

Compiler Construction

Lecture 3: Context-free grammars

Jeremy Yallop

`jeremy.yallop@cl.cam.ac.uk`

Lent 2026

What is the role of a parser?

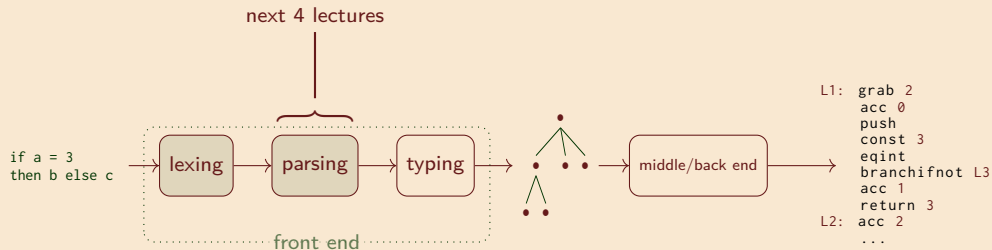
CFGs

Derivations

PDAs

Ambiguity

Top-down &
bottom-up



Context-free grammars

What are context-free grammars?

CFGs



Derivations

PDAs

Ambiguity

Top-down &
bottom-up

A small fragment of the C standard:

6.7 Declarations

Syntax

declaration:

declaration-specifiers init-declarator-list_{opt} ;
static-assert-declaration

declaration-specifiers:

storage-class-specifier declaration-specifiers_{opt}
type-specifier declaration-specifiers_{opt}
type-qualifier declaration-specifiers_{opt}
function-specifier declaration-specifiers_{opt}
alignment-specifier declaration-specifiers_{opt}

init-declarator-list:

init-declarator
init-declarator-list , init-declarator

init-declarator:

declarator
declarator = initializer

Today's Q: how can we turn this **declarative specification** into a program?

Context-Free Grammars (CFGs)

CFGs



Derivations

PDA's

Ambiguity

Top-down &
bottom-up

nonterminals N

productions $P \subseteq N \times (N \cup T)^*$

$$M = \langle N, T, P, S \rangle$$

terminals T

start symbol $S \in N$

Each $\langle A, \alpha \rangle \in P$ is written as $A \rightarrow \alpha$

CFGs



Derivations

PDAs

Ambiguity

Top-down &
bottom-up

$$G_1 = \langle N_1, T_1, P_1, E \rangle$$

where

$$N_1 = \{E\}$$

$$T_1 = \{+, *, (,), \text{id}\}$$

$$P_1 = E \rightarrow \begin{array}{l} E + E \\ E * E \\ (E) \\ \text{id} \end{array}$$

NB: P_1 definition is shorthand for

$$P_1 = \{\langle E, E + E \rangle, \langle E, E * E \rangle, \langle E, (E) \rangle, \langle E, \text{id} \rangle\}$$

Derviations

CFGs

Derivations



PDAs

Ambiguity

Top-down &
bottom-up

Notation conventions:

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

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

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

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

CFGs

Derivations



PDAs

Ambiguity

Top-down &
bottom-up

A **leftmost** derivation

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

A **rightmost** derivation

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

CFGs

Derivations

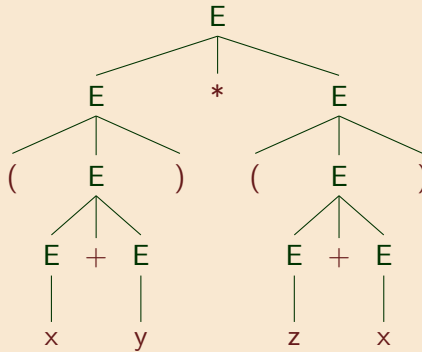


PDA's

Ambiguity

Top-down &
bottom-up

The derivation tree for $(x+y) * (z+x)$:



All derivations of this expression will produce the same derivation tree.

