COMP 442 / 6421 Compiler Design

Tutorial 4
Abstract Syntax Tree Generation and Syntax Directed Translation

Instructor: Dr. Joey Paquet paquet@cse.concordia.ca
TAs: Haotao Lai h_lai@encs.concordia.ca

Lab Instructor

Section: lab hours NNK M------ 20:30-22:20 H819

Name: Haotao Lai (Eric)

Office: EV 8.241

Email: h_lai@encs.concordia

Website: http://laihaotao.me/ta

Concrete Parse Tree vs Abstract Syntax Tree

A parse tree pictorially shows how the start symbol of a grammar derives a string in the language.

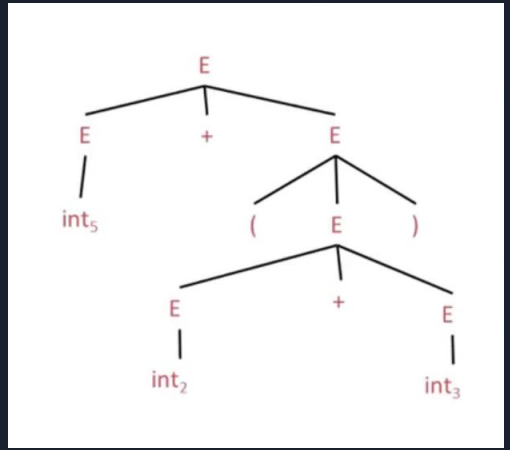
Whereas,

Abstract syntax trees, or simply syntax trees, differ from parse trees because superficial distinctions of form, unimportant for translation, do not appear in syntax trees.

Also, they're most often expressed by the data structures of the language used for implementation. [1]

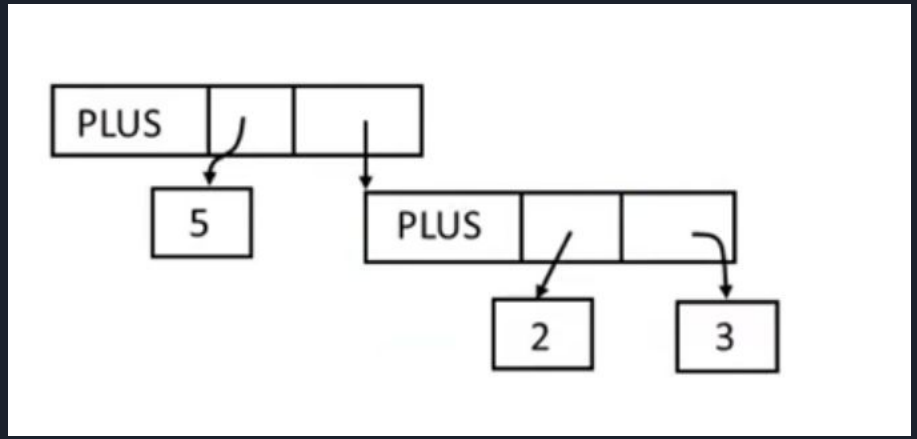
Concrete Parse Tree vs Abstract Syntax Tree

- AST's ingnore some details as compared to Parse Trees.
- For example:
- E → int | (E) | E+ E and a String 5 + (2+3)
- After lexical analysis we get, int_5 '+' '(' int_2 '+' int_3 ')'
- But it does contain too much information.



Concrete Parse Tree vs Abstract Syntax Tree

- E → int | (E) | E+ E and a String 5 + (2+3)



How to construct an AST

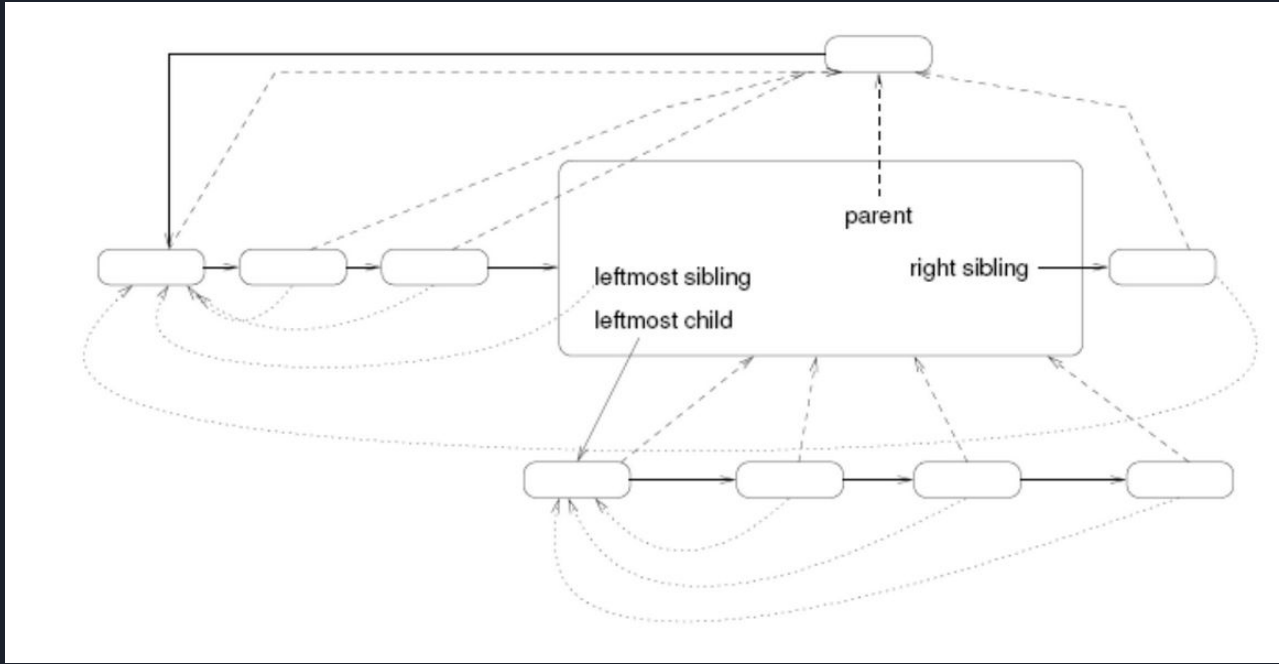
The AST structure is constructed bottom-up where,

- A list of siblings is generated and
- The list is later adopted by a parent.

