Chapter 4: Context-free Languages

In this chapter, we study context-free grammars and languages. Context-free grammars are used to describe the syntax of programming languages, i.e., to specify parsers of programming languages.

Chapter 4: Context-free Languages

In this chapter, we study context-free grammars and languages. Context-free grammars are used to describe the syntax of programming languages, i.e., to specify parsers of programming languages.

A language is called context-free iff it is generated by a context-free grammar. It will turn out that the set of all context-free languages is a proper superset of the set of all regular languages.

Chapter 4: Context-free Languages

In this chapter, we study context-free grammars and languages. Context-free grammars are used to describe the syntax of programming languages, i.e., to specify parsers of programming languages.

A language is called context-free iff it is generated by a context-free grammar. It will turn out that the set of all context-free languages is a proper superset of the set of all regular languages.

On the other hand, the context-free languages have weaker closure properties than the regular languages, and we won't be able to give algorithms for checking grammar equivalence or minimizing the size of grammars.

4.1: Grammars, Parse Trees and Context-free Languages

In this section, we:

- say what (context-free) grammars are;
- use the notion of a parse tree to say what grammars mean;
- say what it means for a language to be context-free; and
- begin to show how grammars can be processed using Forlan.

(Context-free) Grammars

A *context-free grammar* (or just *grammar*) G consists of:

- a finite set Q_G of symbols (we call the elements of Q_G the *variables* of G);
- an element s_G of Q_G (we call s_G the *start variable* of G); and
- a finite subset P_G of $\{\,(q, x) \mid q \in Q_G \text{ and } x \in \mathbf{Str}\,\}$ (we call the elements of P_G the *productions* of G, and we often write (q, x) as $q \to x$).

In a context where we are only referring to a single grammar, G, we sometimes abbreviate Q_G, s_G and P_G to Q, s and P, respectively.

(Context-free) Grammars

A *context-free grammar* (or just *grammar*) G consists of:

- a finite set Q_G of symbols (we call the elements of Q_G the *variables* of G);
- an element s_G of Q_G (we call s_G the *start variable* of G); and
- a finite subset P_G of $\{\, (q, x) \mid q \in Q_G \text{ and } x \in \mathbf{Str} \,\}$ (we call the elements of P_G the *productions* of G, and we often write (q, x) as $q \to x$).

In a context where we are only referring to a single grammar, G, we sometimes abbreviate Q_G, s_G and P_G to Q, s and P, respectively.

We write **Gram** for the set of all grammars. Since every grammar can be described by a finite sequence of ASCII characters, we have that **Gram** is countably infinite.

Example Grammar

As an example, we can define a grammar G (of arithmetic expressions) as follows:

- $Q_G = \{\mathsf{E}\}$;
- $s_G = \mathsf{E}$;
- $P_G = \{\mathsf{E} \to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E},\ \mathsf{E} \to \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E},\ \mathsf{E} \to \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle,\ \mathsf{E} \to \langle\mathsf{id}\rangle\}$.

E.g., we can read the production $\mathsf{E} \to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E}$ as "an expression can consist of an expression, followed by a $\langle\mathsf{plus}\rangle$ symbol, followed by an expression".

Notation for Grammars

We typically describe a grammar by listing its productions, and grouping productions with identical left-sides into production families. Unless we say otherwise, the grammar's variables are the left-sides of all of its productions, and its start variable is the left-side of its first production.

Thus, our grammar G is

$$\begin{aligned}
\mathsf{E} &\to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E},\\
\mathsf{E} &\to \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E},\\
\mathsf{E} &\to \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle,\\
\mathsf{E} &\to \langle\mathsf{id}\rangle,
\end{aligned}$$

or

$$\mathsf{E} \to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle.$$

Forlan Syntax for Grammars

The Forlan syntax for grammars is very similar to the notation of the preceding slide. E.g., here is how our example grammar can be described in Forlan's syntax:

```
{variables} E {start variable} E
{productions}
E -> E<plus>E; E -> E<times>E; E -> <openPar>E<closPar>;
E -> <id>
```

or

```
{variables} E {start variable} E
{productions}
E -> E<plus>E | E<times>E | <openPar>E<closPar> | <id>
```

Processing Grammars in Forlan

The Forlan module `Gram` defines an abstract type `gram` (in the top-level environment) of grammars as well as a number of functions and constants for processing grammars, including:

```
val input          : string -> gram
val output         : string * gram -> unit
val numVariables   : gram -> int
val numProductions : gram -> int
val equal          : gram * gram -> bool
```

During printing, Forlan merges productions into production families whenever possible.

