Compiler Construction
Lent Term 2020
Parsing Part I
Lecture 14

Timothy G. Griffin
tgg22@cam.ac.uk
Computer Laboratory
University of Cambridge

Lecture 14 -- 16
Parsing (some) context-free grammars

1. Context-free grammars
2. The ambiguity problem
3. Top-down parsing (LL(1))
   1. Recursive descent parsing (hand coded)
   2. Predictive parsing (table driven)
4. Bottom-up parsing (Lectures 15,16)
   1. SLR(1) (Simple LR)
   2. LR(1)

Programming Language Syntax

6.7 Declarations

Syntax

declaration:
  declaration-specifiers $\textit{init-declarator-list}_{opt}$ **;**
  static_assert-declaration

declaration-specifiers:
  storage-class-specifier $\textit{declaration-specifiers}_{opt}$
  type-specifier $\textit{declaration-specifiers}_{opt}$
  type-qualifier $\textit{declaration-specifiers}_{opt}$
  function-specifier $\textit{declaration-specifiers}_{opt}$
  alignment-specifier $\textit{declaration-specifiers}_{opt}$

init-declarator-list:
  init-declarator
  init-declarator-list **,** *init-declarator*

init-declarator:
  declarator
  declarator **=** *initializer*

A small fragment of the C standard. How can we turn this specification into a parser that reads a text file and produces a syntax tree?

Context-Free Grammars (CFGs)

$$G = (N, T, P, S)$$

$N:$ set of nonterminals

$T:$ set of terminals

$P \subseteq N \times (N \cup T)^*:$ a set of productions

$S \in \mathrm{N}:$ start symbol

Each $(A, \alpha) \in P$ is written as $A \rightarrow \alpha$

Example CFG

$$G_1 = (N_1, T_1, P_1, E)$$

$N_1 = \{\mathrm{E}\}$ $\quad\quad$ $T_1 = \{+, *, (,), \mathrm{id}\}$

$P_1 :$

$$\mathrm{E} \rightarrow \mathrm{E} + \mathrm{E} \mid \mathrm{E} * \mathrm{E} \mid (\mathrm{E}) \mid \mathrm{id}$$

This is shorthand for

$$P_1 = \{(\mathrm{E}, \mathrm{E} + \mathrm{E}), (\mathrm{E}, \mathrm{E} * \mathrm{E}), (\mathrm{E}, (\mathrm{E})), (\mathrm{E}, \mathrm{id})\}$$

Derivations

Notation conventions:

