DVA339 HT18 - LECTURE 2

grammars and parsing

ACADEMIC HONESTY

Plagiarism and other forms of academic misconduct

- make sure you understand what is not allowed and why
- see, e.g., Academic Honesty and Integrity at Chalmers

Don'ts

- you are not allowed to cooperate on the written assignments
- you are not allowed to plagiarize or *paraphrase* other peoples work (including the slides for you presentation!)
- You are not allowed to copy otherwise make use of other peoples code.

By historic necessity

- all handed in material will be subjected to automatic plagiarism control!

WORD OF WARNING

Do no publish your code on the net!

- it's not prohibited, but …

… if someone else copies your code you may be subject to disciplinary action, where you have to prove authorship of the code

- quite unnecessary stress even if you did nothing wrong!

I understand the will to share and the concept of freedom of information

- but this is not the best way to support it.

BONUS GRADE

If you pass all labs before the written exam (January 17 2019) your final grade on the course will be one higher than the grade on the written exam.

- given that you passed the exam.
- i.e., the additional grade will not pass you on the course if you failed the written exam

Note that the bonus grade only applies to the first exam attempt, and not re-exams.

LAST TIME

Lexical analysis

- string of characters to sequence of tokens

Lexical tokens, token types

- identifiers, keywords, operators, separators, numbers, …

Specifying token types

- regular expressions

Lab 1.1- how is it going?

TODAY

Context-free grammars

- derivations, derivation trees

Introduction to parsing

- depth first search

Ambiguity

Rewriting grammars

- associativity
- precedence
- left factoring

Abstract syntax

AN EXAMPLE GRAMMAR

Non-terminals
- S, L, E (in capital letters)
- sometimes called syntactic categories

Terminals
- corresponding to the tokens
- *id*, *num* – token types (in italics)
- **; := + ( ) , print** – lexemes (in bold)

Start non-terminal
- `S`

Production rules (selected)
- $\mathtt{S} \to \mathtt{S;\ S}$
- $\mathtt{S} \to \mathtt{id\ :=\ E}$
- written $\mathtt{S} \to \mathtt{S;\ S} \mid \mathtt{id\ :=\ E}$

$\mathtt{S} \to \mathtt{S}\ \mathbf{;}\ \mathtt{S}$

$\mathtt{S} \to \mathit{id}\ \mathbf{:=}\ \mathtt{E}$

$\mathtt{S} \to \mathbf{print}\ \mathbf{(}\ \mathtt{L}\ \mathbf{)}$

$\mathtt{E} \to \mathit{id}$

$\mathtt{E} \to \mathit{num}$

$\mathtt{E} \to \mathtt{E}\ \mathbf{+}\ \mathtt{E}$

$\mathtt{E} \to \mathbf{(}\ \mathtt{S}\mathbf{,}\ \mathtt{E}\ \mathbf{)}$

$\mathtt{L} \to \mathtt{E}$

$\mathtt{L} \to \mathtt{L}\mathbf{,}\ \mathtt{E}$

DERIVATIONS

A derivation

- starts in the start non-terminal (the start symbol)
- in this case S
- proceeds by replacing one non-terminal according to one of the possible production rules
- until no more non-terminals exist

(1) $S \rightarrow S;\ S$
(2) $S \rightarrow id := E$
(3) $S \rightarrow$ **print** $(\ L\ )$
(4) $E \rightarrow id$
(5) $E \rightarrow num$
(6) $E \rightarrow E + E$
(7) $E \rightarrow (\ S,\ E\ )$
(8) $L \rightarrow E$
(9) $L \rightarrow L,\ E$

$$S \Rightarrow_1 S;S \Rightarrow_2 x := E;\ S \Rightarrow_5 x := 23;\ S \Rightarrow_3$$
$$x := 23;\ \texttt{print}\ (\ L\ ) \Rightarrow_8 x := 23;\ \texttt{print}\ (\ E\ ) \Rightarrow_4$$
$$x := 23;\ \texttt{print}\ (\ x\ )$$

EXERCISE

Derive `x := 3; y := (z := 5, x + z)`

(1) S → S; S
(2) S → *id* **:=** E
(3) S → **print** **(** L **)**
(4) E → *id*
(5) E → *num*
(6) E → E **+** E
(7) E → **(** S**,** E **)**
(8) L → E
(9) L → L**,** E

Is the derivation unique?

DERIVATIONS

When a production rule has more than one non-terminal on the right hand side

- $\mathtt{S} \rightarrow \mathtt{S;\ S}$

Derivation can continue on either

$$\underline{\mathtt{S}} \Rightarrow_{1} \underline{\mathtt{S}}\mathtt{;S} \Rightarrow_{2,5} \mathtt{x:=1;}\ \underline{\mathtt{S}} \Rightarrow_{2,5} \mathtt{x:=1;\ y:=2}$$

$$\underline{\mathtt{S}} \Rightarrow_{1} \mathtt{S;}\underline{\mathtt{S}} \Rightarrow_{2,5} \underline{\mathtt{S}}\mathtt{;\ y:=2} \Rightarrow_{2,5} \mathtt{x:=1;\ y:=2}$$

Does it matter?

LEFT AND RIGHT DERIVATIONS

When the left-most non-terminal is always selected we have a left derivation

$\texttt{L:}\ \underline{\texttt{S}} \Rightarrow_1 \underline{\texttt{S}}\texttt{;S} \Rightarrow_{2,5} \texttt{x:=1;}\ \underline{\texttt{S}} \Rightarrow_{2,5} \texttt{x:=1; y:=2}$

When the right-most non-terminal is always selected we have a right derivation

$\texttt{R:}\ \underline{\texttt{S}} \Rightarrow_1 \texttt{S;}\underline{\texttt{S}} \Rightarrow_{2,5} \underline{\texttt{S}}\texttt{; y:=2} \Rightarrow_{2,5} \texttt{x:=1; y:=2}$

There will always be a right and a left derivation

- sometimes they coincide (when?)
- are there more possible derivations?