More on Grammars

The *alphabet of* a grammar G ($\textbf{alphabet}\,G$) is

$$\{\, a \in \textbf{Sym} \mid \text{there are } q, x \text{ such that } q \to x \in P_G \text{ and } a \in \textbf{alphabet}\, x \,\} - Q_G.$$

I.e., $\textbf{alphabet}\,G$ is all of the symbols appearing in the strings of G's productions that aren't variables.

For example, the alphabet of our example grammar G is $\{\langle\text{plus}\rangle, \langle\text{times}\rangle, \langle\text{openPar}\rangle, \langle\text{closPar}\rangle, \langle\text{id}\rangle\}$.

Grammar Alphabets in Forlan

The Forlan module `Gram` defines a function

```
val alphabet : gram -> sym set
```

for calculating the alphabet of a grammar.

E.g., if `gram` of type `gram` is bound to our example grammar G, then Forlan will behave as follows:

```
- val bs = Gram.alphabet gram;
val bs = - : sym set
- SymSet.output("", bs);
<id>, <plus>, <times>, <closPar>, <openPar>
val it = () : unit
```

Parse Trees and Grammar Meaning

We will explain when strings are generated by grammars using the notion of a parse tree. The set $\mathbf{PT}$ of *parse trees* is the least subset of $\mathbf{Tree}(\mathbf{Sym} \cup \{\%\})$ (the set of all $(\mathbf{Sym} \cup \{\%\})$-trees; see Section 1.3) such that:

(1) for all $a \in \mathbf{Sym}$ and $pts \in \mathbf{List}\,\mathbf{PT}$, $(a, pts) \in \mathbf{PT}$; and

Parse Trees and Grammar Meaning

We will explain when strings are generated by grammars using the notion of a parse tree. The set $\mathbf{PT}$ of *parse trees* is the least subset of $\mathbf{Tree}(\mathbf{Sym} \cup \{\%\})$ (the set of all $(\mathbf{Sym} \cup \{\%\})$-trees; see Section 1.3) such that:

(1) for all $a \in \mathbf{Sym}$ and $pts \in \mathbf{List}\,\mathbf{PT}$, $(a, pts) \in \mathbf{PT}$; and

(2) for all $a \in \mathbf{Sym}$, $(a, [(\%, [\,])]) = a(\%) \in \mathbf{PT}$.

Parse Trees and Grammar Meaning

We will explain when strings are generated by grammars using the notion of a parse tree. The set $\mathbf{PT}$ of *parse trees* is the least subset of $\mathbf{Tree}(\mathbf{Sym} \cup \{\%\})$ (the set of all $(\mathbf{Sym} \cup \{\%\})$-trees; see Section 1.3) such that:

(1) for all $a \in \mathbf{Sym}$ and $pts \in \mathbf{List}\,\mathbf{PT}$, $(a, pts) \in \mathbf{PT}$; and

(2) for all $a \in \mathbf{Sym}$, $(a, [(\%, [\,])]) = a(\%) \in \mathbf{PT}$.

Note that the $(\mathbf{Sym} \cup \{\%\})$-tree $\% = (\%, [\,])$ is *not* a parse tree.

Parse Trees and Grammar Meaning

We will explain when strings are generated by grammars using the notion of a parse tree. The set $\mathbf{PT}$ of *parse trees* is the least subset of $\mathbf{Tree}(\mathbf{Sym} \cup \{\%\})$ (the set of all $(\mathbf{Sym} \cup \{\%\})$-trees; see Section 1.3) such that:

(1) for all $a \in \mathbf{Sym}$ and $pts \in \mathbf{List}\,\mathbf{PT}$, $(a, pts) \in \mathbf{PT}$; and

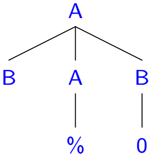
(2) for all $a \in \mathbf{Sym}$, $(a, [(\%, [\,])]) = a(\%) \in \mathbf{PT}$.

Note that the $(\mathbf{Sym} \cup \{\%\})$-tree $\% = (\%, [\,])$ is *not* a parse tree.

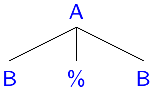
It is easy to see that $\mathbf{PT}$ is countably infinite.

Parse Tree Examples

For example, $\mathsf{A(B, A(\%), B(0))}$, i.e.,



is a parse tree. On the other hand, although $\mathsf{A(B, \%, B)}$, i.e.,



is a $(\mathbf{Sym} \cup \{\%\})$-tree, it's not a parse tree, since it can't be formed using rules (1) and (2).

Principle of Induction on Parse Trees

Since the set **PT** of parse trees is defined inductively, it gives rise to an induction principle.

Theorem 4.1.1 (Principle of Induction on Parse Trees)
Suppose $P(pt)$ *is a property of an element* $pt \in \mathbf{PT}$.