Concrete vs abstract syntax trees

CFGs

Derivations

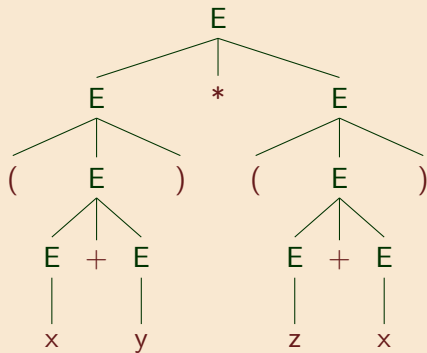


PDAs

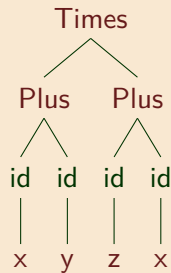
Ambiguity

Top-down &
bottom-up

(Terminology: $=$ **parse** tree
 $=$ **derivation** tree
 $=$ **concrete syntax** tree)



An **abstract syntax** tree contains only the information needed to generate an intermediate representation



The language generated by a grammar

CFGs

$L(G)$: the **language generated by G**

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

For example, if G has productions

$$S \rightarrow aSb \mid \epsilon$$

then

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

So CFGs can capture more than regular languages.

Derivations



PDA's

Ambiguity

Top-down &
bottom-up

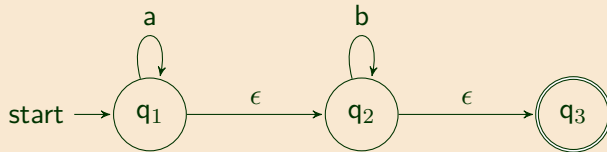
Pushdown automata

Pushdown automata (PDAs)

CFGs

Regular languages are accepted by **finite automata**:

a^*b^*

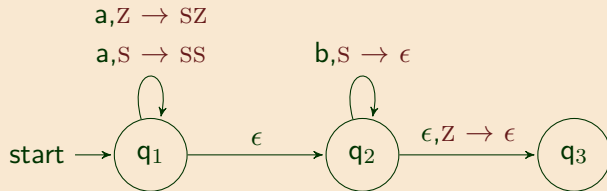


Derivations

PDAs

Context-free languages are accepted by **pushdown automata**, finite automata augmented with stacks.

$a^n b^n$



Ambiguity

Top-down &
bottom-up

Pushdown automata (PDAs)

CFGs

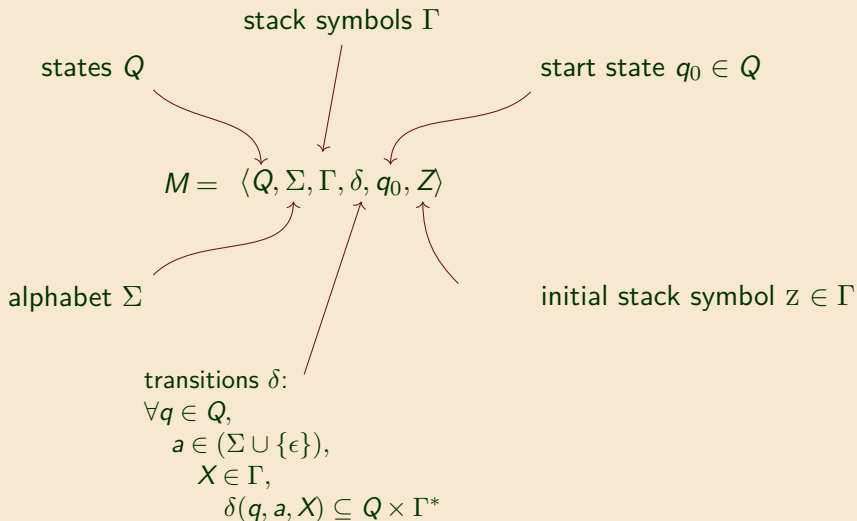
Derivations

PDAs

● ○ ○ ○ ○

Ambiguity

Top-down &
bottom-up



Pushdown automata (PDAs)

CFGs

Derivations

PDAs



Ambiguity

Top-down &
bottom-up

$\langle q', \beta \rangle \in \delta(q, a, X)$ means:

When the machine is $\left\{ \begin{array}{l} \text{in state } q, \text{ and} \\ \text{reading } a \text{ and} \\ \text{with } X \text{ on top of the stack,} \end{array} \right.$

it can $\left\{ \begin{array}{l} \text{move to state } q' \text{ and} \\ \text{replace } X \text{ with } \beta. \end{array} \right.$

i.e. it *pops* X from the
stack and *pushes* β .