ORDER OF REWRITING

Some types differences in order of rewriting are unimportant

- yet give rise to different derivations
- $\mathtt{L:}\ \underline{\mathtt{S}} \Rightarrow_{1} \underline{\mathtt{S}}\mathtt{;S} \Rightarrow_{2,5} \mathtt{x:=1;}\ \underline{\mathtt{S}} \Rightarrow_{2,5} \mathtt{x:=1;\ y:=2}$
- $\mathtt{R:}\ \underline{\mathtt{S}} \Rightarrow_{1} \mathtt{S;}\underline{\mathtt{S}} \Rightarrow_{2,5} \underline{\mathtt{S}}\mathtt{;\ y:=2} \Rightarrow_{2,5} \mathtt{x:=1;\ y:=2}$

Derivations are linear, and force a *total order* on rewriting steps

How can we relax this?

- *partial order* necessary, certain rewriting steps must occur before other
- other rewriting steps can occur in either order (or in parallel)

What decides this?

INDUCED ORDER

The production rules induce an order

For context-free grammars, grammars with

- a single non-terminal on the left hand side of production rules
- an arbitrary number of terminals and non-terminals on the right hand side of production rules

we have that

- all non-terminals on the right hand side of a rule can be rewritten in any order (or in parallel)
- the non-terminal on the left hand side of the rule must be replaced before the non-terminals introduced by the rewriting (duh!)

For production rule

- $\texttt{S} \rightarrow \texttt{S; S}$

consider the derivation

- $\texttt{S}^1 \Rightarrow_1 \texttt{S}^2\texttt{;}\ \texttt{S}^2 \Rightarrow_1 \texttt{S}^2\texttt{;}\ \texttt{S}^3\texttt{;}\ \texttt{S}^3$

where superscripts denote necessary order

DERIVATION TREES

For production rule

- $S \rightarrow S;\ S$

consider the derivation

- $S^1 \Rightarrow_1 S^2;\ S^2 \Rightarrow_1 S^2;\ S^3;\ S^3$

A more reasonable representation is a tree

- the level in the tree represent the necessary order

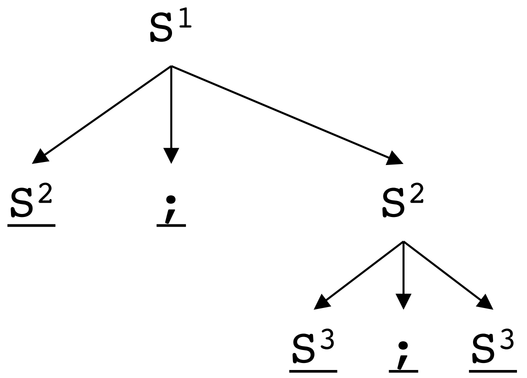
Rewriting a non-terminal with a production rule

- adds the right hand side of the rule as children to the non-terminal

As an added bonus it is clearer

- which non-terminal is rewritten with which production rule

The leaves correspond to the final rewriting product



EXERCISE

Create the derivation tree for `x:=1; y:=2`

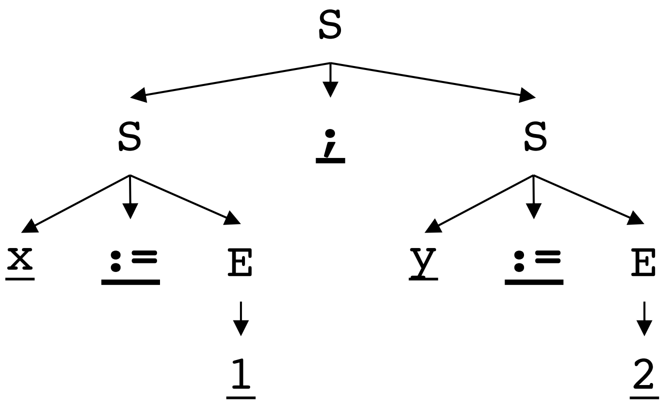
- is it unique?

(1) S → S; S
(2) S → *id* **:=** E
(3) S → **print** **(** L **)**
(4) E → *id*
(5) E → *num*
(6) E → E **+** E
(7) E → **(** S, E **)**
(8) L → E
(9) L → L, E

EXERCISE

Create the derivation tree for `x:=1; y:=2`

- is it unique?
- Yes



(1) S → S; S
(2) S → *id* **:=** E
(3) S → **print (L)**
(4) E → *id*
(5) E → *num*
(6) E → E **+** E
(7) E → **(** S**,** E **)**
(8) L → E
(9) L → L**,** E

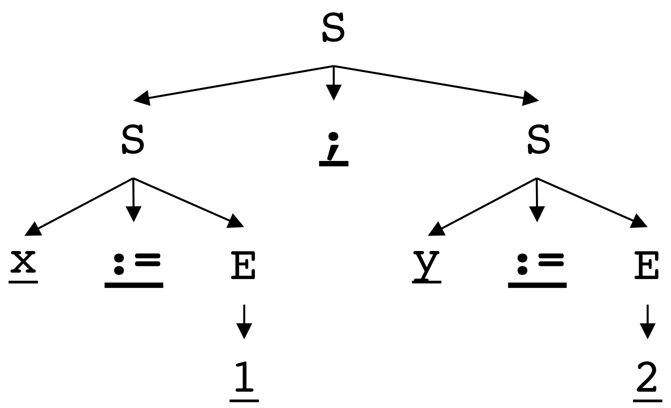
RELATION TO DERIVATIONS

Derivation tree for `x:=1; y:=2`

- result is given by left-to-right traversal of leaves

Left derivations correspond to left-to-right inorder traversal of tree

- $\underline{\texttt{S}} \Rightarrow_1 \underline{\texttt{S}}\texttt{; S} \Rightarrow_2 \texttt{x:=}\underline{\texttt{E}}\texttt{; S} \Rightarrow_5 \texttt{x:=1; }\underline{\texttt{S}} \Rightarrow_2 \texttt{x:=1; y:=}\underline{\texttt{E}} \Rightarrow_5 \texttt{x:=1; y:=2}$