If

(1) *for all* $a \in \mathbf{Sym}$ *and* $trs \in \mathbf{List}\,\mathbf{PT}$, *if* $(\dagger)$ *for all* $i \in [1 : |trs|]$, $P(trs\,i)$, *then* $P((a, trs))$, *and*

then

$$\textit{for all } pt \in \mathbf{PT}, P(pt).$$

We refer to $(\dagger)$ as the inductive hypothesis.

Since the set **PT** of parse trees is defined inductively, it gives rise to an induction principle.

Theorem 4.1.1 (Principle of Induction on Parse Trees)
Suppose $P(pt)$ is a property of an element $pt \in \mathbf{PT}$.

If

(1) *for all $a \in \mathbf{Sym}$ and $trs \in \mathbf{List\,PT}$, if $(\dagger)$ for all $i \in [1 : |trs|]$, $P(trs\, i)$, then $P((a, trs))$, and*

(2) *for all $a \in \mathbf{Sym}$, $P(a(\%))$,*

then

$$\text{for all } pt \in \mathbf{PT}, P(pt).$$

We refer to $(\dagger)$ as the inductive hypothesis.

The Yield of a Parse Tree

We define the yield of a parse tree, as follows. The function $\mathbf{yield} \in \mathbf{PT} \to \mathbf{Str}$ is defined by structural recursion:

- for all $a \in \mathbf{Sym}$, $\mathbf{yield}\, a =$
- for all $q \in \mathbf{Sym}$, $n \in \mathbb{N} - \{0\}$ and $pt_1, \ldots, pt_n \in \mathbf{PT}$, $\mathbf{yield}(q(pt_1, \ldots, pt_n)) =$
- for all $q \in \mathbf{Sym}$, $\mathbf{yield}(q(\%)) =$

We say that w is the *yield of* pt iff $w = \mathbf{yield}\, pt$.

The Yield of a Parse Tree

We define the yield of a parse tree, as follows. The function **yield** $\in$ **PT** $\to$ **Str** is defined by structural recursion:

- for all $a \in$ **Sym**, **yield** $a = a$;
- for all $q \in$ **Sym**, $n \in \mathbb{N} - \{0\}$ and $pt_1, \ldots, pt_n \in$ **PT**, **yield**$(q(pt_1, \ldots, pt_n)) =$
- for all $q \in$ **Sym**, **yield**$(q(\%)) =$

We say that w is the *yield of* pt iff $w =$ **yield** pt.

The Yield of a Parse Tree

We define the yield of a parse tree, as follows. The function $\mathbf{yield} \in \mathbf{PT} \to \mathbf{Str}$ is defined by structural recursion:

- for all $a \in \mathbf{Sym}$, $\mathbf{yield}\, a = a$;
- for all $q \in \mathbf{Sym}$, $n \in \mathbb{N} - \{0\}$ and $pt_1, \ldots, pt_n \in \mathbf{PT}$, $\mathbf{yield}(q(pt_1, \ldots, pt_n)) = \mathbf{yield}\, pt_1 \cdots \mathbf{yield}\, pt_n$; and
- for all $q \in \mathbf{Sym}$, $\mathbf{yield}(q(\%)) =$

We say that w is the *yield of* pt iff $w = \mathbf{yield}\, pt$.

The Yield of a Parse Tree

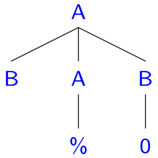
We define the yield of a parse tree, as follows. The function $\textbf{yield} \in \textbf{PT} \to \textbf{Str}$ is defined by structural recursion:

- for all $a \in \textbf{Sym}$, $\textbf{yield}\, a = a$;
- for all $q \in \textbf{Sym}$, $n \in \mathbb{N} - \{0\}$ and $pt_1, \ldots, pt_n \in \textbf{PT}$, $\textbf{yield}(q(pt_1, \ldots, pt_n)) = \textbf{yield}\, pt_1 \cdots \textbf{yield}\, pt_n$; and
- for all $q \in \textbf{Sym}$, $\textbf{yield}(q(\%)) = \%$.

We say that w is the *yield of* pt iff $w = \textbf{yield}\, pt$.