Pushdown automata (PDAs)

CFGs

PDAs

Derivations

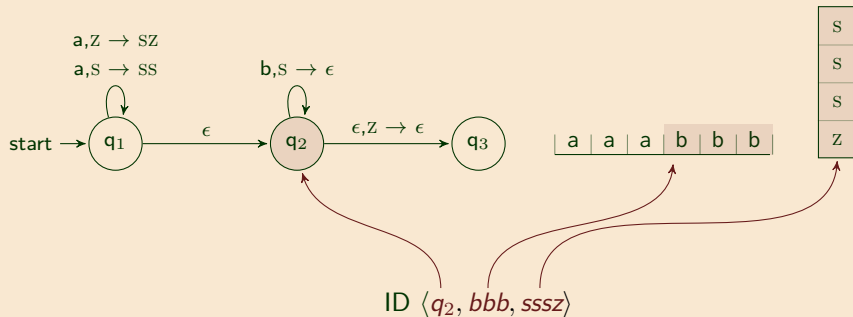
Ambiguity

Top-down &
bottom-up

For $q \in Q$, $w \in \Sigma^*$, $\alpha \in \Gamma^*$, $\langle q, w, \alpha \rangle$ is called an **instantaneous description** (ID).

in state q

It denotes the PDA looking at the first symbol of w
with α on the stack



Language accepted by a PDA

CFGs

Derivations

PDA's

Ambiguity

Top-down &
bottom-up

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

$$\langle q, aw, X\alpha \rangle \rightarrow \langle q', w, \beta\alpha \rangle$$

and for $\langle q', \beta \rangle \in \delta(q, \epsilon, X)$ as

$$\langle q, w, X\alpha \rangle \rightarrow \langle q', w, \beta\alpha \rangle$$

Then the **language accepted by M** , $L(M)$, is:

$$L(M) = \{w \in \Sigma^* \mid \exists q \in Q, \langle q_0, w, Z \rangle \rightarrow^+ \langle q, \epsilon, \epsilon \rangle\}$$

NB: M accepts words **in any state** when the stack and remaining input are empty

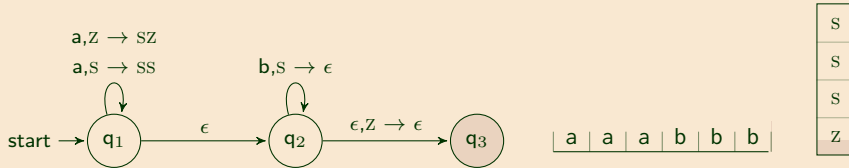
CFGs

Derivations

PDAs

Ambiguity

Top-down &
bottom-up



$\langle q_1, aaabbb, Z \rangle$

$\langle q_1, aabbb, SZ \rangle$

$\langle q_1, abbb, SSZ \rangle$

$\langle q_2, bbb, SSSZ \rangle$

$\langle q_2, bb, SSZ \rangle$

$\langle q_2, b, SZ \rangle$

$\langle q_2, \epsilon, Z \rangle$

$\langle q_3, \epsilon, \epsilon \rangle$

PDA and CFG facts (without proof)

CFGs

Derivations

PDA's



Ambiguity

Top-down &
bottom-up

PDA and CFG facts:

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

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

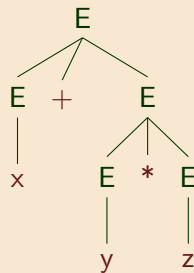
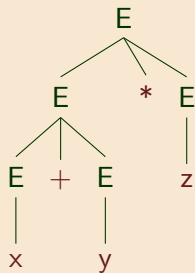
Is the parsing problem solved? Given a CFG G we can construct the PDA M .
No! For programming languages we want M to be **deterministic**

Ambiguity

The origin of nondeterminism is ambiguity

CFGs

Derivations



PDAs

Ambiguity

● ○ ○ ○

Top-down &
bottom-up

Both derivation trees correspond to $x + y * z$.

But $(x + y) * z$ is not the same as $x + (y * z)$.

Ambiguity causes problems going from program texts to derivation trees.