Right derivations correspond to right-to-left inorder traversal of tree

- $\underline{\texttt{S}} \Rightarrow_1 \texttt{S; }\underline{\texttt{S}} \Rightarrow_2 \texttt{S; y:=}\underline{\texttt{E}} \Rightarrow_5 \underline{\texttt{S}}\texttt{; y:=2} \Rightarrow_2 \texttt{x:=}\underline{\texttt{E}}\texttt{; y:=2} \Rightarrow_5 \texttt{x:=1; y:=2}$

EXERCISE

Create the derivation tree for `x := (y := 1, y + 1)`

- How many derivations does it represent?

(1) S → S; S
(2) S → *id* **:=** E
(3) S → **print (L)**
(4) E → *id*
(5) E → *num*
(6) E → E **+** E
(7) E → **(** S**,** E **)**
(8) L → E
(9) L → L**,** E

INTRODUCTION TO PARSING

generating derivation trees

CREATION OF DERIVATION TREES

Given a program

- `1+2`

and a grammar

$$E \rightarrow T \mid T + E$$

$$T \rightarrow NUM \mid (E)$$

How can we construct a derivation tree corresponding to the program in the grammar?

TWO APPROACHES

Top down

Start in S, find derivation to program

- simpler to understand, intuitively corresponds to the derivation process
- slow in general (n^3)
- fast subsets exists (LL(1) – Lecture 3)
- LL(1) easy to implement by hand

Bottom up

Start with program, rewrite backwards to S

- harder to understand
- slow in general (n^3)
- fast subsets exists (LR – Lecture 4)
- very cumbersome to implement by hand
- more used, subsets more expressive than LL(1)

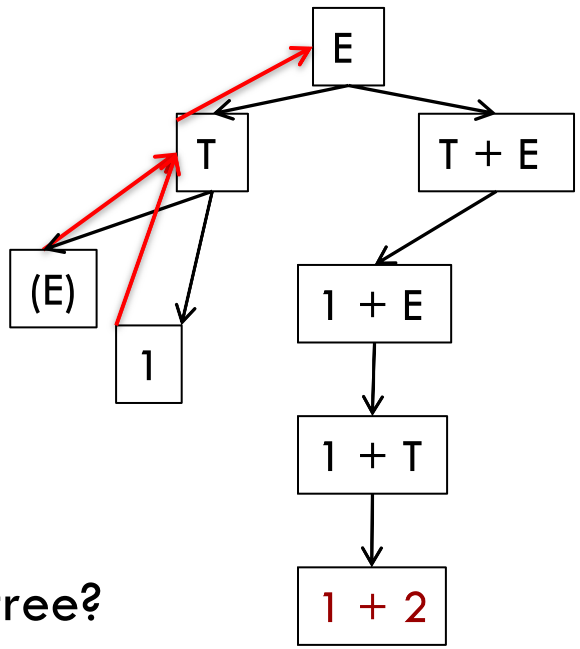
PARSING AS A SEARCH PROBLEM

Push E on stack

1. If stack is empty, fail
   - Let X be top of stack
2. if X is equal to input, done
3. if X is incompatible with input or if X does not contain any more non-terminals
   - backtrack (pop, and continue with 1)
4. Let A be left-most non-terminal in X
5. Select next unused rule P for A in X
   - if no more rules, backtrack
   - mark rule as used for A in X
6. Push result of applying P to A in X
7. Continue with 1