Yield Example

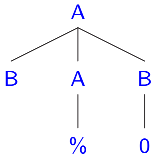
For example, the yield of



is

Yield Example

For example, the yield of



is

$$\mathbf{yield}\,\mathsf{B}\,\mathbf{yield}(A(\%))\,\mathbf{yield}(\mathsf{B}(0)) = \mathsf{B}\,\%\,\mathbf{yield}\,0 = \mathsf{B}\%0 = \mathsf{B}0.$$

Validity of Parse Trees for Grammars

We say when a parse tree is valid for a grammar G as follows. Define a function $\mathbf{valid}_G \in \mathbf{PT} \to \mathbf{Bool}$ by structural recursion:

- for all $a \in \mathbf{Sym}$, $\mathbf{valid}_G\, a =$
- for all $q \in \mathbf{Sym}$, $n \in \mathbb{N} - \{0\}$ and $pt_1, \ldots, pt_n \in \mathbf{PT}$,

$$\mathbf{valid}_G(q(pt_1, \ldots, pt_n))$$

$$=$$

- for all $q \in \mathbf{Sym}$, $\mathbf{valid}_G(q(\%)) =$

We say that pt is *valid for* G iff $\mathbf{valid}_G\, pt = \mathbf{true}$. We often abbreviate $\mathbf{valid}_G$ to $\mathbf{valid}$.

We can prove by induction on parse trees that, for all parse trees pt, if pt is valid for G, then $\mathbf{rootLabel}\, pt \in Q_G \cup \mathbf{alphabet}\, G$ and $\mathbf{yield}\, pt \in (Q_G \cup \mathbf{alphabet}\, G)^*$.

Validity of Parse Trees for Grammars

We say when a parse tree is valid for a grammar G as follows. Define a function $\mathbf{valid}_G \in \mathbf{PT} \to \mathbf{Bool}$ by structural recursion:

- for all $a \in \mathbf{Sym}$, $\mathbf{valid}_G\, a = a \in \mathbf{alphabet}\, G$ **or** $a \in Q_G$;
- for all $q \in \mathbf{Sym}$, $n \in \mathbb{N} - \{0\}$ and $pt_1, \ldots, pt_n \in \mathbf{PT}$,

$$\mathbf{valid}_G(q(pt_1, \ldots, pt_n))$$
$$=$$

- for all $q \in \mathbf{Sym}$, $\mathbf{valid}_G(q(\%)) =$

We say that pt is *valid for* G iff $\mathbf{valid}_G\, pt = \mathbf{true}$. We often abbreviate $\mathbf{valid}_G$ to $\mathbf{valid}$.

We can prove by induction on parse trees that, for all parse trees pt, if pt is valid for G, then $\mathbf{rootLabel}\, pt \in Q_G \cup \mathbf{alphabet}\, G$ and $\mathbf{yield}\, pt \in (Q_G \cup \mathbf{alphabet}\, G)^*$.

Validity of Parse Trees for Grammars

We say when a parse tree is valid for a grammar G as follows. Define a function $\mathbf{valid}_G \in \mathbf{PT} \to \mathbf{Bool}$ by structural recursion:

- for all $a \in \mathbf{Sym}$, $\mathbf{valid}_G\, a = a \in \mathbf{alphabet}\, G\ \mathbf{or}\ a \in Q_G$;
- for all $q \in \mathbf{Sym}$, $n \in \mathbb{N} - \{0\}$ and $pt_1, \ldots, pt_n \in \mathbf{PT}$,

$$\begin{aligned} &\mathbf{valid}_G(q(pt_1, \ldots, pt_n)) \\ = {}& q \to \mathbf{rootLabel}\, pt_1 \cdots \mathbf{rootLabel}\, pt_n \in P_G\ \mathbf{and} \end{aligned}$$

- for all $q \in \mathbf{Sym}$, $\mathbf{valid}_G(q(\%)) =$

We say that pt is *valid for G* iff $\mathbf{valid}_G\, pt = \mathbf{true}$. We often abbreviate $\mathbf{valid}_G$ to $\mathbf{valid}$.

We can prove by induction on parse trees that, for all parse trees pt, if pt is valid for G, then $\mathbf{rootLabel}\, pt \in Q_G \cup \mathbf{alphabet}\, G$ and $\mathbf{yield}\, pt \in (Q_G \cup \mathbf{alphabet}\, G)^*$.

Validity of Parse Trees for Grammars

We say when a parse tree is valid for a grammar G as follows. Define a function $\mathbf{valid}_G \in \mathbf{PT} \to \mathbf{Bool}$ by structural recursion:

- for all $a \in \mathbf{Sym}$, $\mathbf{valid}_G\, a = a \in \mathbf{alphabet}\, G\ \mathbf{or}\ a \in Q_G$;
- for all $q \in \mathbf{Sym}$, $n \in \mathbb{N} - \{0\}$ and $pt_1, \ldots, pt_n \in \mathbf{PT}$,

$$\begin{aligned} &\mathbf{valid}_G(q(pt_1, \ldots, pt_n)) \\ =\ & q \to \mathbf{rootLabel}\, pt_1 \cdots \mathbf{rootLabel}\, pt_n \in P_G\ \mathbf{and} \\ & \mathbf{valid}_G\, pt_1\ \mathbf{and} \cdots \mathbf{and}\ \mathbf{valid}_G\, pt_n; \text{ and} \end{aligned}$$

- for all $q \in \mathbf{Sym}$, $\mathbf{valid}_G(q(\%)) =$

We say that pt is *valid for* G iff $\mathbf{valid}_G\, pt = \mathbf{true}$. We often abbreviate $\mathbf{valid}_G$ to $\mathbf{valid}$.