Modifying the grammar to eliminate ambiguity

CFGs

We can often modify the grammar to eliminate ambiguity.

$$G_2 = \langle N_2, T_1, P_2, E \rangle$$

where

$$\begin{array}{ll} E \rightarrow E + T \mid T & \text{(expressions)} \\ P_2 = \quad T \rightarrow T * F \mid F & \text{(terms)} \\ \quad F \rightarrow (E) \mid id & \text{(factors)} \end{array}$$

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

Derivations

PDA's

Ambiguity

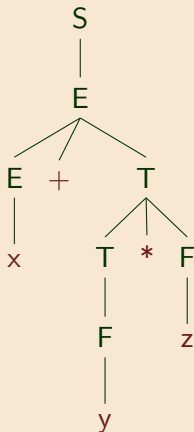


Top-down &
bottom-up

The modified grammar eliminates ambiguity

CFGs

The modified grammar eliminates ambiguity. The following is now the **unique** derivation tree for $x + y * z$:



Derivations

PDA's

Ambiguity



Top-down &
bottom-up

CFGs

1. Some context-free languages are **inherently ambiguous** — every CFG for them is ambiguous. For example

$$\begin{aligned} L &= \{a^n b^n c^m d^m \mid m \geq 1, n \geq 1\} \\ &\cup \{a^n b^m c^m d^n \mid m \geq 1, n \geq 1\} \end{aligned}$$

Derivations

PDAs

2. Checking for **ambiguity** in an arbitrary CFG is **not decidable**.
3. Given two grammars G_1 and G_2 , checking $L(G_1) = L(G_2)$ is **not decidable**.

Ambiguity

Top-down &
bottom-up

(See Hopcroft & Ullman, “Introduction to Automata Theory, Languages, and Computation”)

Top-down & bottom-up

Two approaches to building stack-based parsing machines

CFGs

Top-down: attempts a **leftmost derivation**. We'll look at two techniques:

Recursive
descent
(hand coded)

Predictive
parsing
(table driven)

Derivations

PDAs

Bottom-up: attempts a **rightmost derivation backwards**. We'll look at two techniques:

SLR(1)
(Simple LR(1))

LR(1)

Ambiguity

Top-down &
bottom-up

Bottom-up techniques are strictly more powerful (can parse more grammars)



Recursive descent parsing

CFGs

```
type token =  
  ADD | MUL | LPAREN | RPAREN | IDENT of string
```

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \epsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid \epsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

Derivations

```
let rec  
  e toks = e' (t toks)  
and e' = function  
  | ADD :: toks → e' (t toks)  
  | toks       → toks (* ε *)  
and t toks = t' (f toks)  
and t' = function  
  | MUL :: toks → t' (f toks)  
  | toks       → toks (* ε *)  
and f = function  
  | LPAREN :: toks →  
    (match e toks with  
    | RPAREN :: toks → toks  
    | _              → failwith "RPAREN")  
  | IDENT _ :: toks → toks  
  | _              → failwith "F"
```

PDAs

Ambiguity

Top-down &
bottom-up

Parse corresponds to a leftmost derivation constructed in a **top-down** manner



Left recursion & recursive-descent parsing

CFGs

Derivations

Recursive descent parsing is not suitable for G_2 .

Left-recursion $E \rightarrow E + T$ will lead to an infinite loop:

```
let rec
  e toks = match e toks (* loop! *) with
    | ADD :: toks → ...
```

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

PDA's

Ambiguity

Top-down &
bottom-up



Eliminating left recursion

CFGs

$$G_2 = \langle N_2, T_1, P_2, E \rangle$$

$$G_3 = \langle N_3, T_1, P_3, E \rangle$$

Derivations

where

where

...

...

$$P_2 = \begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid id \end{array}$$

$$P_3 = \begin{array}{lcl} E & \rightarrow & T E' \\ E' & \rightarrow & + T E' \mid \epsilon \\ T & \rightarrow & F T' \\ T' & \rightarrow & * F T' \mid \epsilon \\ F & \rightarrow & (E) \mid id \end{array}$$

PDAs

Ambiguity

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

Top-down &
bottom-up



The stack machine is *implicit in the call stack*

CFGs

Derivations

PDAs

Ambiguity

Top-down &
bottom-up

```
let rec
  e toks = e' (t toks)
and e' = function
| ADD :: toks → e' (t toks)
| toks       → toks (* ∈ *)
and t toks = t' (f toks)
and t' = function
| MUL :: toks → t' (f toks)
| toks       → toks (* ∈ *)
and f = function
| LPAREN :: toks →
  (match e toks with
  | RPAREN :: toks → toks
  | _              → failwith "RPAREN")
| IDENT _ :: toks → toks
| _              → failwith "F"
```

Parsing $x + y * z$, i.e.

```
[IDENT "x";
  ADD;
  IDENT "y";
  MUL;
  IDENT "z"]
```

Evaluation trace:

```
e toks
~> e' (t toks)
~> e' (t' (f toks))
~> ...
```

Next time: LL parsing