Try program 1 + 2 in grammar

$$E \to T \mid T + E$$
$$T \to num \mid (E)$$



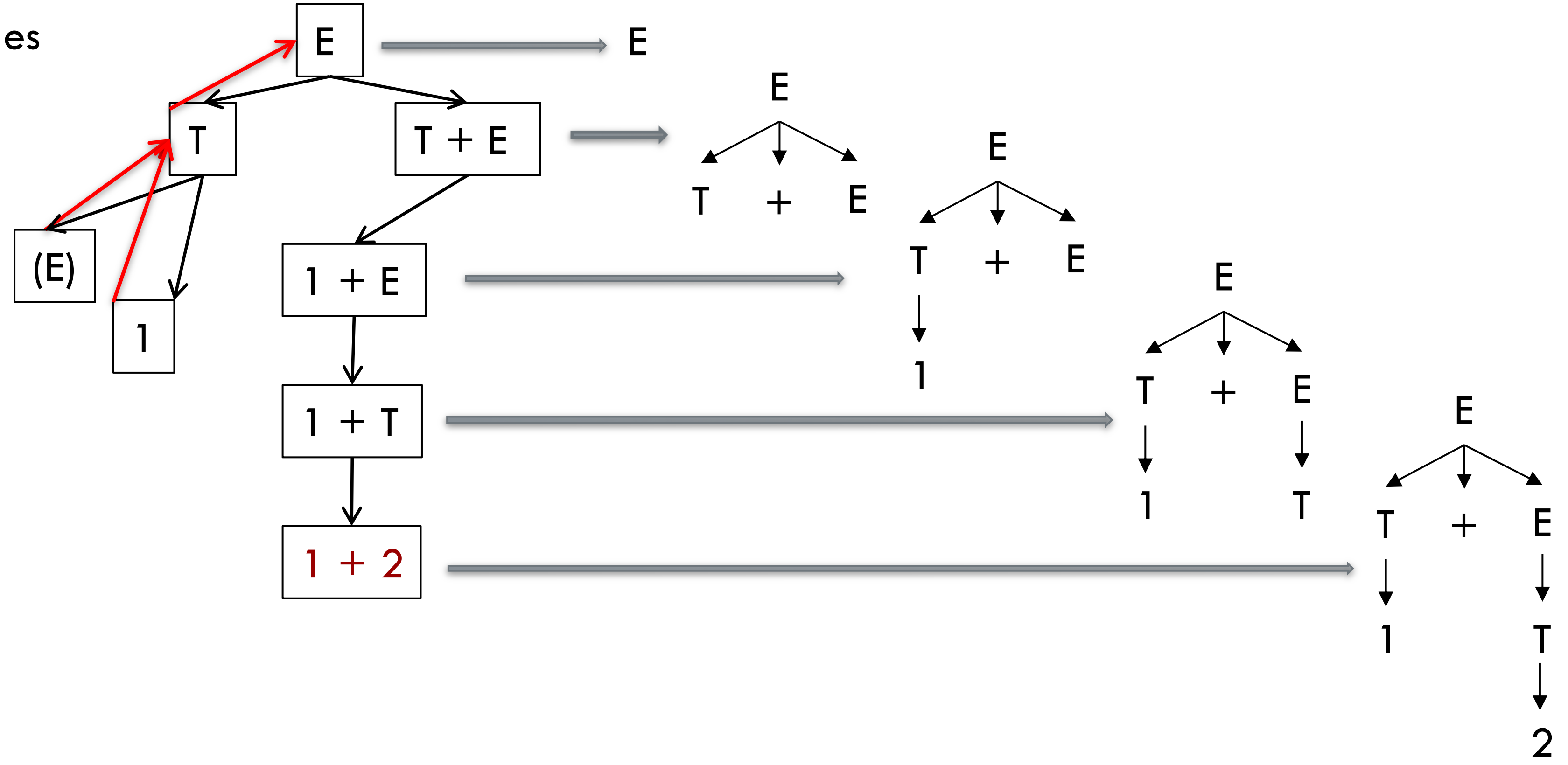
What is the resulting tree?

- this tree is a view of the search

WHAT IS THE DERIVATION TREE

The derivation tree lives on the stack

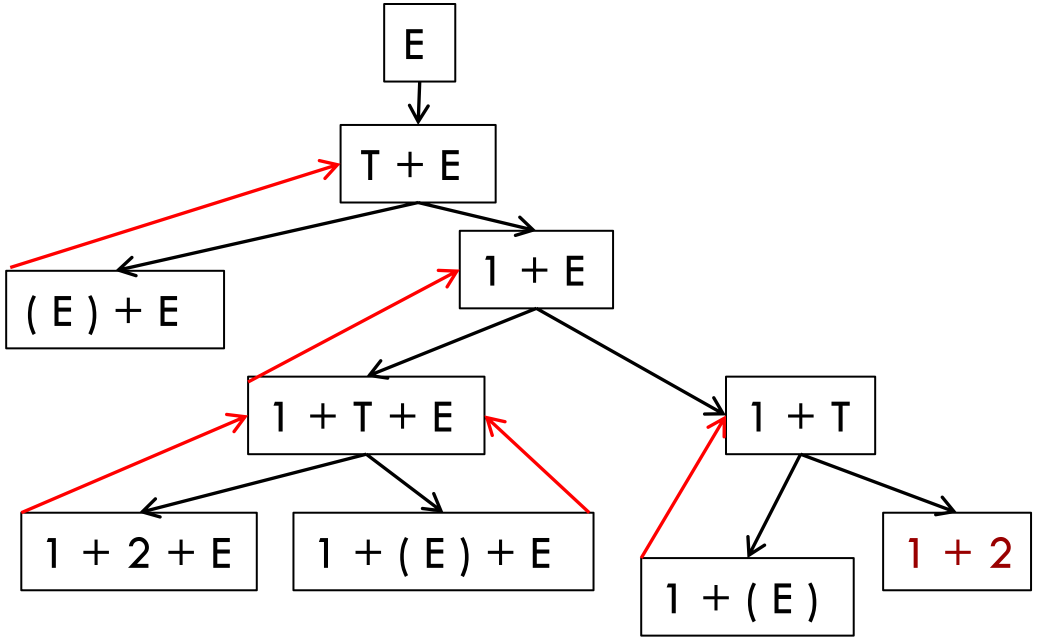
- updated by the selected rules



ORDER MATTERS

Consider a small rearrangement of the grammar

$$E \rightarrow T + E \mid T$$
$$T \rightarrow (E) \mid num$$



The resulting search tree is much bigger!

IS IT REALLY THIS SIMPLE?

Sadly not!

What about *left recursive* grammars?

- `A → Aa`

The left recursion does not have to be immediate, mutually left recursive is problematic too

- `A → Bb`
- `B → Cc`
- `C → Aa`

For any cycle length

- it's the presence of unproductive cycles that may be a problem.

LEFT RECURSION

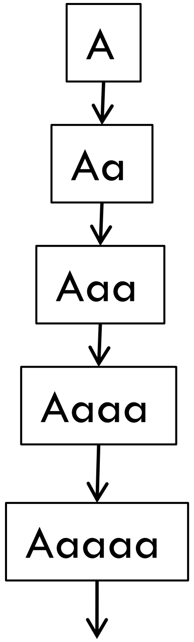
Naïve search may non-terminate on left-recursive grammars

- it's possible to enhance the algorithm with cycle detection
- not very efficient

Other solution – rewrite grammar to remove left recursion!