We can prove by induction on parse trees that, for all parse trees pt, if pt is valid for G, then $\mathbf{rootLabel}\, pt \in Q_G \cup \mathbf{alphabet}\, G$ and $\mathbf{yield}\, pt \in (Q_G \cup \mathbf{alphabet}\, G)^*$.

Validity of Parse Trees for Grammars

We say when a parse tree is valid for a grammar G as follows. Define a function $\mathbf{valid}_G \in \mathbf{PT} \to \mathbf{Bool}$ by structural recursion:

- for all $a \in \mathbf{Sym}$, $\mathbf{valid}_G\, a = a \in \mathbf{alphabet}\, G$ **or** $a \in Q_G$;
- for all $q \in \mathbf{Sym}$, $n \in \mathbb{N} - \{0\}$ and $pt_1, \ldots, pt_n \in \mathbf{PT}$,

$$\begin{aligned} &\mathbf{valid}_G(q(pt_1, \ldots, pt_n)) \\ = {}& q \to \mathbf{rootLabel}\, pt_1 \cdots \mathbf{rootLabel}\, pt_n \in P_G \ \mathbf{and} \\ &\mathbf{valid}_G\, pt_1 \ \mathbf{and} \ \cdots \ \mathbf{and}\ \mathbf{valid}_G\, pt_n; \text{ and} \end{aligned}$$

- for all $q \in \mathbf{Sym}$, $\mathbf{valid}_G(q(\%)) = q \to \% \in P_G$.

We say that pt is *valid for G* iff $\mathbf{valid}_G\, pt = \mathbf{true}$. We often abbreviate $\mathbf{valid}_G$ to **valid**.

We can prove by induction on parse trees that, for all parse trees pt, if pt is valid for G, then $\mathbf{rootLabel}\, pt \in Q_G \cup \mathbf{alphabet}\, G$ and $\mathbf{yield}\, pt \in (Q_G \cup \mathbf{alphabet}\, G)^*$.

Parse Tree Validity Examples

Suppose G is the grammar

$$\begin{aligned} \mathsf{A} &\to \mathsf{BAB} \mid \%, \\ \mathsf{B} &\to \mathsf{0} \end{aligned}$$

(by convention, its variables are A and B and its start variable is A). Let's see why the parse tree $\mathsf{A}(\mathsf{B}, \mathsf{A}(\%), \mathsf{B}(\mathsf{0}))$ is valid for G.

Parse Tree Validity Examples

Suppose G is the grammar

$$\mathsf{A} \to \mathsf{BAB} \mid \%,$$
$$\mathsf{B} \to \mathsf{0}$$

(by convention, its variables are A and B and its start variable is A). Let's see why the parse tree $\mathsf{A}(\mathsf{B}, \mathsf{A}(\%), \mathsf{B}(\mathsf{0}))$ is valid for G.

- Since $\mathsf{A} \to \mathsf{BAB} \in P_G$ and the concatenation of the root labels of the children B, $\mathsf{A}(\%)$ and $\mathsf{B}(\mathsf{0})$ is BAB, the overall tree will be valid for G if these children are valid for G.

Parse Tree Validity Examples

Suppose G is the grammar

$$\mathsf{A} \rightarrow \mathsf{BAB} \mid \%, \\ \mathsf{B} \rightarrow \mathsf{0}$$

(by convention, its variables are A and B and its start variable is A). Let's see why the parse tree $\mathsf{A}(\mathsf{B}, \mathsf{A}(\%), \mathsf{B}(\mathsf{0}))$ is valid for G.

- Since $\mathsf{A} \rightarrow \mathsf{BAB} \in P_G$ and the concatenation of the root labels of the children B, $\mathsf{A}(\%)$ and $\mathsf{B}(\mathsf{0})$ is BAB, the overall tree will be valid for G if these children are valid for G.
- The parse tree B is valid for G since $\mathsf{B} \in Q_G$.

Parse Tree Validity Examples

Suppose G is the grammar

$$\mathsf{A} \to \mathsf{BAB} \mid \%,$$
$$\mathsf{B} \to \mathsf{0}$$

(by convention, its variables are A and B and its start variable is A). Let's see why the parse tree $\mathsf{A(B, A(\%), B(0))}$ is valid for G.