Each node needs connection to:

- **Parent**: to migrate information upwards in the tree
  - Link to parent
- **Siblings**: to iterate through (1) a list of operands or (2) members of a group, e,g, members of a class, or statements.
  - Link to right sibling (thus creating a single linked list of siblings)
  - Link to leftmost sibling (in case one needs to traverse the list as a sibling is being processed).
- **Children**: to generate/traverse the tree
  - Link to leftmost child (who represents the head of a linked list of children).

Data Structure Design



How to construct an AST

class AST

- \+ static makeNode( ) -> AstNode
- \+ static makeFamily( ) -> AstNode
- \+ root: AstNode

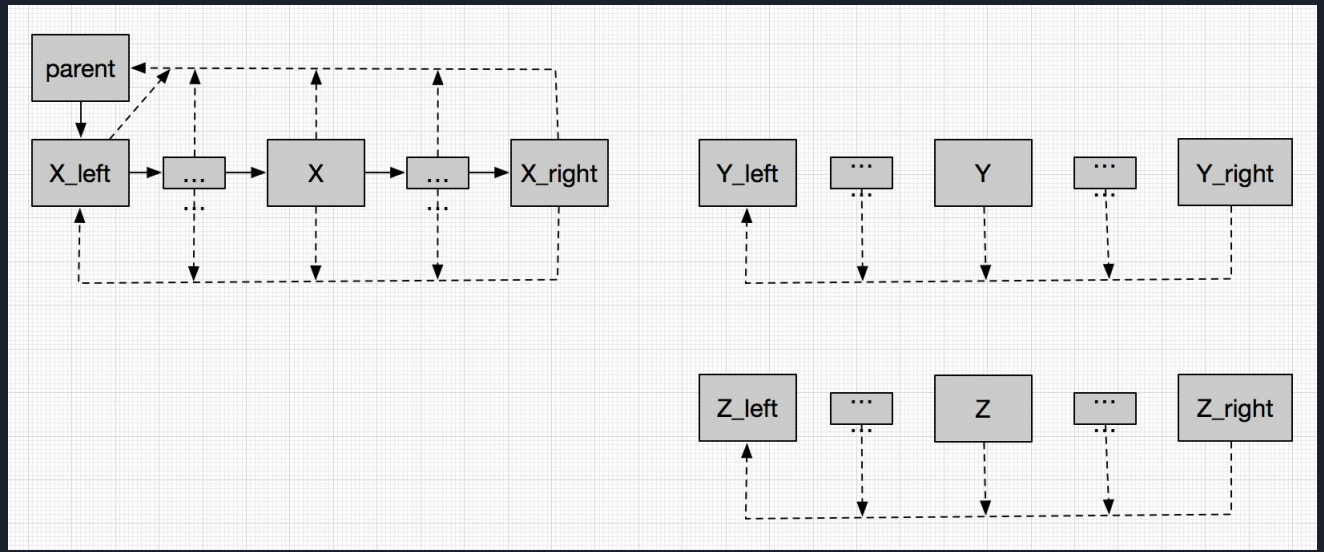
class AstNode

- \+ makeSibling( )
- \+ adoptChildren( )

** note: only the key fields and methods were shown here, you may need extra fields or methods*

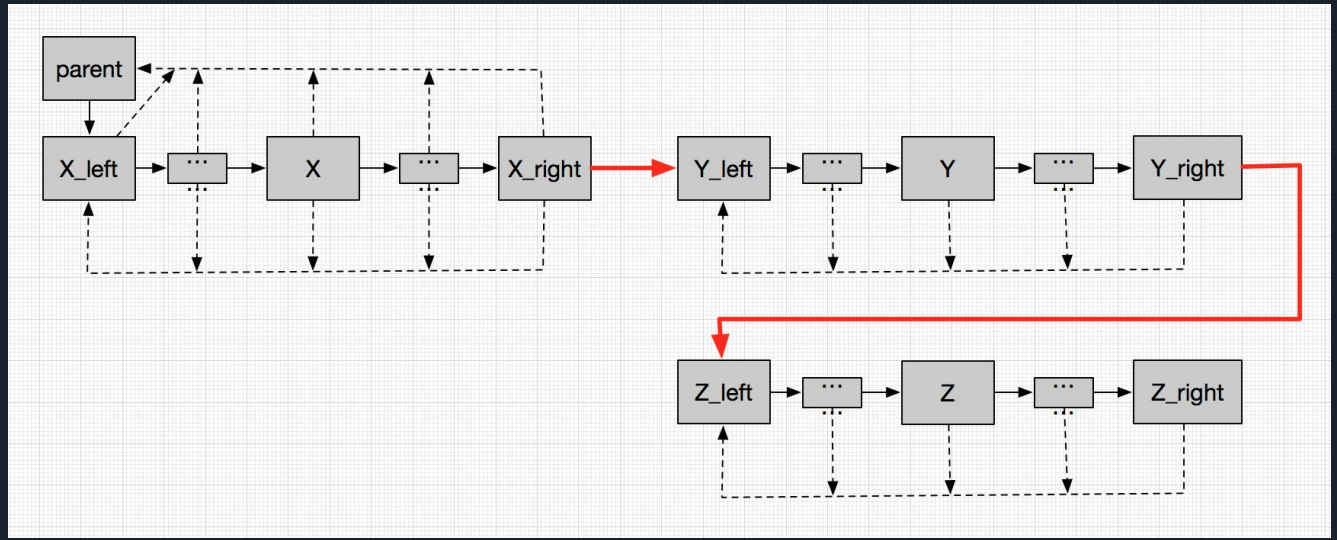
makeSibling( )