$$\alpha, \beta, \gamma, \cdots \in (N \cup T)^*$$

$$A, B, C, \cdots \in N$$

Given: $\alpha A \beta$ and a production $A \rightarrow \gamma$

a derivation step is written as

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

$\Rightarrow^+$ means one or more derivation steps and

$\Rightarrow^*$ means zero or more derivation steps

Example derivations

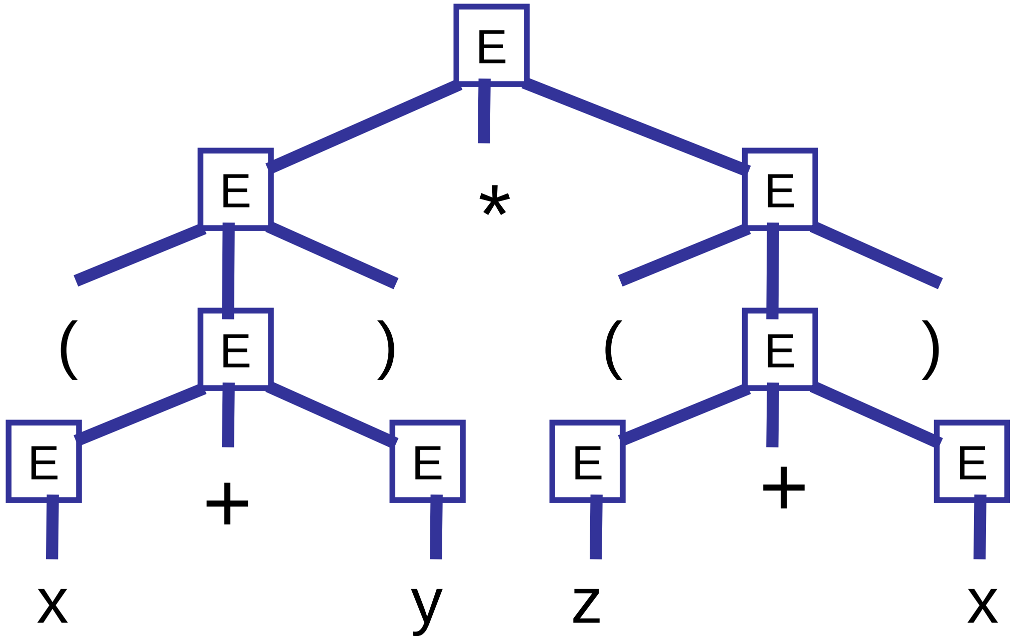
$$
\begin{aligned}
\mathrm{E} &\Rightarrow \mathrm{E} * \mathrm{E} \\
&\Rightarrow (E) * E \\
&\Rightarrow (E + E) * E \\
&\Rightarrow (x + E) * E \\
&\Rightarrow (x + y) * E \\
&\Rightarrow (x + y) * (E) \\
&\Rightarrow (x + y) * (E + E) \\
&\Rightarrow (x + y) * (z + E) \\
&\Rightarrow (x + y) * (z + x)
\end{aligned}
$$

A leftmost derivation

$$
\begin{aligned}
\mathrm{E} &\Rightarrow \mathrm{E} * \mathrm{E} \\
&\Rightarrow E * (E) \\
&\Rightarrow E * (E + E) \\
&\Rightarrow E * (E + x) \\
&\Rightarrow E * (z + x) \\
&\Rightarrow (E) * (z + x) \\
&\Rightarrow (E + E) * (z + x) \\
&\Rightarrow (E + y) * (z + x) \\
&\Rightarrow (x + y) * (z + x)
\end{aligned}
$$