- Since $\mathsf{A} \to \mathsf{BAB} \in P_G$ and the concatenation of the root labels of the children B, A(%) and B(0) is BAB, the overall tree will be valid for G if these children are valid for G.
- The parse tree B is valid for G since $\mathsf{B} \in Q_G$.
- Since $\mathsf{A} \to \% \in P_G$, the parse tree A(%) is valid for G.

Validity Examples

- Since $\mathsf{B} \to 0 \in P_G$ and the root label of the child 0 is 0, the parse tree $\mathsf{B}(0)$ will be valid for G if the child 0 is valid for G.

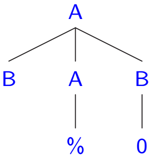
Validity Examples

- Since $\mathsf{B} \to \mathsf{0} \in P_G$ and the root label of the child $\mathsf{0}$ is $\mathsf{0}$, the parse tree $\mathsf{B(0)}$ will be valid for G if the child $\mathsf{0}$ is valid for G.
- The child $\mathsf{0}$ is valid for G since $\mathsf{0} \in$ **alphabet** G.

Validity Examples

- Since $\mathsf{B} \to \mathsf{0} \in P_G$ and the root label of the child $\mathsf{0}$ is $\mathsf{0}$, the parse tree $\mathsf{B(0)}$ will be valid for G if the child $\mathsf{0}$ is valid for G.
- The child $\mathsf{0}$ is valid for G since $\mathsf{0} \in$ **alphabet** G.

Thus, we have that



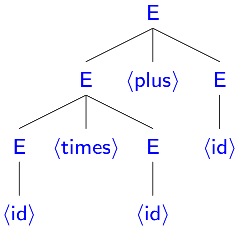
is valid for G.

Validity Examples

Suppose G is our grammar of arithmetic expressions

$$\mathsf{E} \rightarrow \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle.$$

Then the parse tree



is valid for G.

The Meaning of Grammars

Suppose G is a grammar, $w \in \textbf{Str}$ and $a \in \textbf{Sym}$. We say that w *is parsable from* a *using* G iff there is a parse tree pt such that:

- pt is valid for G;
- a is the root label of pt; and
- the yield of pt is w.

Thus we will have that $w \in (Q_G \cup \textbf{alphabet}\, G)^*$, and either $a \in Q_G$ or $[a] = w$.

The Meaning of Grammars

Suppose G is a grammar, $w \in \mathbf{Str}$ and $a \in \mathbf{Sym}$. We say that w *is parsable from a using G* iff there is a parse tree pt such that:

- pt is valid for G;
- a is the root label of pt; and
- the yield of pt is w.

Thus we will have that $w \in (Q_G \cup \mathbf{alphabet}\, G)^*$, and either $a \in Q_G$ or $[a] = w$.

We say that a string w *is generated from* a variable $q \in Q_G$ *using G* iff $w \in (\mathbf{alphabet}\, G)^*$ and w is parsable from q.

The Meaning of Grammars

Suppose G is a grammar, $w \in \textbf{Str}$ and $a \in \textbf{Sym}$. We say that w *is parsable from* a *using* G iff there is a parse tree pt such that:

- pt is valid for G;
- a is the root label of pt; and
- the yield of pt is w.

Thus we will have that $w \in (Q_G \cup \textbf{alphabet}\, G)^*$, and either $a \in Q_G$ or $[a] = w$.

We say that a string w *is generated from* a variable $q \in Q_G$ *using* G iff $w \in (\textbf{alphabet}\, G)^*$ and w is parsable from q.

And, we say that a string w *is generated by* a grammar G iff w is generated from s_G using G.

The Meaning of Grammars

The *language generated by* a grammar G ($L(G)$) is

$$\{\, w \in \mathbf{Str} \mid w \text{ is generated by } G \,\}.$$

Proposition 4.1.3
For all grammars G, $\mathbf{alphabet}(L(G)) \subseteq \mathbf{alphabet}\, G$.

Grammar Meaning Examples

Let G be the example grammar

$$\begin{aligned} &\mathsf{A} \rightarrow \mathsf{BAB} \mid \%, \\ &\mathsf{B} \rightarrow \mathsf{0}. \end{aligned}$$

Then 00 is generated by G since

Grammar Meaning Examples

Let G be the example grammar

$$\begin{aligned} &\mathsf{A} \to \mathsf{BAB} \mid \%, \\ &\mathsf{B} \to \mathsf{0}. \end{aligned}$$

Then $\mathsf{00}$ is generated by G since $\mathsf{00} \in \{\mathsf{0}\}^* = (\textbf{alphabet}\ G)^*$ and the parse tree

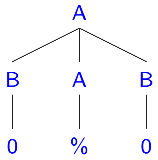
is valid for G, has $s_G = \mathsf{A}$ as its root label, and has $\mathsf{00}$ as its yield.

