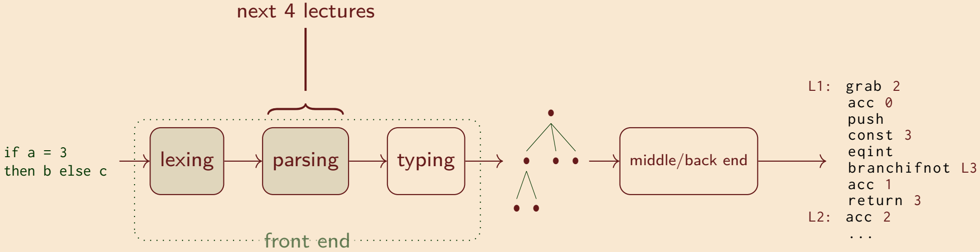
Compiler Construction

Lecture 3: Context-free grammars

Jeremy Yallop
jeremy.yallop@cl.cam.ac.uk

Lent 2023

What is a parser?



What are context-free grammars?

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

Context-Free Grammars (CFGs)

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 $A \to \alpha$

Example CFG

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

where

$$N_1 = \{E\}$$

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

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

NB: P_1 definition is shorthand for

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

Derivations

Notation conventions:

$$\alpha, \beta, \gamma \ldots \in (N \cup T)*$$
$$A, B, C, \ldots \in N$$

Given: $\alpha A \beta$ and a production $A \rightarrow \gamma$ a derivation step is written as

$$\alpha AB \Rightarrow \alpha\gamma\beta$$

$\Rightarrow^{+}$ means one or more derivation steps

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

Parsing
CFGs
Derivations
PDAs
Ambiguity
Top-down & bottom-up

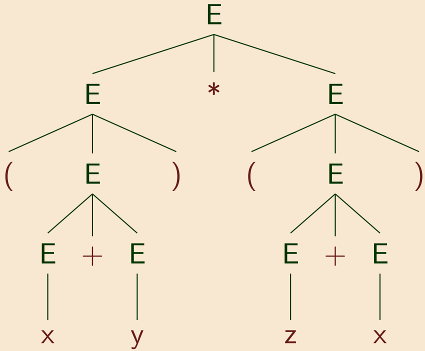
A **leftmost** derivation

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

A **rightmost** derivation

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

Derivation Trees

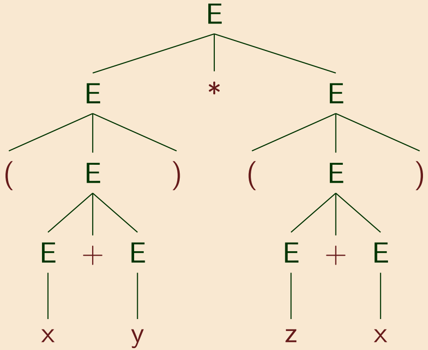


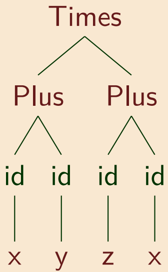
The derivation tree for (x+y) * (z+x). All derivations of this expression will produce the same derivation tree.

Concrete *vs* Abstract Syntax Trees

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

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





L(G) = The Language Generated by Grammar G

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

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

For example, if G has productions

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

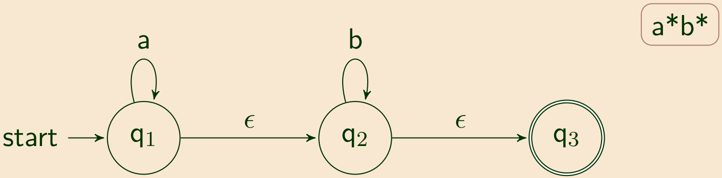
then

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

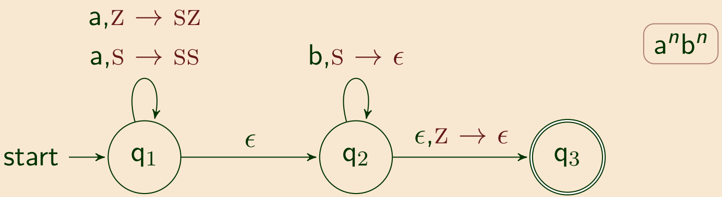
So CFGs can capture more than regular languages.