assume we are calling: X.makeSibling(Y, Z)



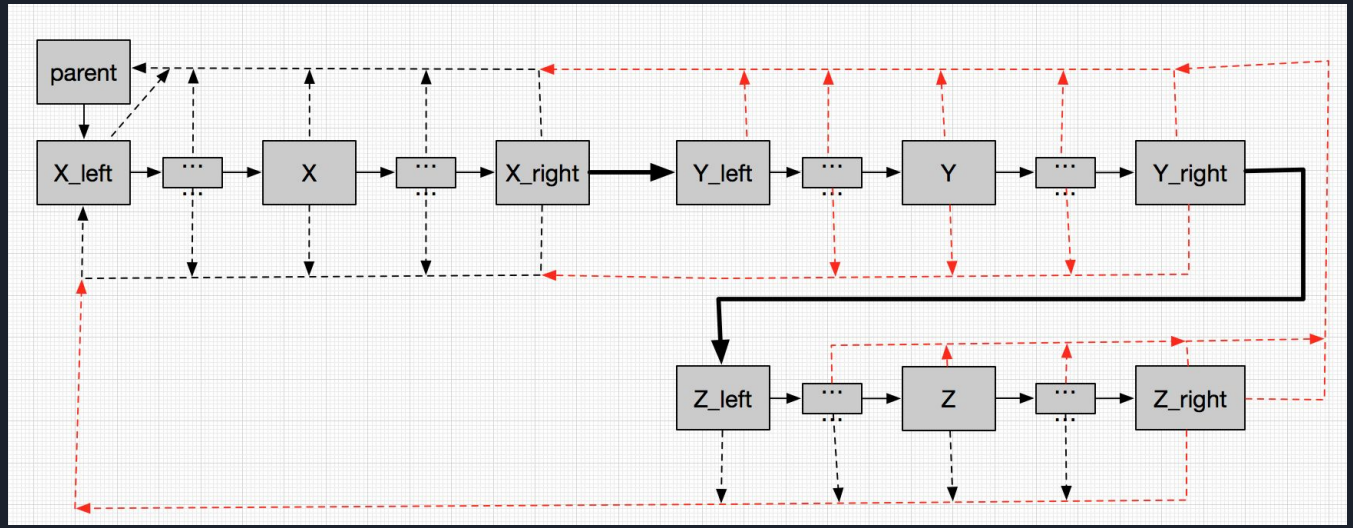
makeSibling( )

find X's rightmost sibling, link it with the Y's leftmost sibling, treat them as a whole and link to Z's leftmost sibling (repeat until no more new unlinked)



makeSibling( )

set all new added elements' parent to X's parent and all their leftmost sibling pointer to the X's leftmost sibling



adpotChildren( )

assume we are calling X.adoptChildren(Y)

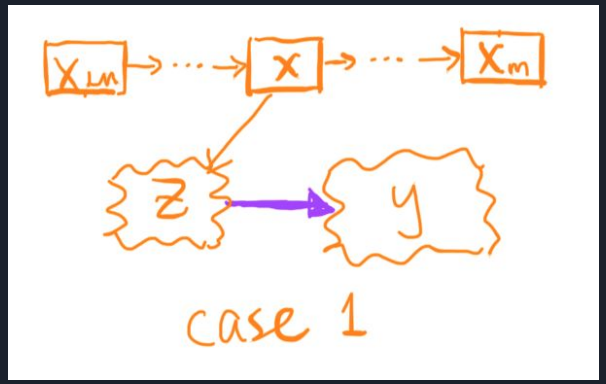
adpotChildren( )

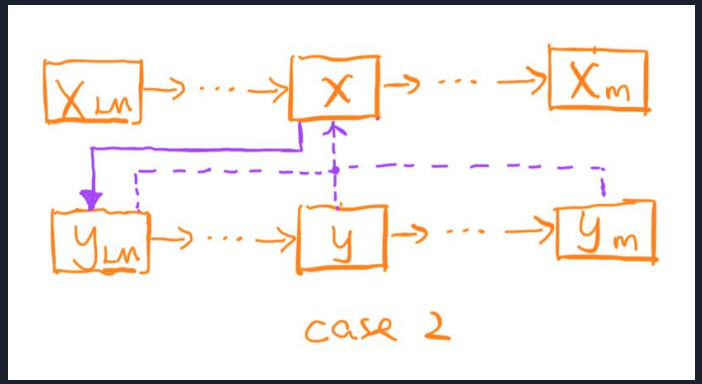
There will be two possibilities:

1. X already has a leftmost child
2. X don't have any leftmost child

adpotChildren( )

1. just link Z to Y, do what we do in makeSibling( )
2. link new element's parent to X and set X's leftmost child pointer to Y's leftmost sibling