Grammar Meaning Examples

Let G be the example grammar

$$\mathsf{A} \to \mathsf{BAB} \mid \%,$$
$$\mathsf{B} \to \mathsf{0}.$$

Then $\mathsf{00}$ is generated by G since $\mathsf{00} \in \{\mathsf{0}\}^* = (\textbf{alphabet}\ G)^*$ and the parse tree



is valid for G, has $s_G = \mathsf{A}$ as its root label, and has $\mathsf{00}$ as its yield.

Grammar Meaning Examples

Suppose G is our grammar of arithmetic expressions

$$\mathsf{E} \rightarrow \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle.$$

Then $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle$ is generated by G since $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle \in (\textbf{alphabet}\, G)^*$ and the parse tree

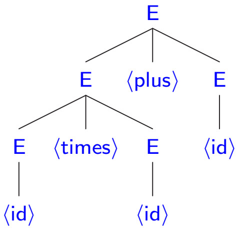
is valid for G, has $s_G = \mathsf{E}$ as its root label, and has $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle$ as its yield.

Grammar Meaning Examples

Suppose G is our grammar of arithmetic expressions

$$\mathsf{E} \to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle.$$

Then $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle$ is generated by G since $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle \in (\textbf{alphabet}\ G)^*$ and the parse tree



is valid for G, has $s_G = \mathsf{E}$ as its root label, and has $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle$ as its yield.

Context-free Languages

A language L is *context-free* iff $L = L(G)$ for some $G \in \mathbf{Gram}$. We define

$$\begin{aligned}\mathbf{CFLan} &= \{\, L(G) \mid G \in \mathbf{Gram} \,\} \\ &= \{\, L \in \mathbf{Lan} \mid L \text{ is context-free} \,\}.\end{aligned}$$

Since $\{0^0\}$, $\{0^1\}$, $\{0^2\}$, . . . , are all context-free languages, we have that $\mathbf{CFLan}$ is infinite. But, since $\mathbf{Gram}$ is countably infinite, it follows that $\mathbf{CFLan}$ is also countably infinite.

Since $\mathbf{Lan}$ is uncountable, it follows that $\mathbf{CFLan} \subsetneq \mathbf{Lan}$, i.e., there are non-context-free languages. Later, we will see that $\mathbf{RegLan} \subsetneq \mathbf{CFLan}$.

Equivalence of Grammars

We say that grammars G and H are *equivalent* iff $L(G) = L(H)$. In other words, G and H are equivalent iff G and H generate the same language.

We define a relation $\approx$ on **Gram** by: $G \approx H$ iff G and H are equivalent. It is easy to see that $\approx$ is reflexive on **Gram**, symmetric and transitive.

Processing Parse Trees in Forlan

The Forlan module PT defines an abstract type pt of parse trees (in the top-level environment) along with some functions for processing parse trees:

```
val input     : string -> pt
val output    : string * pt -> unit
val height    : pt -> int
val size      : pt -> int
val equal     : pt * pt -> bool
val rootLabel : pt -> sym
val yield     : pt -> str
```

The Forlan syntax for parse trees is simply the linear syntax that we've been using in this section.

Graphical Editor for Parse Trees

The Java program JForlan, can be used to view and edit parse trees. It can be invoked directly, or run via Forlan. See the Forlan website for more information.

More Parse Tree Processing

The Forlan module `Gram` also defines the functions

```
val checkPT : gram -> pt -> unit
val validPT : gram -> pt -> bool
```

The function `checkPT` is used to check whether a parse tree is valid for a grammar; if the answer is "no", it explains why not and raises an exception; otherwise it simply returns `()`.

The function `validPT` checks whether a parse tree is valid for a grammar, silently returning `true` if it is, and silently returning `false` if it isn't.

Forlan Examples

Suppose the identifier `gram` of type `gram` is bound to the grammar

$$\mathsf{A} \to \mathsf{BAB} \mid \%,$$
$$\mathsf{B} \to 0.$$

And, suppose that the identifier `gram'` of type `gram` is bound to our grammar of arithmetic expressions

$$\mathsf{E} \to \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle.$$

Forlan Examples

Here are some examples of how we can process parse trees using Forlan:

```
- val pt = PT.input "";
@ A(B, A(%), B(0))
@ .
val pt = - : pt
- Sym.output("", PT.rootLabel pt);
A
val it = () : unit
- Str.output("", PT.yield pt);
B0
val it = () : unit
- Gram.validPT gram pt;
val it = true : bool
```

Forlan Examples

```
- val pt' = PT.input "";
@ E(E(E(<id>), <times>, E(<id>)), <plus>, E(<id>))
@ .
val pt' = - : pt
- Sym.output("", PT.rootLabel pt');
E
val it = () : unit
- Str.output("", PT.yield pt');
<id><times><id><plus><id>
val it = () : unit
- Gram.validPT gram' pt';
val it = true : bool
```