Pushdown automata

Regular languages are accepted by finite automata:



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



Pushdown Automata (PDAs)

stack symbols Γ

states Q

start state $q_0 \in Q$

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, Z \rangle$$

alphabet T

initial stack symbol $\mathrm{z} \in \Gamma$S

transitions δ:
$\forall q \in Q,$
$a \in (\Sigma \cup \{\epsilon\}),$
$X \in \Gamma,$
$\delta(q, a, X) \subseteq Q \times \Gamma^*$

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

When the machine is
- in state q, *and*
- reading a *and*
- with X on top of the stack,

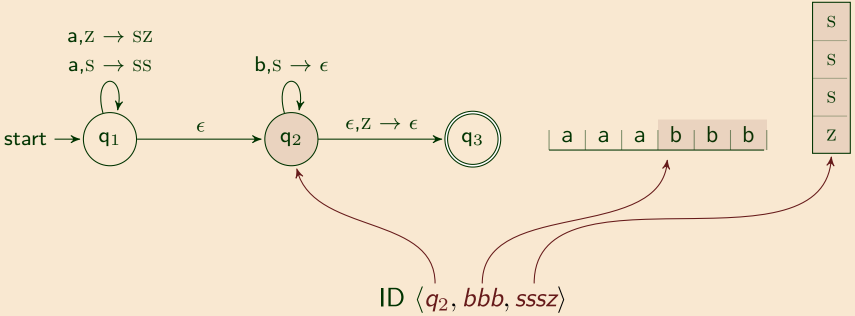
it can
- move to state q' *and*
- replace X with β.

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

Parsing
CFGs
Derivations
PDAs
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).

It denotes the PDA
- in state q
- looking at the first symbol of w
- with α on the stack



Language accepted by a PDA

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

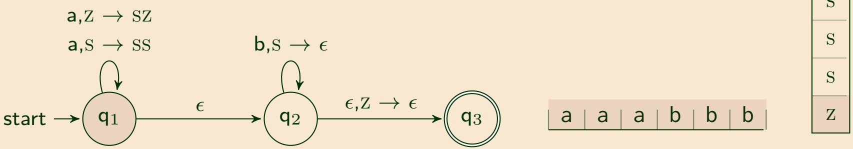
$$\langle q, w, X\alpha \rangle \rightarrow \langle q', w, \beta\alpha \rangle$$

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

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

PDA execution

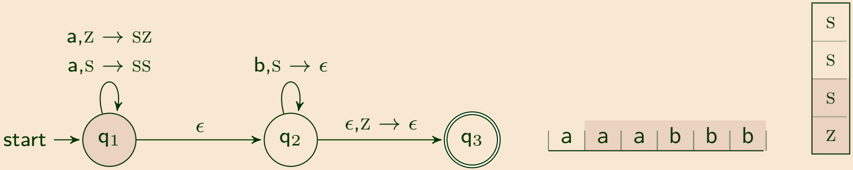
;



$\langle q_1, aaabbb, \mathrm{Z}\rangle$

PDA execution

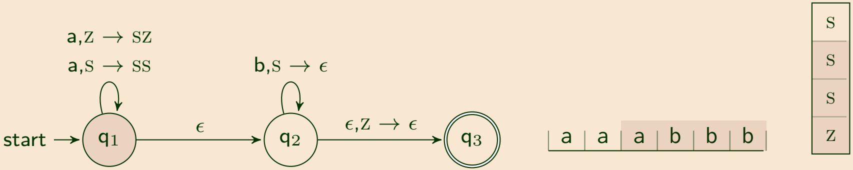
;



$\langle q_1, aaabbb, \mathrm{Z}\rangle$
$\langle q_1, aabbb, \mathrm{SZ}\rangle$

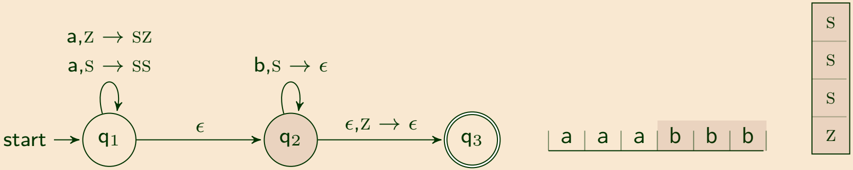
PDA execution

;



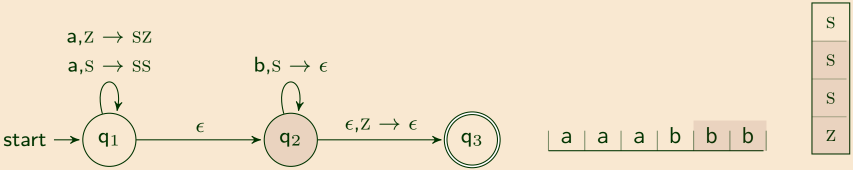
$\langle q_1, aaabbb, \mathrm{Z}\rangle$
$\langle q_1, aabbb, \mathrm{SZ}\rangle$
$\langle q_1, abbb, \mathrm{SSZ}\rangle$

;



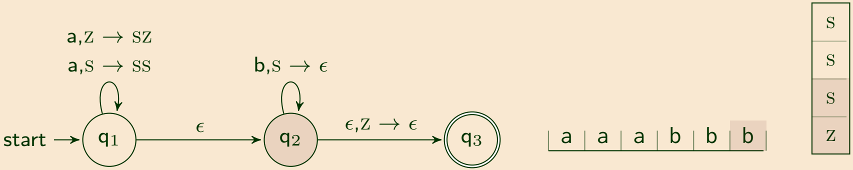
$\langle q_1, aaabbb, \text{Z}\rangle$
$\langle q_1, aabbb, \text{SZ}\rangle$
$\langle q_1, abbb, \text{SSZ}\rangle$
$\langle q_2, bbb, \text{SSSZ}\rangle$