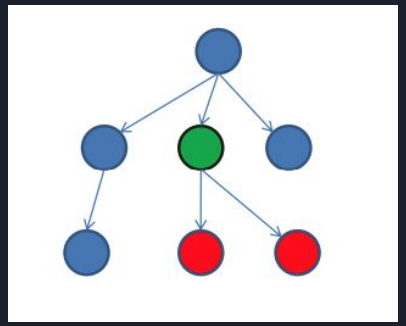


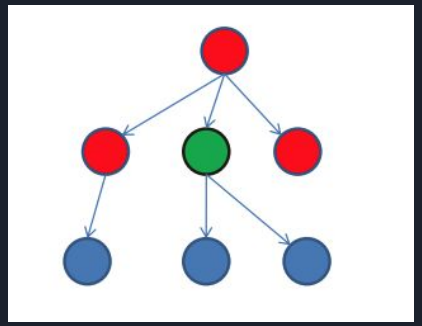
Syntax Directed Translation (SDT)

Attributes

Synthesized attributes at node N are defined only in terms of the attribute values of the children of N, and N itself. (left image shown below)

Inherited attributes at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings. (right image shown below)



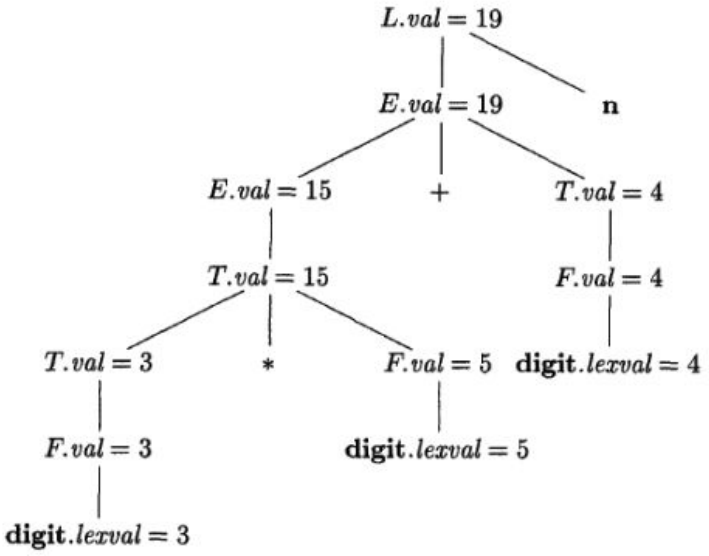


Example

Can you see what's the problem here?

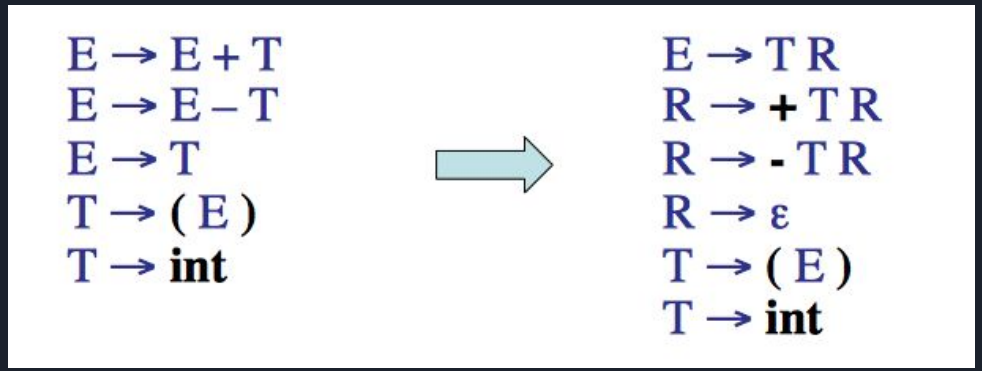
	PRODUCTION	SEMANTIC RULES
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

