4.6: Ambiguity of Grammars

In this section, we say what it means for a grammar to be ambiguous. We also give a straightforward method for disambiguating grammars for languages with operators of various precedences and associativities, and consider an efficient parsing algorithm for such disambiguated grammars.

Motivating Example

Suppose G is our grammar of arithmetic expressions:

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

Question: are there multiple ways of parsing the string $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle$ according to this grammar?

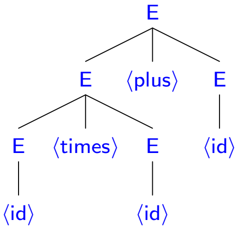
Answer:

Suppose G is our grammar of arithmetic expressions:

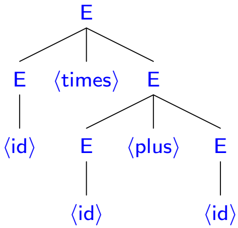
$$\mathsf{E} \rightarrow \mathsf{E}\langle\mathsf{plus}\rangle\mathsf{E} \mid \mathsf{E}\langle\mathsf{times}\rangle\mathsf{E} \mid \langle\mathsf{openPar}\rangle\mathsf{E}\langle\mathsf{closPar}\rangle \mid \langle\mathsf{id}\rangle.$$

Question: are there multiple ways of parsing the string $\langle\mathsf{id}\rangle\langle\mathsf{times}\rangle\langle\mathsf{id}\rangle\langle\mathsf{plus}\rangle\langle\mathsf{id}\rangle$ according to this grammar?

Answer: Yes:



(pt_1)



(pt_2)

Definition

In pt_1, multiplication has higher precedence than addition; in pt_2, the situation is reversed. Because there are multiple ways of parsing this string, we say that our grammar is “ambiguous”.

A grammar G is *ambiguous* iff there is a $w \in (\textbf{alphabet}\, G)^*$ such that w is the yield of multiple valid parse trees for G whose root labels are s_G; otherwise, G is *unambiguous*.

Examples

The grammar

$$\mathsf{A} \rightarrow \% \mid \mathsf{0A1A} \mid \mathsf{1A0A}$$

is a grammar generating all elements of $\{0, 1\}^*$ with a **diff** of 0, for the **diff** function such that $\textbf{diff}\, 0 = -1$ and $\textbf{diff}\, 1 = 1$.

It is

Examples

The grammar

$$\mathsf{A} \rightarrow \% \mid 0\mathsf{A}1\mathsf{A} \mid 1\mathsf{A}0\mathsf{A}$$

is a grammar generating all elements of $\{0,1\}^*$ with a **diff** of 0, for the **diff** function such that $\textbf{diff}\,0 = -1$ and $\textbf{diff}\,1 = 1$.

It is ambiguous as, e.g., 0101 can be parsed as 0%1(01) or 0(10)1%.

Examples

The grammar

$$\mathsf{A} \to \% \mid \mathsf{0A1A} \mid \mathsf{1A0A}$$

is a grammar generating all elements of $\{0, 1\}^*$ with a **diff** of 0, for the **diff** function such that **diff** $0 = -1$ and **diff** $1 = 1$.

It is ambiguous as, e.g., 0101 can be parsed as 0%1(01) or 0(10)1%.

In Section 4.5, we saw another grammar for this language:

$$\mathsf{A} \to \% \mid \mathsf{0BA} \mid \mathsf{1CA},$$
$$\mathsf{B} \to \mathsf{1} \mid \mathsf{0BB},$$
$$\mathsf{C} \to \mathsf{0} \mid \mathsf{1CC},$$

which turns out to be

Examples

The grammar

$$\mathsf{A} \to \% \mid \mathsf{0A1A} \mid \mathsf{1A0A}$$

is a grammar generating all elements of $\{0,1\}^*$ with a **diff** of 0, for the **diff** function such that **diff** $0 = -1$ and **diff** $1 = 1$.

It is ambiguous as, e.g., 0101 can be parsed as 0%1(01) or 0(10)1%.

In Section 4.5, we saw another grammar for this language:

$$\mathsf{A} \to \% \mid \mathsf{0BA} \mid \mathsf{1CA},$$
$$\mathsf{B} \to \mathsf{1} \mid \mathsf{0BB},$$
$$\mathsf{C} \to \mathsf{0} \mid \mathsf{1CC},$$

which turns out to be unambiguous.