;



$\langle q_1, aaabbb, \mathrm{Z}\rangle$
$\langle q_1, aabbb, \mathrm{SZ}\rangle$
$\langle q_1, abbb, \mathrm{SSZ}\rangle$
$\langle q_2, bbb, \mathrm{SSSZ}\rangle$
$\langle q_2, bb, \mathrm{SSZ}\rangle$

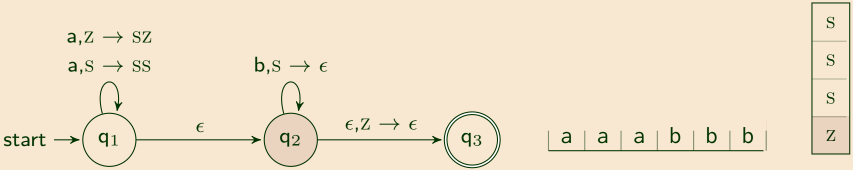
;



$\langle q_1, aaabbb, \mathrm{Z}\rangle$
$\langle q_1, aabbb, \mathrm{SZ}\rangle$
$\langle q_1, abbb, \mathrm{SSZ}\rangle$
$\langle q_2, bbb, \mathrm{SSSZ}\rangle$
$\langle q_2, bb, \mathrm{SSZ}\rangle$
$\langle q_2, b, \mathrm{SZ}\rangle$

PDA execution

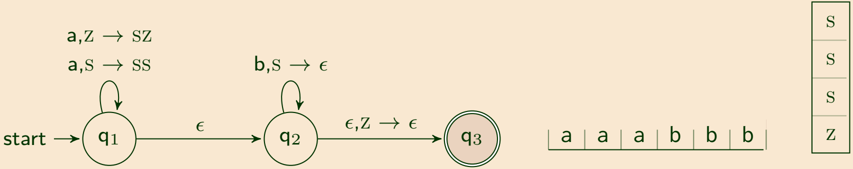
;



$\langle q_1, aaabbb, \mathrm{Z}\rangle$
$\langle q_1, aabbb, \mathrm{SZ}\rangle$
$\langle q_1, abbb, \mathrm{SSZ}\rangle$
$\langle q_2, bbb, \mathrm{SSSZ}\rangle$
$\langle q_2, bb, \mathrm{SSZ}\rangle$
$\langle q_2, b, \mathrm{SZ}\rangle$
$\langle q_2, \epsilon, \mathrm{Z}\rangle$

PDA execution

;



$\langle q_1, aaabbb, \mathrm{Z}\rangle$

$\langle q_1, aabbb, \mathrm{SZ}\rangle$

$\langle q_1, abbb, \mathrm{SSZ}\rangle$

$\langle q_2, bbb, \mathrm{SSSZ}\rangle$

$\langle q_2, bb, \mathrm{SSZ}\rangle$

$\langle q_2, b, \mathrm{SZ}\rangle$

$\langle q_2, \epsilon, \mathrm{Z}\rangle$

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

PDA and CFG facts (without proof)

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

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.

We can often modify the grammar in order to eliminate ambiguity

We can often modify the grammar to eliminate ambiguity.

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

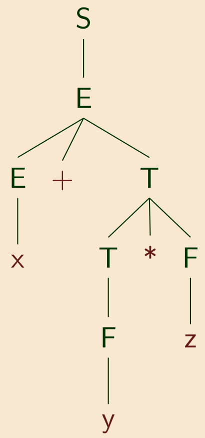
where

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

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

The modified grammar eliminates ambiguity

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



More facts

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

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

2. Checking for ambiguity in an arbitrary CFG is not decidable.
3. Given two grammars $G1$ and $G2$, checking $L(G1) = L(G2)$ is not decidable.

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

Top-down & bottom-up

Two approaches to building stack-based parsing machines

Top-down: attempts a left-most derivation. We'll look at two techniques:

Recursive descent (hand coded)	Predictive parsing (table driven)

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

SLR(1) (Simple LR(1))	LR(1)

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

Recursive Descent Parsing


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

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"
```

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

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

Left recursion & recursive-descent parsing

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
                 | PLUS :: toks → ...
```

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

Eliminating left recursion

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

where

$$\ldots$$

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

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

where

$$\ldots$$

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

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

The stack machine is *implicit in the call stack*

```
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"
```

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

```
[IDENT "x";
 PLUS;
 IDENT "y";
 TIMES;
 IDENT "z"]
```

Evaluation trace:

```
    e toks
⇝   e' (t toks)
⇝   e' (t' (f toks))
⇝   ...
```

Next time: LL parsing