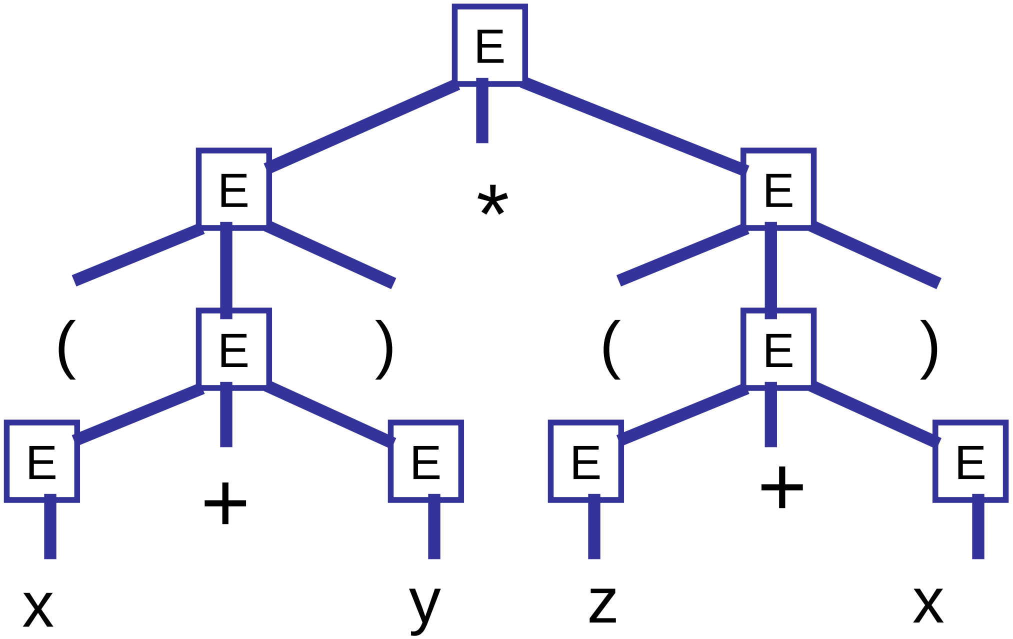
A rightmost derivation

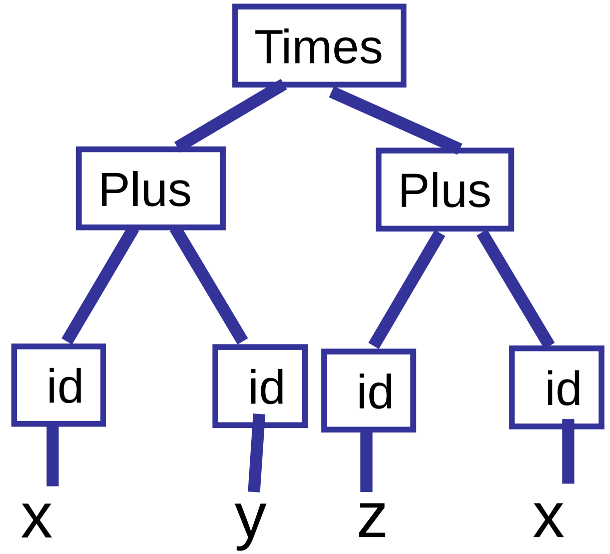
Derivation Trees



The derivation tree for (x + y) * (z + x). All derivations of this expression will produce the same derivation tree.

Concrete vs. Abstract Syntax Trees





parse tree = derivation tree = concrete syntax tree

An AST contains only the information needed to generate an intermediate representation