- today or next time depending on time

$A \rightarrow Aa \mid c$



COMPLEXITY

The algorithm presented can be seen as a variant of *depth first search*.

The worst case complexity is *exponential* in the length of the input.

- not that hard to construct grammars with this behavior (try!)

Not very practical

- in the current form
- but for some grammars it can be made very efficient!
- predictive recursive descent – lecture 3

AMBIGUITY

EXERCISE

Consider the grammar of expressions

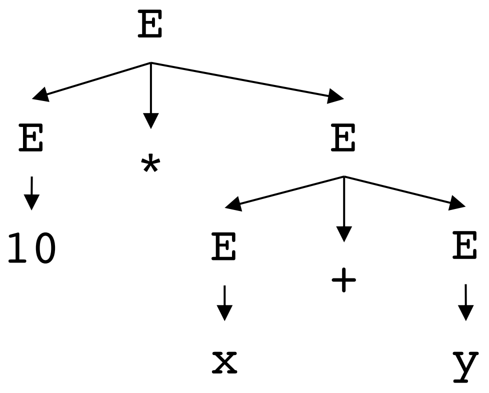
- create a derivation tree for `10 * x + y`

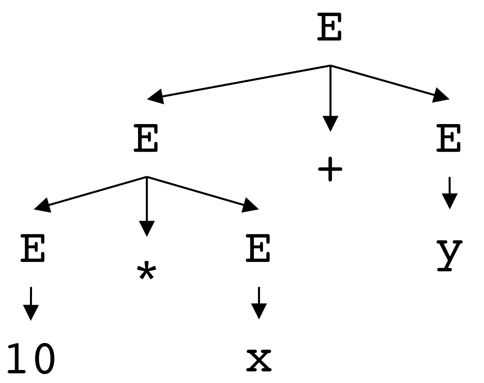
```
E → E + E
  | E * E
  | E - E
  | E / E
  | ( E )
  | id
  | num
```

EXERCISE

Consider the grammar of expressions

- create a derivation tree for `10 * x + y`






```
E → E + E
  | E * E
  | E - E
  | E / E
  | ( E )
  | id
  | num
```

One program, two derivation trees

- does it matter?

AMBIGUITY

One program, two derivation trees

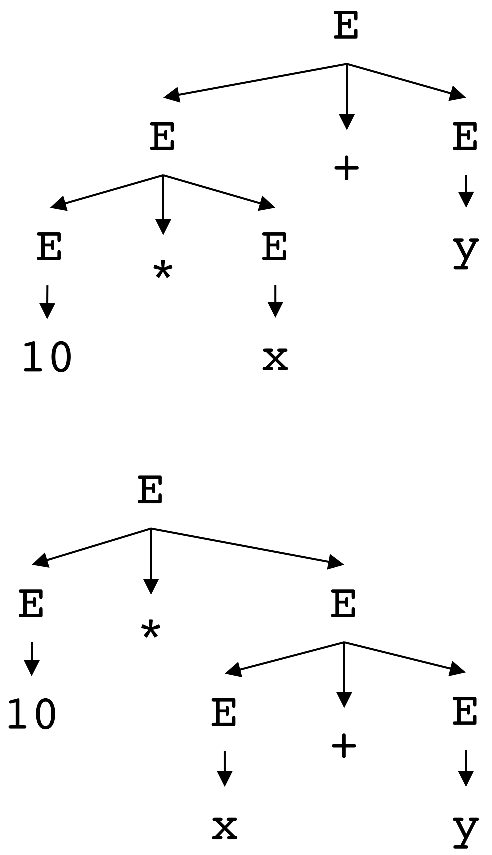
- Does it matter? Yes

Original program

- `10 * x + y`

The trees correspond to

- (10 * x) + y
- 12 * (x + y)
- which is correct?



AMBIGUITY

A grammar is ambiguous if there exists *at least one* program

- with *at least* two different derivation trees

Problems with ambiguity

- parsing should create abstract syntax tree
- more than one derivation tree means more than one abstract syntax tree
- parsing should be deterministic, i.e., same result for same input
- parser must chose one of the possible trees
- *the choice may change the semantics of the program*

EXERCISE

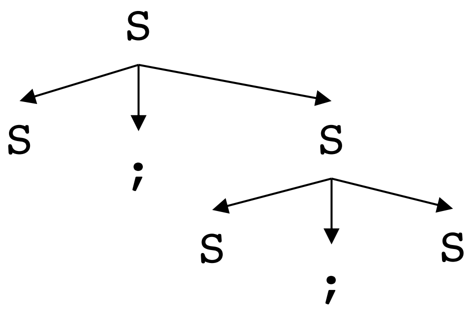
Is the grammar of straight line programs ambiguous?

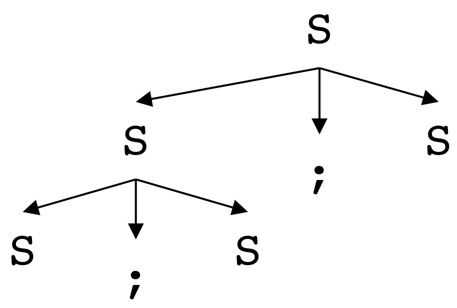
$$
\begin{aligned}
S &\rightarrow S;\ S \\
S &\rightarrow \mathit{id} := E \\
S &\rightarrow \textbf{print}\ (\ L\ ) \\
E &\rightarrow \mathit{id} \\
E &\rightarrow \mathit{num} \\
E &\rightarrow E + E \\
E &\rightarrow (\ S,\ E\ ) \\
L &\rightarrow E \\
L &\rightarrow L,\ E
\end{aligned}
$$

EXERCISE

Is the grammar of straight line programs ambiguous?