Disambiguating Grammars of Operators

Not every ambiguous grammar can be turned into an equivalent unambiguous one. However, we can use a simple technique to disambiguate our grammar of arithmetic expressions, and this technique works for many commonly occurring grammars involving operators of various precedences and associativities.

Since there are two binary operators in our language of arithmetic expressions, we have to decide:

Disambiguating Grammars of Operators

Not every ambiguous grammar can be turned into an equivalent unambiguous one. However, we can use a simple technique to disambiguate our grammar of arithmetic expressions, and this technique works for many commonly occurring grammars involving operators of various precedences and associativities.

Since there are two binary operators in our language of arithmetic expressions, we have to decide:

- whether multiplication has higher or lower precedence than addition; and

Disambiguating Grammars of Operators

Not every ambiguous grammar can be turned into an equivalent unambiguous one. However, we can use a simple technique to disambiguate our grammar of arithmetic expressions, and this technique works for many commonly occurring grammars involving operators of various precedences and associativities.

Since there are two binary operators in our language of arithmetic expressions, we have to decide:

- whether multiplication has higher or lower precedence than addition; and
- whether multiplication and addition are left or right associative.

Disambiguating Grammars of Operators

Not every ambiguous grammar can be turned into an equivalent unambiguous one. However, we can use a simple technique to disambiguate our grammar of arithmetic expressions, and this technique works for many commonly occurring grammars involving operators of various precedences and associativities.

Since there are two binary operators in our language of arithmetic expressions, we have to decide:

- whether multiplication has higher or lower precedence than addition; and
- whether multiplication and addition are left or right associative.

As usual, we'll make multiplication have higher precedence than addition, and let addition and multiplication be left associative.

Example Disambiguation

As a first step towards disambiguating our grammar, we can form a new grammar with the three variables: E (expressions), T (terms) and F (factors), start variable E and productions:

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

The idea is that the lowest precedence operator "lives" at the highest level of the grammar, that the highest precedence operator lives at the middle level of the grammar, and that the basic expressions, including the parenthesized expressions, live at the lowest level of the grammar.

Example Disambiguation

Now, there is only one way to parse the string $\langle \text{id} \rangle \langle \text{times} \rangle \langle \text{id} \rangle \langle \text{plus} \rangle \langle \text{id} \rangle$, since, if we begin by using the production $\text{E} \rightarrow \text{T}$, our yield will only include a $\langle \text{plus} \rangle$ if this symbol occurs within parentheses.

If we had more levels of precedence in our language, we would simply add more levels to our grammar.

Example Disambiguation

On the other hand, there are still two ways of parsing the string $\langle \mathsf{id} \rangle \langle \mathsf{plus} \rangle \langle \mathsf{id} \rangle \langle \mathsf{plus} \rangle \langle \mathsf{id} \rangle$: with left associativity or right associativity.

To finish disambiguating our grammar, we must break the symmetry of the right-sides of the productions

$$\mathsf{E} \to \mathsf{E} \langle \mathsf{plus} \rangle \mathsf{E},$$
$$\mathsf{T} \to \mathsf{T} \langle \mathsf{times} \rangle \mathsf{T},$$

Example Disambiguation

On the other hand, there are still two ways of parsing the string $\langle\text{id}\rangle\langle\text{plus}\rangle\langle\text{id}\rangle\langle\text{plus}\rangle\langle\text{id}\rangle$: with left associativity or right associativity.

To finish disambiguating our grammar, we must break the symmetry of the right-sides of the productions

$$\text{E} \to \text{E}\langle\text{plus}\rangle\text{E},$$
$$\text{T} \to \text{T}\langle\text{times}\rangle\text{T},$$

turning one of the E's into T, and one of the T's into F. To make our operators be left associative, we must

Example Disambiguation

On the other hand, there are still two ways of parsing the string $\langle\text{id}\rangle\langle\text{plus}\rangle\langle\text{id}\rangle\langle\text{plus}\rangle\langle\text{id}\rangle$: with left associativity or right associativity.

To finish disambiguating our grammar, we must break the symmetry of the right-sides of the productions

$$\begin{aligned} \text{E} &\to \text{E}\langle\text{plus}\rangle\text{E}, \\ \text{T} &\to \text{T}\langle\text{times}\rangle\text{T}, \end{aligned}$$

turning one of the E's into T, and one of the T's into F. To make our operators be left associative, we must use *left recursion*, changing the second E to T, and the second T to F; right associativity would result from making the opposite choices, i.e., using *right recursion*.