3*5+4n

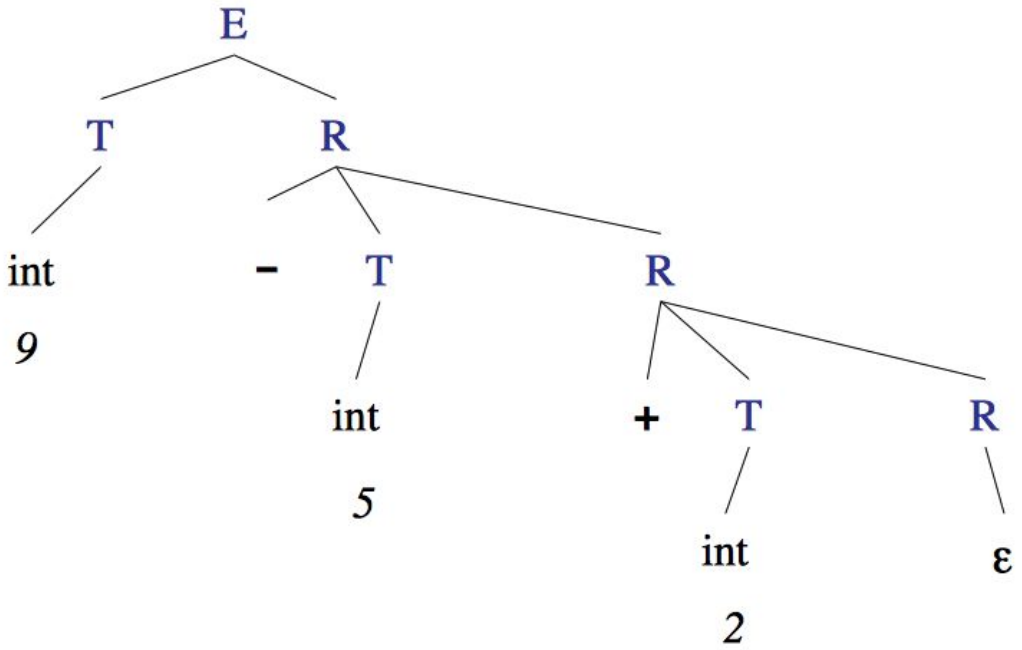


Topdown Translation Example

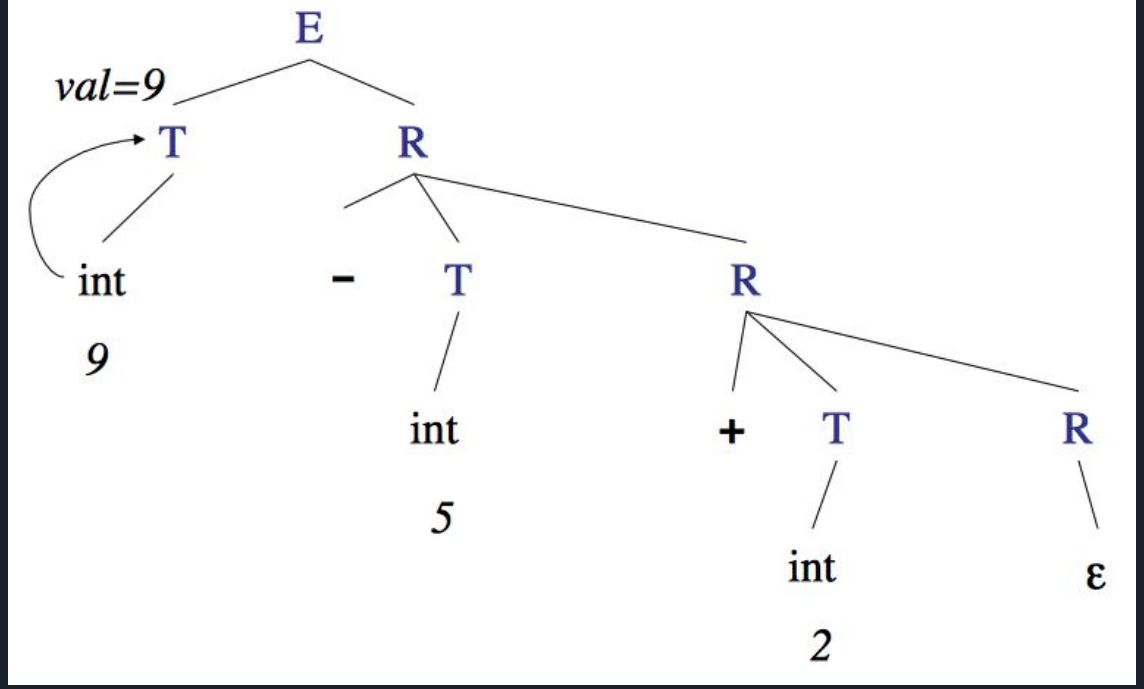
Remove left recursion from the grammar in the left, we end up with the new grammar in the right which can be used by a top down parser.