- Indeed, consider `S; S; S`





Does it matter?

S → S; S
S → *id* := E
S → **print** (L)
E → *id*
E → *num*
E → E + E
E → (S, E)
L → E
L → L, E

HANDLING AMBIGUITY

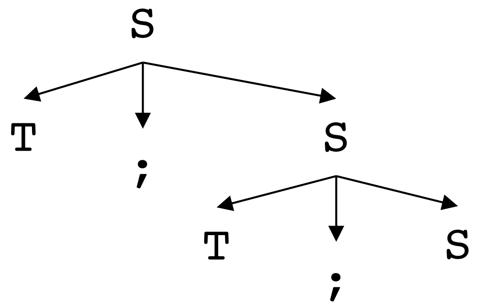
Even though ambiguity doesn't always matter we want to avoid it

We do this by rewriting the grammar to a grammar that is not ambiguous

For the straight line programs we can do this by factoring out the primitive statements

S → T; S | T
T → *id* := E
T → **print** (L)

This forces the derivation trees to a the following form



REWRITING EXPRESSIONS

For expressions we must rewrite the grammar to take

- precedence, and
- associativity into account

The first applied rule

- root, lowest precedence

The last applied rule

- leaf, highest precedence

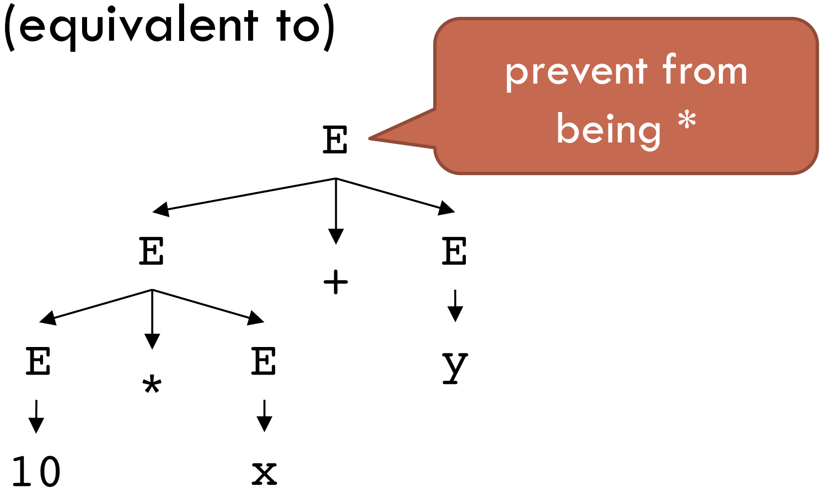
How can we encode this in the grammar?

- make sure that productions corresponding to operators with lower precedence occur earlier in the grammar

The only possible derivation tree for

- `10 * x + y`

should be (equivalent to)



ASSOCIATIVITY AND PRECEDENCE

Precedence of operators (lowest first)

- + -
- * /

a * b + c = (a * b) + c

a * b – c = (a * b) – c

What about associativity?

- + * are (both left and right) associative
- (a + b) + c = a + (b + c)

- - / are left associative
- a – b – c = (a – b) – c

REWRITING EXPRESSIONS

The first applied rule

- root, lowest precedence

The last applied rule

- leaf, highest precedence

$$
\begin{aligned}
\mathtt{E} \rightarrow\ & \mathtt{E} + \mathtt{E} \\
\mid\ & \mathtt{E} * \mathtt{E} \\
\mid\ & \mathtt{E} - \mathtt{E} \\
\mid\ & \mathtt{E} / \mathtt{E} \\
\mid\ & (\ \mathtt{E}\ ) \\
\mid\ & \mathit{id} \\
\mid\ & \mathit{num}
\end{aligned}
$$

First applied rule

- earliest in grammar

Last applied rule

- last in grammar

$$
\begin{aligned}
\mathtt{E} &\rightarrow \mathtt{E} + \mathtt{E} \mid \mathtt{E} - \mathtt{E} \\
&\ \mid \mathtt{F} \\[1em]
\mathtt{F} &\rightarrow \mathtt{F} * \mathtt{F} \mid \mathtt{F} / \mathtt{F} \\
&\ \mid \mathtt{T} \\[1em]
\mathtt{T} &\rightarrow (\ \mathtt{E}\ ) \mid \mathit{id} \mid \mathit{num}
\end{aligned}
$$

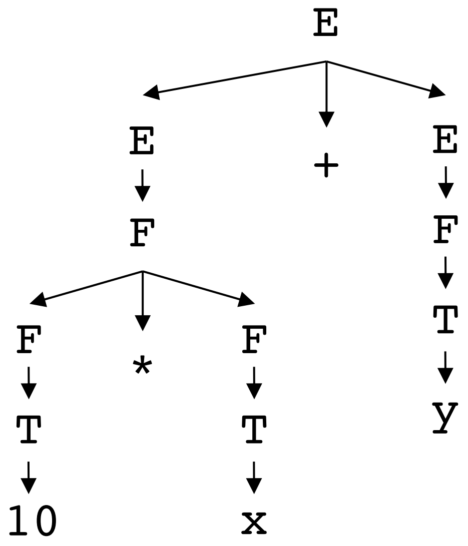
DID IT WORK?