L(G) = The Language Generated by Grammar G

$$L(G) = \{ w \in T^* \mid S \Rightarrow^+ w \}$$

For example, if G has production s

$$S \rightarrow aSb \mid \varepsilon$$

then

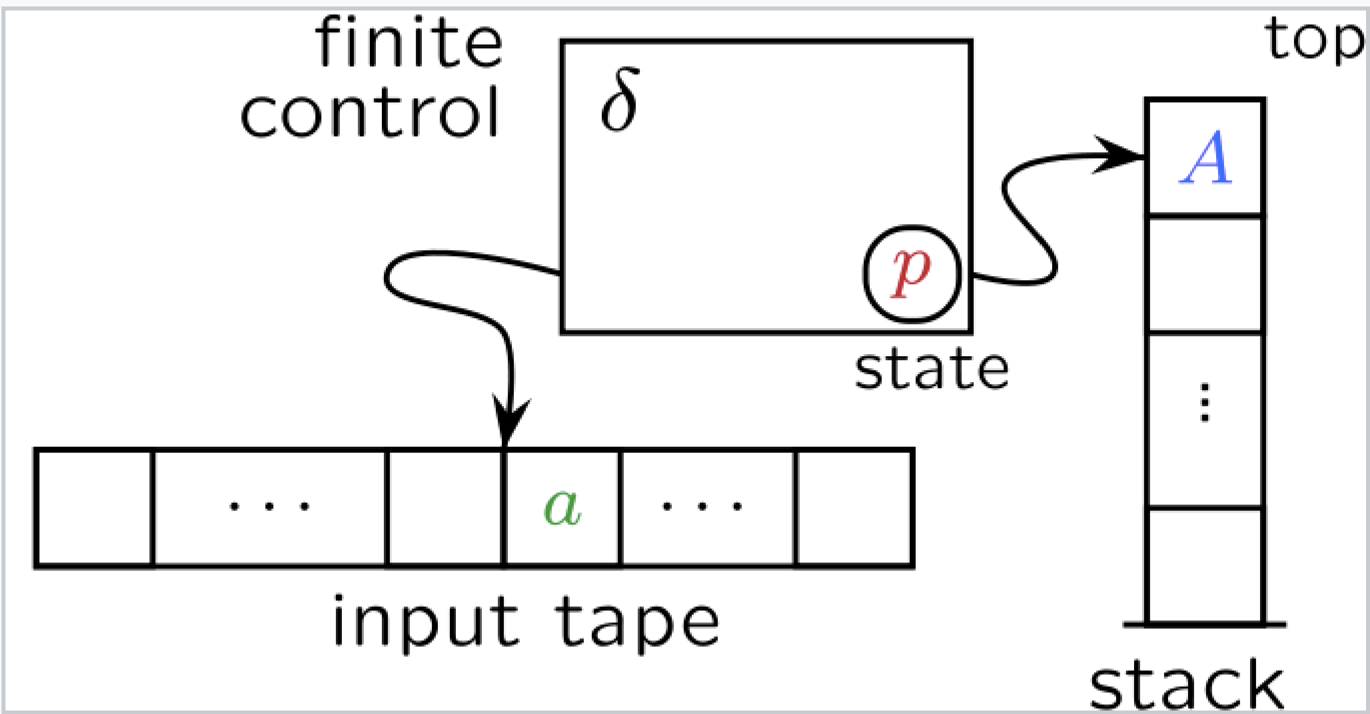
$$L(G) = \{ a^n b^n \mid n \geq 0 \}.$$

So CFGs can capture more than regular languages!

Pushdown Automata (PDAs)

Regular languages are accepted by Finite Automata.
Context-free languages are accepted by Pushdown Automata, a finite automata augmented with a stack.

Illustration from https://en.wikipedea.org/wiki/Pushdown_automaton



A diagram of a pushdown automaton

Pushdown Automata (PDAs)

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z)$$

$Q:\text{states}$ $\Sigma:\text{alphabet}$ $\Gamma:\text{stack symbols}$

$q_0 \in \mathrm{Q}:\text{start state}$

$\mathrm{Z} \in \Gamma:\text{initial stack symbol}$

$$\delta : \forall q \in Q, a \in (\Sigma \cup \{\varepsilon\}), X \in \Gamma,$$

$$\delta(\mathrm{q}, \mathrm{a}, \mathrm{X}) \subseteq \mathrm{Q} \times \Gamma^*$$

Pushdown Automata (PDAs)

$(q', \beta) \in \delta(\mathrm{q}, \mathrm{a}, \mathrm{X})$ means that when the machine is in state q reading a with X on top of the stack, it can move to state q' and replace X with β. That is, it "pops" X and "pushes" β (leftmost symbol is top of stack).

Pushdown Automata (PDAs)

For $q \in Q, w \in \Sigma^*, \alpha \in \Gamma^*$

$$(q, w, \alpha)$$

is called an instantaneous description (ID). It denotes the PDA in state q looking at the first symbol of w, with α on the stack (top at left).

Language accepted by a PDA

For $(q,\beta)\in\delta(q,a,X),\ a\in\Sigma$ define the relation $\rightarrow$ on IDs as

$$(q,aw,X\alpha)\rightarrow(q',w,\beta\alpha)$$

and for $(q,\beta)\in\delta(q,\varepsilon,X)$ as

$$(q,w,X\alpha)\rightarrow(q',w,\beta\alpha)$$

$$L(M)=$$

$$\{w\in\Sigma^{*}\mid\exists q\in Q,(q_0,w,Z)\rightarrow^{+}(q,\varepsilon,\varepsilon)\}$$