Forlan Examples

```
- Gram.checkPT gram pt';
invalid production: "E -> E<plus>E"

uncaught exception Error
- Gram.checkPT gram' pt;
invalid production: "A -> BAB"

uncaught exception Error
- PT.input "";
@ A(B,%,B)
@ .
line 1: "%" unexpected

uncaught exception Error
```

Grammar Synthesis

We conclude this section with a grammar synthesis example. Suppose $X = \{ 0^n 1^m 2^m 3^n \mid n, m \in \mathbb{N} \}$. How can we find a grammar G such that $L(G) = X$?

The key is to think of generating the strings of X

Grammar Synthesis

We conclude this section with a grammar synthesis example. Suppose $X = \{\, 0^n 1^m 2^m 3^n \mid n, m \in \mathbb{N} \,\}$. How can we find a grammar G such that $L(G) = X$?

The key is to think of generating the strings of X from the outside in,

Grammar Synthesis

We conclude this section with a grammar synthesis example. Suppose $X = \{\, 0^n 1^m 2^m 3^n \mid n, m \in \mathbb{N} \,\}$. How can we find a grammar G such that $L(G) = X$?

The key is to think of generating the strings of X from the outside in, in two phases.

In the first phase, one generates pairs of 0's and 3's, and, in the second phase, one generates pairs of 1's and 2's. E.g., a string could be formed in the following stages:

Grammar Synthesis

We conclude this section with a grammar synthesis example. Suppose $X = \{\, 0^n 1^m 2^m 3^n \mid n, m \in \mathbb{N} \,\}$. How can we find a grammar G such that $L(G) = X$?

The key is to think of generating the strings of X from the outside in, in two phases.

In the first phase, one generates pairs of 0's and 3's, and, in the second phase, one generates pairs of 1's and 2's. E.g., a string could be formed in the following stages:

$$0 \quad\; 3,$$

Grammar Synthesis

We conclude this section with a grammar synthesis example. Suppose $X = \{\, 0^n 1^m 2^m 3^n \mid n, m \in \mathbb{N} \,\}$. How can we find a grammar G such that $L(G) = X$?

The key is to think of generating the strings of X from the outside in, in two phases.

In the first phase, one generates pairs of 0's and 3's, and, in the second phase, one generates pairs of 1's and 2's. E.g., a string could be formed in the following stages:

$$
\begin{array}{cc}
0 & 3, \\
00 & 33,
\end{array}
$$

Grammar Synthesis

We conclude this section with a grammar synthesis example. Suppose $X = \{\, 0^n 1^m 2^m 3^n \mid n, m \in \mathbb{N} \,\}$. How can we find a grammar G such that $L(G) = X$?

The key is to think of generating the strings of X from the outside in, in two phases.

In the first phase, one generates pairs of 0's and 3's, and, in the second phase, one generates pairs of 1's and 2's. E.g., a string could be formed in the following stages:

$$
\begin{array}{c}
0 \quad\;\; 3, \\
00 \quad 33, \\
001233.
\end{array}
$$

Grammar Synthesis

This analysis leads us to the grammar

$$A \rightarrow$$

where A corresponds to the first phase, and B to the second phase.

Grammar Synthesis

This analysis leads us to the grammar

$$\begin{aligned} A &\to 0A3, \\ A &\to \end{aligned}$$

where A corresponds to the first phase, and B to the second phase.

Grammar Synthesis

This analysis leads us to the grammar

$$
\begin{aligned}
\mathsf{A} &\to \mathsf{0A3}, \\
\mathsf{A} &\to \mathsf{B}, \\
\mathsf{B} &\to
\end{aligned}
$$

where A corresponds to the first phase, and B to the second phase.

Grammar Synthesis

This analysis leads us to the grammar

$$
\begin{aligned}
\mathsf{A} &\rightarrow \mathsf{0A3}, \\
\mathsf{A} &\rightarrow \mathsf{B}, \\
\mathsf{B} &\rightarrow \mathsf{1B2}, \\
\mathsf{B} &\rightarrow
\end{aligned}
$$

where A corresponds to the first phase, and B to the second phase.

Grammar Synthesis

This analysis leads us to the grammar

$$
\begin{aligned}
\mathsf{A} &\to \mathsf{0A3}, \\
\mathsf{A} &\to \mathsf{B}, \\
\mathsf{B} &\to \mathsf{1B2}, \\
\mathsf{B} &\to \%,
\end{aligned}
$$

where A corresponds to the first phase, and B to the second phase.

Grammar Synthesis

For example, here is how the string 001233 may be parsed using G: