFTMOST, LOOKAHEAD 1

- Parser algorithm
  1. Remove the first symbol α from the top element of the stack
     - If $\alpha \in T$, it should be the next token on the input, otherwise report error
     - If $\alpha \in NT$, we need to know which RHS of α to choose
       - Based on the next token in the input (lookahead!)
       - Let's say $\alpha \to \beta$ with $\beta = \beta_1\beta_2 \cdots \beta_n$
       - Put $\beta_1\beta_2 \cdots \beta_n$ on the stack and in the parse tree (as children of α)
  2. Pop the stack as long as the top element is the empty string ϵ
  3. If the stack is empty, we're done
- Consider parsing of `1 - 3 + 4`

Expr → `Operand` *Next* ❶
Next → `+` *Expr* ❷
| `-` *Expr* ❸
| ϵ ❹

LL(1): LEFT-TO-RIGHT, LEFTMOST, LOOKAHEAD 1

- Top-down parsing with lookahead 1
  - When is a grammar suitable for this?
  - Recall: In a CFG, rules are of the form $A \rightarrow \beta$ with $A \in NT$ and $\beta \in (NT \cup T)^*$
- Algorithm: compute *FIRST*, *FOLLOW*, *FIRST+*
  - *FIRST*: $(NT \cup T)^* \rightarrow 2^T$ (function from sentential forms to sets of terminals)
    - *FIRST*(β) yields *initial* terminals with which a β-derivation may start
    - In particular applied to nonterminals (*FIRST*(*A*))
  - *FOLLOW*: $NT \rightarrow 2^T$ (function from non-terminals to sets of terminals)
    - *FOLLOW*(*A*) yields terminals that may *follow A* in a derivable sentential form
  - *FIRST+*: *Rule* $\rightarrow 2^T$ (function from grammar rules to sets of terminals)
    - *FIRST+*($A \rightarrow \beta$) yields *initial* terminals of *A*-derivations starting with this rule
- LL(1) criterion
  - *FIRST+*($A \rightarrow \beta_1$) and *FIRST+*($A \rightarrow \beta_2$) should be disjoint for distinct β_1 and β_2
  - If they overlap on terminal a, we wouldn't know what to do when finding a

→ use for "based on the next token in the input (lookahead!)"

EXAMPLE COMPUTATION OF LL(1) CRITERION

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr′*
2	*Expr′*	→	+ *Term Expr′*
3		|	- *Term Expr′*
4		|	ϵ
5	*Term*	→	*Factor Term′*
6	*Term′*	→	x *Factor Term′*
7		|	÷ *Factor Term′*
8		|	ϵ
9	*Factor*	→	(*Expr*)
10		|	num
11		|	name

FIRST for terminal: terminal itself

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST	num	name	+	-	×	÷	(	)	eof	ϵ

Special "fake" symbols

FIRST for non-terminal:

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FIRST	(, num, name	+, -, ϵ	(, num, name	x, ÷, ϵ	(, num, name

FOLLOW: never ϵ

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FOLLOW					

	Production	FIRST set	FIRST$^+$ set
4	*Expr′* → ϵ		
8	*Term′* → ϵ		

FIRST+ equals *FIRST* except if ϵ is in *FIRST*; then take union with *FOLLOW*

EXAMPLE COMPUTATION OF LL(1) CRITERION

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

FIRST for terminal: terminal itself

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST										

FIRST for non-terminal:

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FIRST					

FOLLOW: never ϵ

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FOLLOW					

	Production	FIRST set	FIRST$^+$ set
4	*Expr′* → ϵ		
8	*Term′* → ϵ		

FIRST+ equals *FIRST* except if ϵ is in *FIRST*; then take union with *FOLLOW*