Exercise : work out the details of this PDA

$$(q_{0,} aaabbb, Z)$$
$$\rightarrow (q_a, aabbb, A)$$
$$\rightarrow (q_a, abbb, AA)$$
$$\rightarrow (q_a, bbb, AAA)$$
$$\rightarrow (q_b, bb, AA)$$
$$\rightarrow (q_b, b, A)$$
$$\rightarrow (q_b, \varepsilon, \varepsilon)$$

$$L(M) = \{ a^n b^n \mid n \geq 0 \}$$

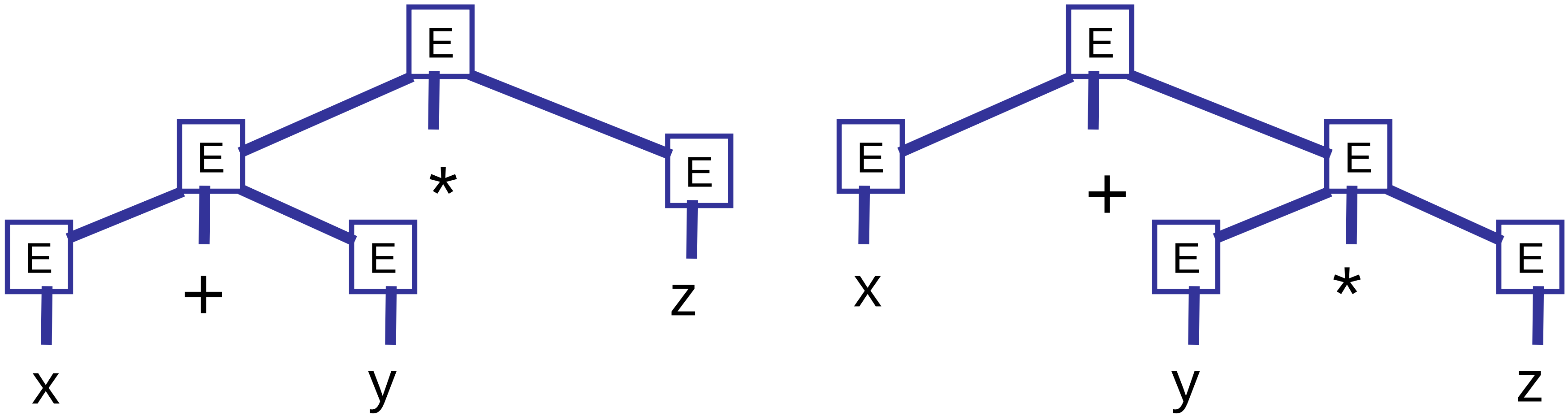
PDAs and CFGs Facts
(we will not prove them)

1) For every CFG G there is a PDA M such that $L(G) = L(M)$.

2) For every PDA M there is a CFG G such that $L(G) = L(M)$.

Parsing problem solved? Given a CFG G just construct the PDA M? Not so fast! For programmin g languages we want M to be determinis tic!

Origins of nondeterminism? Ambiguity!



Both derivation trees correspond "x + y * z".
But (x+y) * z is not the same as x + (y * z).

This type of ambiguity will cause problems when we try to go from program texts to derivation trees! Semantic ambiguity!