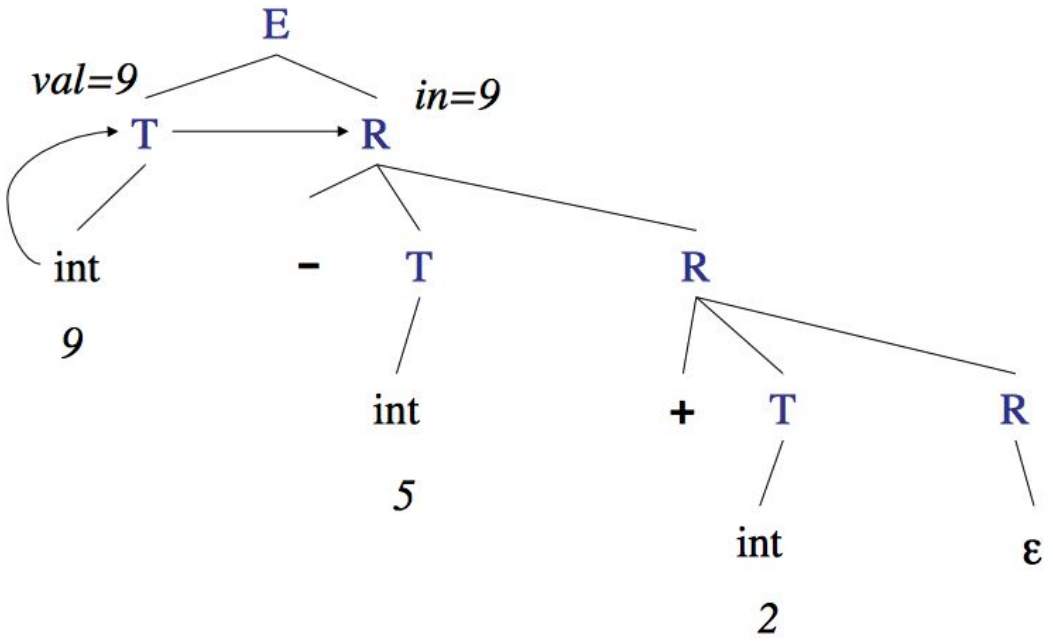
input: 9 - 5 + 2



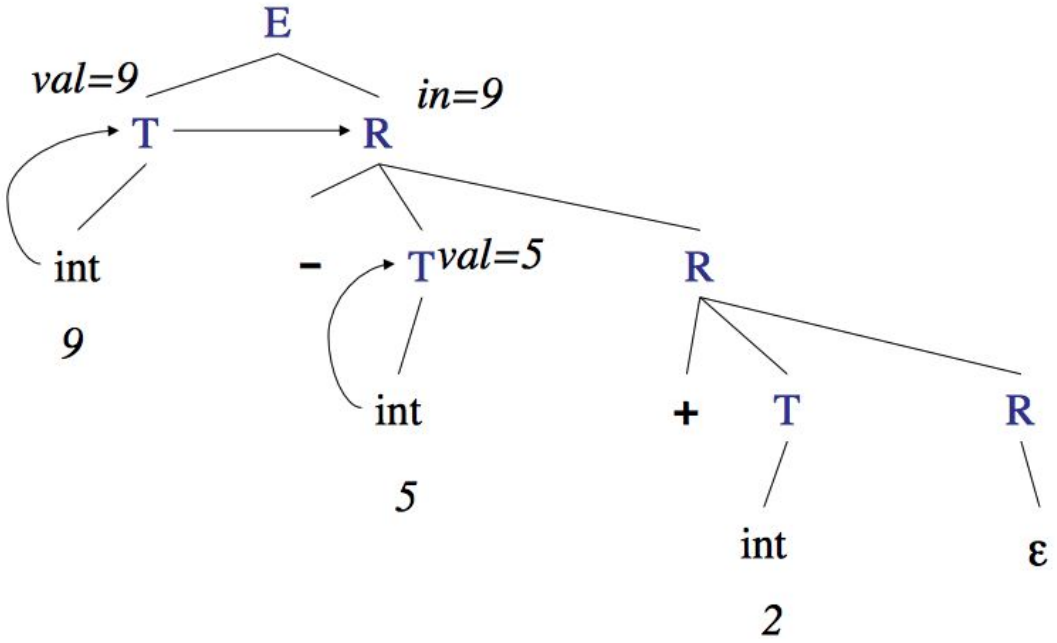
input: 9 - 5 + 2



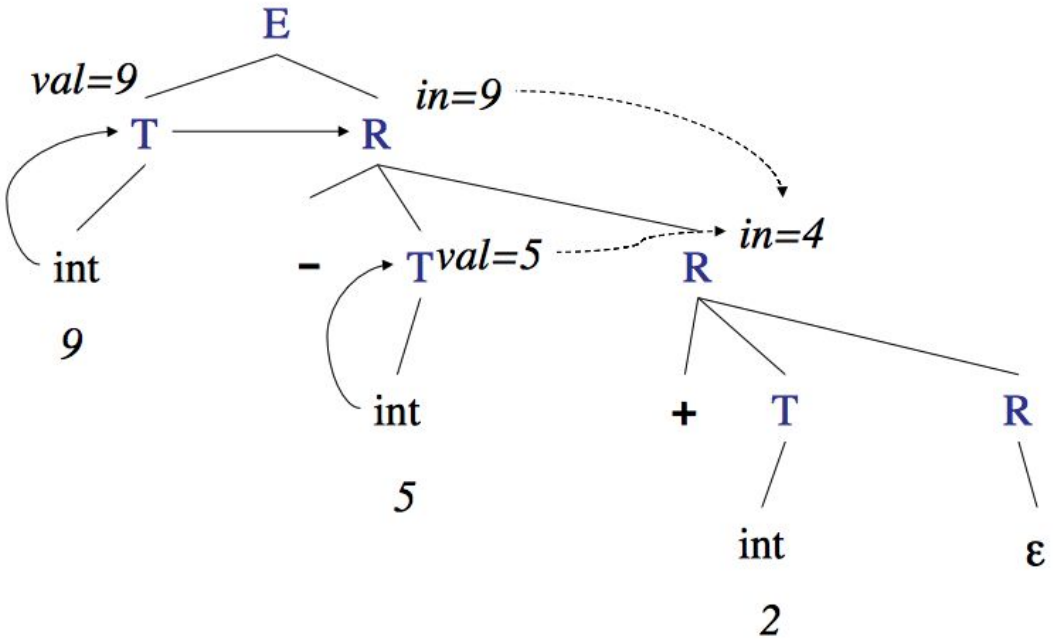
input: 9 - 5 + 2



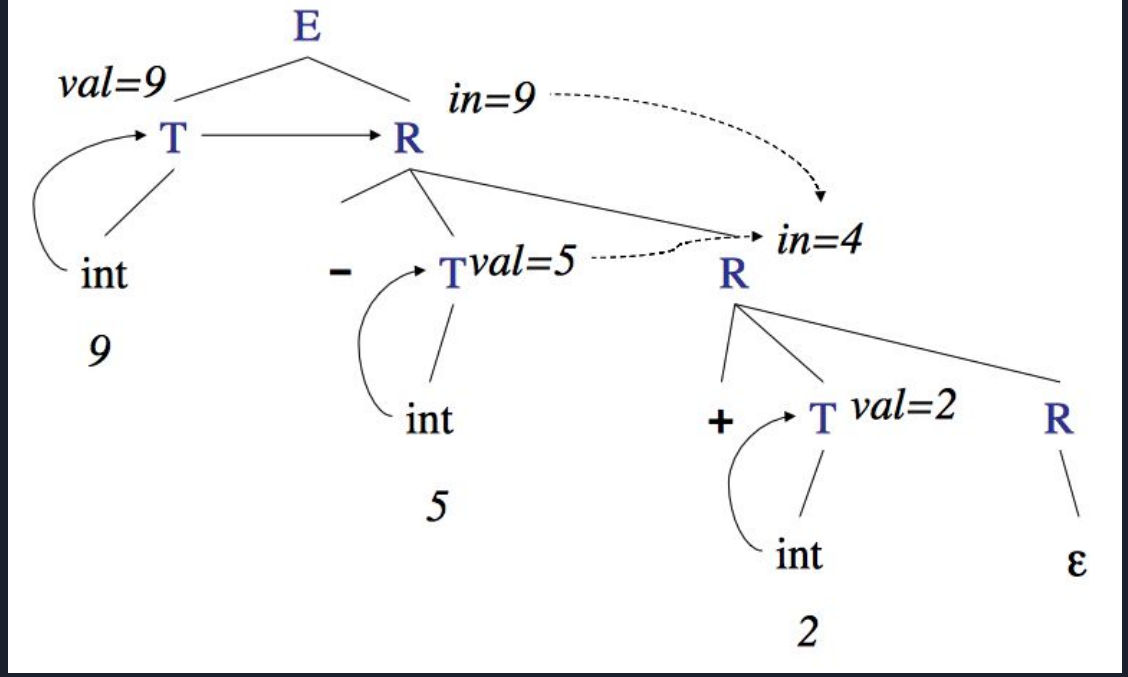
input: 9 - 5 + 2



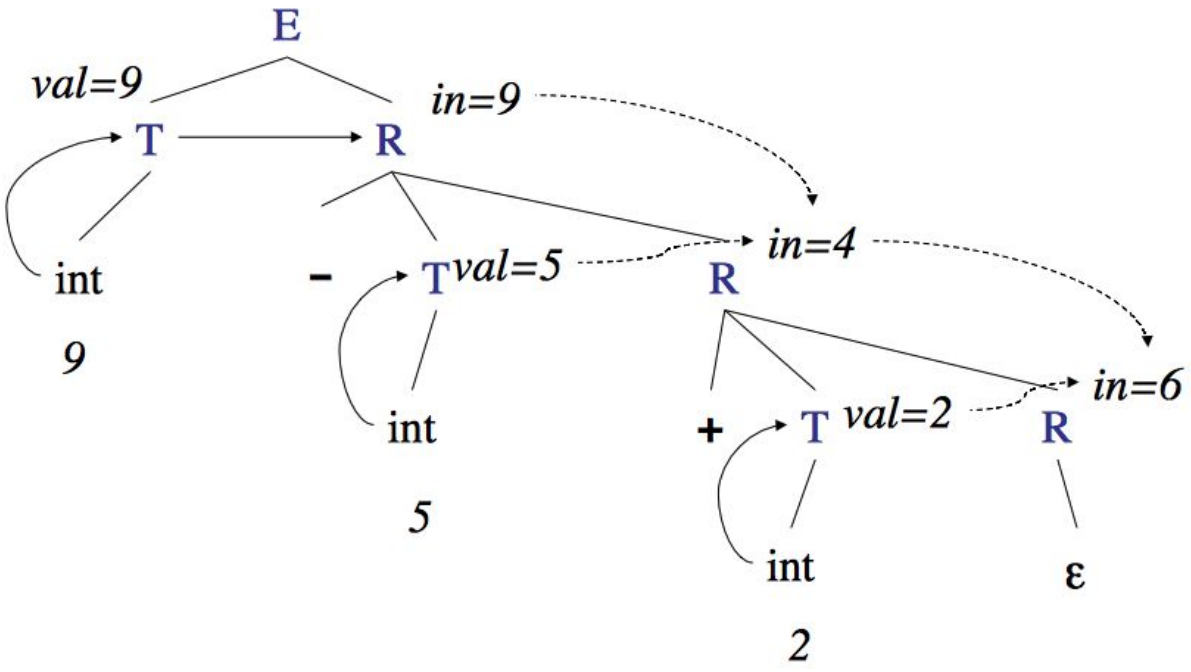
input: 9 - 5 + 2



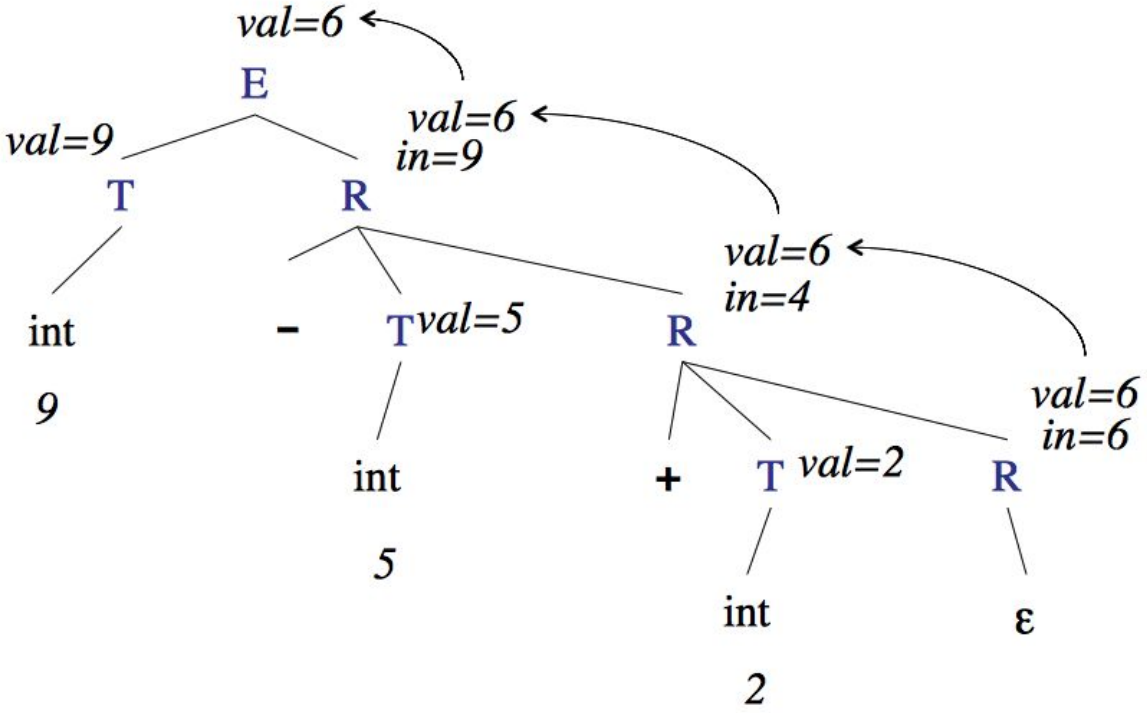
input: 9 - 5 + 2



input: 9 - 5 + 2



input: 9 - 5 + 2


val=6
E
val=9
T
R
val=6
in=9
int
9
-
T
val=5
int
5
R
val=6
in=4
+
T
val=2
int
2
R
val=6
in=6
ε

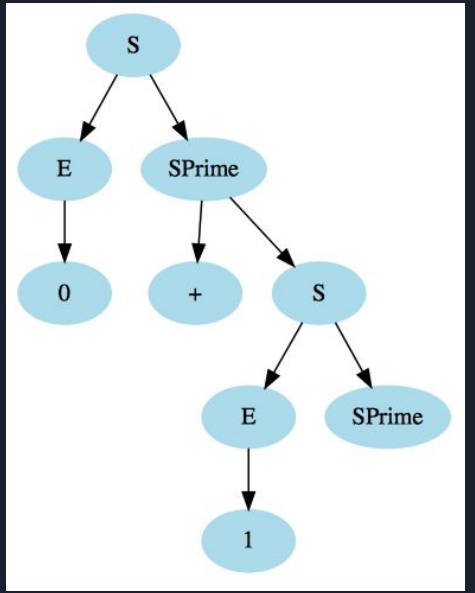
Another Example

Try to do it by yourself

Grammar:

S → E S'
S' → ε | '+' S
E → '0' | '1' | '('S')'

Input: 0 + 1



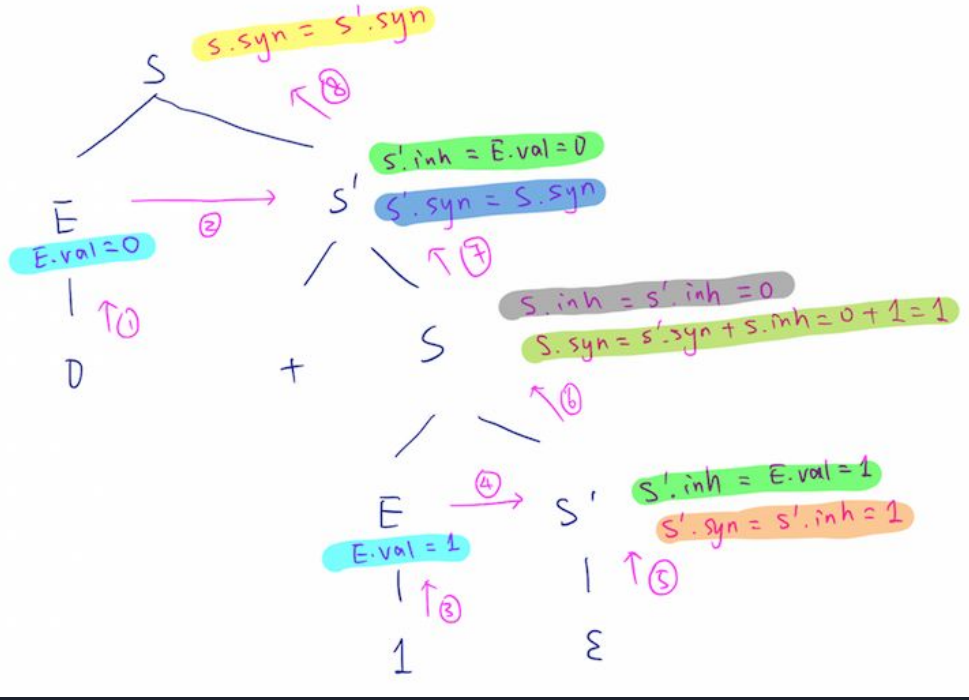
$S \rightarrow E\,S'$ $\quad S'.inh = E.val$; $S.syn = S'.syn$;

$S' \rightarrow \varepsilon$ $\quad S'.syn = S'.inh$

$S' \rightarrow +S$ $\quad S.inh = S'.inh$; $S.syn = S.inh + S'.syn$; $S'.syn = S.syn$

$E \rightarrow 0 \mid 1$ $\quad E.val = lexval$

$E \rightarrow (S)$



References

[1] Compilers: Principles, Techniques, and Tools" by Aho, Sethi and Ullman

[2] https://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees/

[3] C.N. Fischer, R.K. Cytron, R.J. LeBlanc Jr., Crafting a Compiler, Adison-Wesley, 2009. Chapter 7.

[4] https://cs.nyu.edu/courses/spring11/G22.2130-001/lecture8.pdf