Again, try `10 * x + y`

```
E → E + E
  | E - E
  | F

F → F * F
  | F / F
  | T

T → ( E ) | id | num
```



EXERCISE

Is the result unambiguous?

$$
\begin{aligned}
E \;&\rightarrow\; E + E \\
&\mid\; E - E \\
&\mid\; F \\[1em]
F \;&\rightarrow\; F * F \\
&\mid\; F \,/\, F \\
&\mid\; T \\[1em]
T \;&\rightarrow\; (\; E\;) \;\mid\; \mathit{id} \;\mid\; \mathit{num}
\end{aligned}
$$

EXERCISE

Is the result unambiguous?

```
E → E + E
  | E - E
  | F

F → F * F
  | F / F
  | T

T → ( E ) | id | num
```

No, try, e.g., 1 + 2 + 3

- can be derived left associatively (1 + 2) + 3
- can be derived right associatively 1 + (2 + 3)
- ok, from an operator perspective
- but ambiguous

Same for * and /

- but there only the left associative derivation should be possible

How?

ASSOCIATIVITY

Left associative

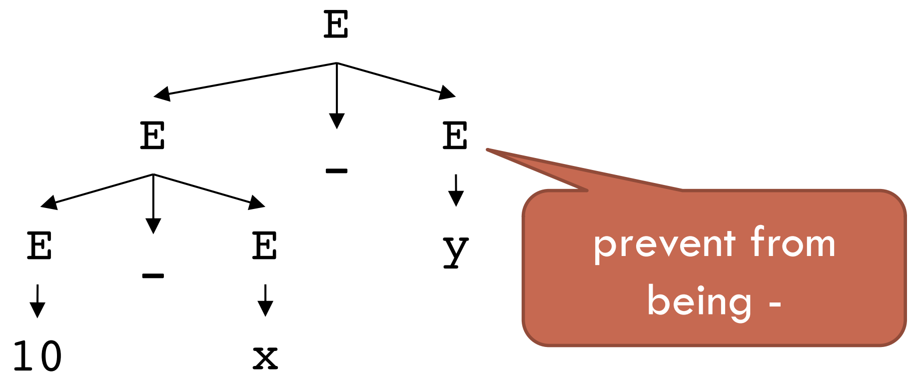
- left recursive trees
- consider 10 – x – y

Left recursive trees

- left recursive grammar

What about + and *

- does it matter?
- no, pick one



REWRITING EXPRESSIONS

Precedence

- + -
- * /

Associativity

- + * left associative (by choice)
- / - left associative (by necessity)

$$E \rightarrow E + F \mid E - F \mid F$$

$$F \rightarrow F * T \mid F \, / \, T \mid T$$

$$T \rightarrow (E) \mid \mathit{id} \mid \mathit{num}$$

Why did we pick + and * to be left associative?

- could we have done otherwise?

LEFT RECURSION

… but left associative operators give left recursive grammars …

$$E \rightarrow E + F \mid E - F \mid F$$

$$F \rightarrow F * T \mid F / T \mid T$$

$$T \rightarrow (E) \mid \mathit{id} \mid \mathit{num}$$

Most unfortunate for predictive recursive descent parser

Fortunately, this form of immediate left recursion is quite simple to remove

LEFT FACTORING

Consider the following simplified expression grammar

$E \rightarrow E + F \mid F$

Possible derivations are

- F
- F + F
- F + F + F
- …

Can be written

- F (+ F)*

Is there a way to express the same language

- that is not left recursive?

LEFT FACTORING

Consider the following simplified expression grammar

$$E \rightarrow E + F \mid F$$

Possible derivations are

- F
- F + F
- F + F + F
- …

Can be written

- F (+ F)*

Is there a way to express the same language

- that is not left recursive?

Yes!

$$E \rightarrow F\ T$$
$$T \rightarrow +\ F\ T \mid \lambda$$

Same derivations, but

- right recursive!

LEFT FACTORING

Immediate left recursion, general case

$A \rightarrow A\ \alpha\ |\ \beta$

Rewrite as follows

$A ::= \beta\ T$
$T ::= \alpha\ T\ |\ \lambda$

α and β are meta variables

- that match agains the grammatical rules

Compare

$E \rightarrow E + F\ |\ F$

and

$E \rightarrow F\ T$
$T \rightarrow +\ F\ T\ |\ \lambda$

EXERCISE

Immediate left recursion, general case

$\mathtt{A} \rightarrow \mathtt{A}\ \alpha\ |\ \beta$

Rewrite as follows

$\mathtt{A}\ ::=\ \beta\ \mathtt{T}$
$\mathtt{T}\ ::=\ \alpha\ \mathtt{T}\ |\ \lambda$

Left factorize the grammar for comma separated lists of numbers

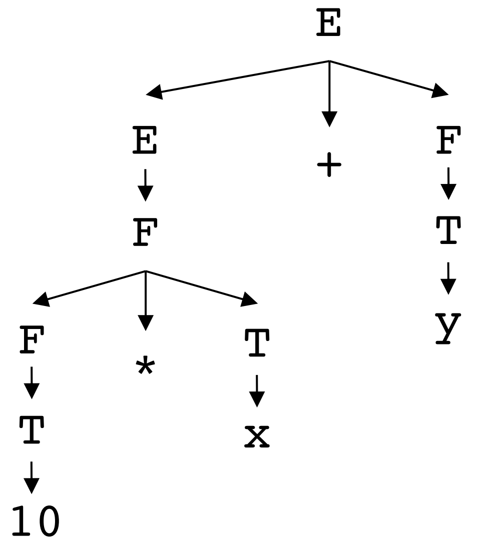
$\mathtt{L} \rightarrow \mathtt{L,}\ \mathit{num}\ |\ \lambda$

ABSTRACT SYNTAX TREES

ABSTRACT SYNTAX

Consider our rewritten expression grammar

$$\texttt{E} \rightarrow \texttt{E + F} \mid \texttt{E - F} \mid \texttt{F}$$
$$\texttt{F} \rightarrow \texttt{F * T} \mid \texttt{F / T} \mid \texttt{T}$$
$$\texttt{T} \rightarrow \texttt{(E)} \mid \mathit{id} \mid \mathit{num}$$