We can often modify the grammar in order to eliminate ambiguity

$$G_2 = (N_2, T_1, P_2, E)$$

$N_2 = \{\mathrm{E, T, F}\}$ $\qquad$ $T_1 = \{+, *, (,), \mathrm{id}\}$

$P_2:$

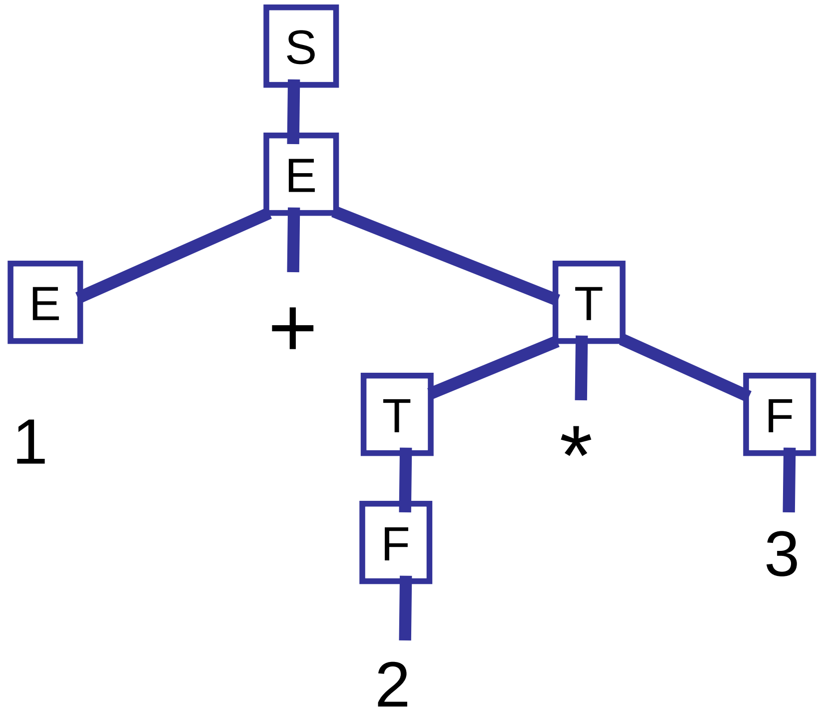
$\mathrm{E} \rightarrow \mathrm{E} + \mathrm{T} \mid \mathrm{T}$ $\qquad$ (expressions)

$\mathrm{T} \rightarrow \mathrm{T} * \mathrm{F} \mid \mathrm{F}$ $\qquad$ (terms)

$\mathrm{F} \rightarrow (\mathrm{E}) \mid \mathrm{id}$ $\qquad$ (factors)

Can you prove that $L(G_1) = L(G_2)$?

The modified grammar eliminates ambiguity



This is now the *unique* derivation tree for

x + y * z

Fun Fun Facts

(1) Some context-free languages are *inherently ambiguous* --- every context-free grammar for them will be ambiguous. For example:

$$L = \left\{ a^n b^n c^m d^m \mid m \geq 1, n \geq 1 \right\}$$
$$\cup \left\{ a^n b^m c^m d^n \mid m \geq 1, n \geq 1 \right\}$$

(2) Checking for ambiguity in an arbitrary context-free grammar is not decidable! Ouch!

(3) Given two grammars G1 and G2, checking L(G1) = L(G2) is not decidable! Ouch!

See Hopcroft and Ullman, "Introduction to Automata Theory, Languages, and Computation"

Two approaches to building stack-based parsing machines: top-down and bottom-up

- **Top Down : attempts a left-most derivation. We will look at two techniques:**
  - **Recursive decent (hand coded)**
  - **Predictive parsing (table driven)**
- **Bottom-up : attempts a right-most derivation backwards. We will look at two techniques:**
  - **SL(1) : Simple LR(1)**
  - **LR(1)**

Bottom-up techniques are strictly more powerful. That is, they can parse more grammars.

Recursive Descent Parsing

(G5)

S :: = if E then S else S
| begin S L
| print E

E ::= NUM = NUM

L ::= end
| ; S L

```
int tok = getToken();

void advance() {tok = getToken();}
void eat (int t) {if (tok == t) advance(); else
error();}

void S() {switch(tok) {
      case IF:    eat(IF); E(); eat(THEN);
                  S(); eat(ELSE); S(); break;
      case BEGIN: eat(BEGIN); S(); L(); break;
      case PRINT: eat(PRINT); E(); break;
      default: error();
     }}

void L() {switch(tok) {
      case END:  eat(END); break;
      case SEMI: eat(SEMI); S(); L(); break;
      default: error();
     }}

void E() {eat(NUM) ; eat(EQ); eat(NUM); }
```