Example Disambiguation

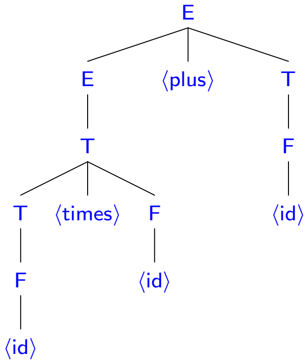
Thus, our unambiguous grammar of arithmetic expressions is

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

It can be proved that this grammar is indeed unambiguous, and that it is equivalent to the original grammar.

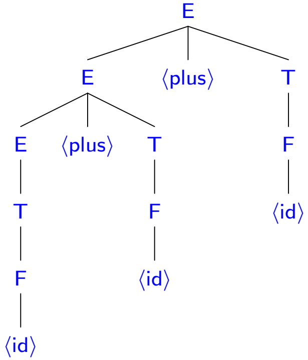
Example Disambiguation

Now, the only parse of ⟨id⟩⟨times⟩⟨id⟩⟨plus⟩⟨id⟩ is



Example Disambiguation

And, the only parse of ⟨id⟩⟨plus⟩⟨id⟩⟨plus⟩⟨id⟩ is



Top-down Parsing for Grammars of Operators

Top-down parsing is a simple and efficient parsing method for unambiguous grammars of operators like

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

Parsing

Let $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{F}$ be all of the parse trees that are valid for our grammar, have yields containing no variables, and whose root labels are E, T and F, respectively.

Because this grammar has three mutually recursive variables, we will need three mutually recursive parsing functions,

$$\begin{aligned}\mathbf{parE} &\in \mathbf{Str} \to \mathbf{Option}(\mathcal{E} \times \mathbf{Str}),\\ \mathbf{parT} &\in \mathbf{Str} \to \mathbf{Option}(\mathcal{T} \times \mathbf{Str}),\\ \mathbf{parF} &\in \mathbf{Str} \to \mathbf{Option}(\mathcal{F} \times \mathbf{Str}),\end{aligned}$$

which attempt to parse an element pt of $\mathcal{E}$, $\mathcal{T}$ or $\mathcal{F}$ out of a string w, returning **none** to indicate failure, and $\mathbf{some}(pt, y)$, where y is the remainder of w, otherwise.

Parsing

Given a string w, **parE** operates as follows. Because all elements of $\mathcal{E}$ have yields beginning with the yield of an element of

Parsing

Given a string w, **parE** operates as follows. Because all elements of $\mathcal{E}$ have yields beginning with the yield of an element of $\mathcal{T}$, it starts by evaluating **parT** w. If this results in **none**, it returns **none**. Otherwise, it results in $\textbf{some}(pt, x)$, for some $pt \in \mathcal{T}$ and $x \in \textbf{Str}$, in which case **parE** returns $\textbf{parELoop}(E(pt), x)$, where $\textbf{parELoop} \in \mathcal{E} \times \textbf{Str} \to \textbf{Option}(\mathcal{E} \times \textbf{Str})$ is defined recursively, as follows.

Parsing

Given a string w, **parE** operates as follows. Because all elements of $\mathcal{E}$ have yields beginning with the yield of an element of $\mathcal{T}$, it starts by evaluating **parT** w. If this results in **none**, it returns **none**. Otherwise, it results in **some**(pt, x), for some $pt \in \mathcal{T}$ and $x \in$ **Str**, in which case **parE** returns **parELoop**$(E(pt), x)$, where **parELoop** $\in \mathcal{E} \times$ **Str** $\to$ **Option**$(\mathcal{E} \times$ **Str**$)$ is defined recursively, as follows.

Given $(pt, x) \in \mathcal{E} \times$ **Str**, **parELoop** proceeds as follows.