The derivation trees contain a lot of unnecessary information

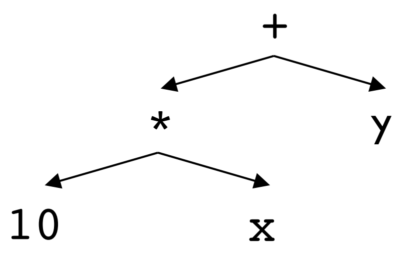
- separators (punctuation) only present to disambiguate
  - ( )
- syntactic categories only introduced to disambiguate
  - F and T (to encode precedence)

ABSTRACT SYNTAX

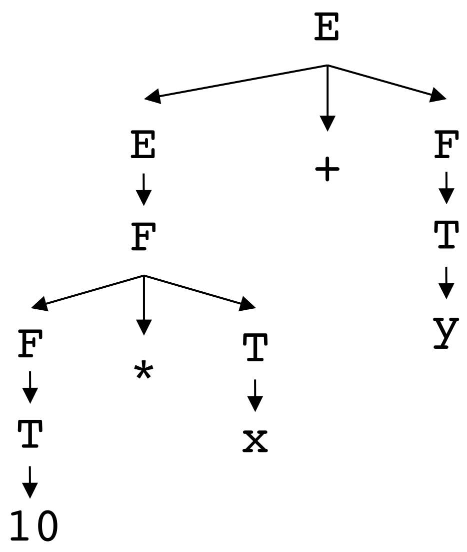
Later compiler stages are only interested in information that is important for the semantics of the program

For `10 * x + y`

Abstract syntax tree



Derivation tree/concrete syntax tree



DESIGNING ABSTRACT SYNTAX

Keep only the essentials

- ambiguities not an issue – will be generated from parsing unambiguous grammars
- remove all syntax that only guides the parsing process

For the expressions, start in the grammar that was before the encoding of associativity and precedence

$$E \rightarrow E + E \mid E * E \mid E - E \mid E \,/\, E \mid (E) \mid \mathit{id} \mid \mathit{num}$$

Remove unnecessary decorations

$$E \rightarrow E + E \mid E * E \mid E - E \mid E \,/\, E \mid \mathit{id} \mid \mathit{num}$$

DESIGNING ABSTRACT SYNTAX

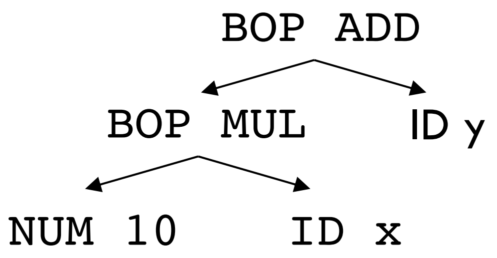
Give proper names (easier to map to code)

$$\texttt{E} \longrightarrow \texttt{ADD E E} \mid \texttt{MUL E E} \mid \texttt{SUB E E} \mid \texttt{DIV E E} \mid \texttt{ID } \textit{id}$$
$$\mid \texttt{NUM } \textit{num}$$

Each syntactic category corresponds to one superclass and each production corresponds to one class

Contract operators to reduce number of needed classes

$$\texttt{E} \longrightarrow \texttt{BOP OP E E} \mid \texttt{ID } \textit{id} \mid \texttt{NUM } \textit{num}$$
$$\textit{OP} \longrightarrow \texttt{ADD} \mid \texttt{SUB} \mid \texttt{MUL} \mid \texttt{DIV}$$



MAPPING TO CODE

Chose longer names!

- only to illustrate mapping

$$
\begin{aligned}
\mathtt{E} &\rightarrow \mathtt{BOP}\ \mathtt{OP}\ \mathtt{E}\ \mathtt{E} \\
&\mid \mathtt{ID}\ \mathit{id} \\
&\mid \mathtt{NUM}\ \mathit{num} \\
\mathit{OP} &\rightarrow \mathtt{ADD} \mid \mathtt{SUB} \mid \mathtt{MUL} \mid \mathtt{DIV}
\end{aligned}
$$

```
public class E {}

public class BOP : E {
  public OP op;
  public E left, right;
  public enum OP { ADD, SUB, MUL, DIV }
}
}

public class ID : E {
  public string id;

}

public class NUM : E {
  public int num;
}
```

EXERCISE

Create abstract syntax for

$S \rightarrow S;\ S$
$S \rightarrow id\ :=\ E$
$S \rightarrow \mathbf{print}\ (\ L\ )$
$E \rightarrow id$
$E \rightarrow num$
$E \rightarrow E\ +\ E$
$E \rightarrow (\ S,\ E\ )$
$L \rightarrow E$
$L \rightarrow L,\ E$

EXERCISE

Create abstract syntax for

S → S; S
S → *id* := E
S → **print** (L)
E → *id*
E → *num*
E → E **+** E
E → (S, E)
L → E
L → L, E

Remove all extra decoration

S → S; S
S → *id* := E
S → **print** L
E → *id*
E → *num*
E → E **+** E
E → S, E
L → E
L → L, E

EXERCISE

Replace list encodings with actual lists

P → [S]
S → *id* := E
S → **print** [L]
E → *id*
E → *num*
E → E **+** E
E → [S], E

Notice new syntactic category P for programs

It does not matter that we extend the language (the empty list)

- print statements without parameters will never be generated from the original grammar

Introduce proper names

P → Prog [S]
S → Asn *id* E
S → Print [L]
E → Var *id*
E → Num *num*
E → Edd E E
E → Let [S] E

Only contains the essentials, and translates well to classes

NEXT TIME

Predictive recursive descent!