Parse corresponds to a left-most derivation constructed in a “top-down” manner

Example From Andrew Appel, “Modern Compiler Implementation in Java” page 46

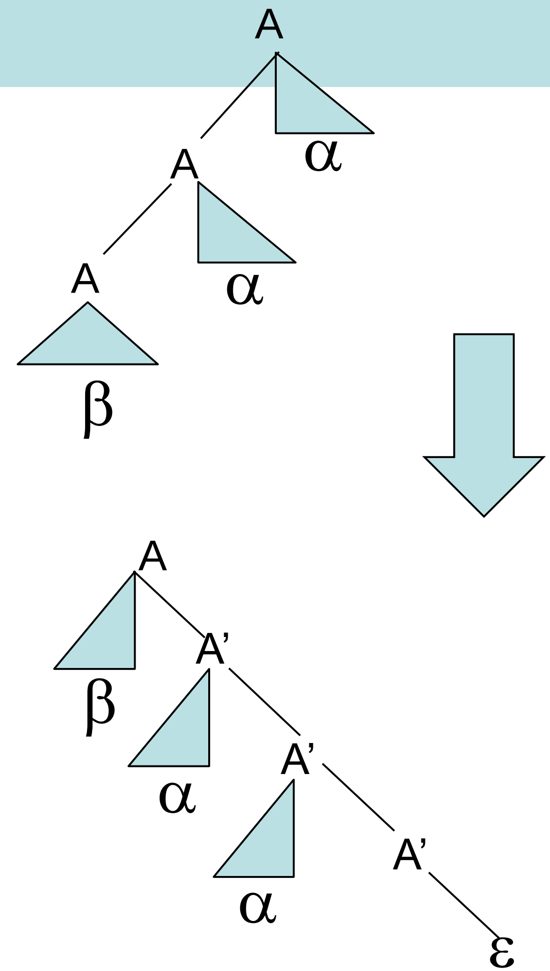
But the production $E \to T$ in G_2 will lead to an infinite loop!

Eliminate left recursion!

A -> Aα1 | Aα2 | . . . | Aαk |
β1 | β2 | . . . | βn

A -> β1 A' | β2 A' | . . . | βn A'

A' -> α1 A' | α2 A'| . . . | αk A' | ε



For eliminating left-recursion in general, see Aho and Ullman.

Eliminate left recursion

$$G_3 = (N_3, T_1, P_3, E)$$

$N_2 = \{\mathrm{E}, \mathrm{E'}, \mathrm{T}, \mathrm{T'}, \mathrm{F}\}$ $\quad$ $T_1 = \{+, *, (,), \mathrm{id}\}$

P_2 :

$$\mathrm{E} \to \mathrm{T\ E'}$$

$$E' \to +T\ E' \mid \varepsilon$$

$$\mathrm{T} \to \mathrm{F\ T'}$$

$$T' \to\ *F\ T' \mid \varepsilon$$

$$\mathrm{F} \to (\mathrm{E}) \mid \mathrm{id}$$

Can you prove that $L(G_2) = L(G_3)$?

Recursive descent pseudocode

$$\text{getE()} = \text{getT()};\ \ \text{getE}'\ ()$$

$$\text{get}E'() = \text{if}\ \ \text{token}() = "+"\ \text{then}\ \ \text{eat}("+");\ \text{get}T();\ \text{get}E'()$$

$$\text{getT}(\ ) = \text{getF}\ ();\ \text{getT}'\ ()$$

$$\text{get}T'() = \text{if}\ \ \text{token}() = "*"\ \text{then}\ \ \text{eat}("*");\ \text{get}F();\ \text{get}T'()$$

$$\text{getF}() = \text{if}\ \ \text{token}() = id$$

$$\text{then}\ \ \text{eat}(id\,)$$

$$\text{else}\ \text{eat}("\,(");\ \text{get}E();\ \text{eat}("\,)")$$

Where's the stack machine?
It's implicit in the call stack!

Parsing (x+y)*(z+x) using a call to getE()

			eat("(")	getE()	
		getF()	getF()	getF()	
	getT()	getT()	getT()	getT()	. . .
getE()	getE()	getE()	getE()	getE()	

call stack over time …