- If $x =$ for some y,

Parsing

Given a string w, **parE** operates as follows. Because all elements of $\mathcal{E}$ have yields beginning with the yield of an element of $\mathcal{T}$, it starts by evaluating **parT** w. If this results in **none**, it returns **none**. Otherwise, it results in $\textbf{some}(pt, x)$, for some $pt \in \mathcal{T}$ and $x \in \textbf{Str}$, in which case **parE** returns $\textbf{parELoop}(E(pt), x)$, where $\textbf{parELoop} \in \mathcal{E} \times \textbf{Str} \to \textbf{Option}(\mathcal{E} \times \textbf{Str})$ is defined recursively, as follows.

Given $(pt, x) \in \mathcal{E} \times \textbf{Str}$, **parELoop** proceeds as follows.

- If $x = \langle\text{plus}\rangle y$ for some y,

Parsing

Given a string w, **parE** operates as follows. Because all elements of $\mathcal{E}$ have yields beginning with the yield of an element of $\mathcal{T}$, it starts by evaluating **parT** w. If this results in **none**, it returns **none**. Otherwise, it results in $\textbf{some}(pt, x)$, for some $pt \in \mathcal{T}$ and $x \in \textbf{Str}$, in which case **parE** returns $\textbf{parELoop}(E(pt), x)$, where $\textbf{parELoop} \in \mathcal{E} \times \textbf{Str} \to \textbf{Option}(\mathcal{E} \times \textbf{Str})$ is defined recursively, as follows.

Given $(pt, x) \in \mathcal{E} \times \textbf{Str}$, **parELoop** proceeds as follows.

- If $x = \langle\text{plus}\rangle y$ for some y, then **parELoop** evaluates **parT** y.
    - If this results in **none**, then **parELoop** returns **none**.

Parsing

Given a string w, **parE** operates as follows. Because all elements of $\mathcal{E}$ have yields beginning with the yield of an element of $\mathcal{T}$, it starts by evaluating **parT** w. If this results in **none**, it returns **none**. Otherwise, it results in $\mathbf{some}(pt, x)$, for some $pt \in \mathcal{T}$ and $x \in \mathbf{Str}$, in which case **parE** returns $\mathbf{parELoop}(E(pt), x)$, where $\mathbf{parELoop} \in \mathcal{E} \times \mathbf{Str} \to \mathbf{Option}(\mathcal{E} \times \mathbf{Str})$ is defined recursively, as follows.

Given $(pt, x) \in \mathcal{E} \times \mathbf{Str}$, **parELoop** proceeds as follows.

- If $x = \langle\mathrm{plus}\rangle y$ for some y, then **parELoop** evaluates **parT** y.
  - If this results in **none**, then **parELoop** returns **none**.
  - Otherwise, it results in $\mathbf{some}(pt', z)$ for some $pt' \in \mathcal{T}$ and $z \in \mathbf{Str}$, and **parELoop** returns

Parsing

Given a string w, **parE** operates as follows. Because all elements of $\mathcal{E}$ have yields beginning with the yield of an element of $\mathcal{T}$, it starts by evaluating **parT** w. If this results in **none**, it returns **none**. Otherwise, it results in $\mathbf{some}(pt, x)$, for some $pt \in \mathcal{T}$ and $x \in \mathbf{Str}$, in which case **parE** returns $\mathbf{parELoop}(E(pt), x)$, where $\mathbf{parELoop} \in \mathcal{E} \times \mathbf{Str} \to \mathbf{Option}(\mathcal{E} \times \mathbf{Str})$ is defined recursively, as follows.

Given $(pt, x) \in \mathcal{E} \times \mathbf{Str}$, **parELoop** proceeds as follows.

- If $x = \langle\mathsf{plus}\rangle y$ for some y, then **parELoop** evaluates **parT** y.
  - If this results in **none**, then **parELoop** returns **none**.
  - Otherwise, it results in $\mathbf{some}(pt', z)$ for some $pt' \in \mathcal{T}$ and $z \in \mathbf{Str}$, and **parELoop** returns $\mathbf{parELoop}(E(pt, \langle\mathsf{plus}\rangle, pt'), z)$.

Parsing

Given a string w, **parE** operates as follows. Because all elements of $\mathcal{E}$ have yields beginning with the yield of an element of $\mathcal{T}$, it starts by evaluating **parT** w. If this results in **none**, it returns **none**. Otherwise, it results in $\textbf{some}(pt, x)$, for some $pt \in \mathcal{T}$ and $x \in \textbf{Str}$, in which case **parE** returns $\textbf{parELoop}(E(pt), x)$, where $\textbf{parELoop} \in \mathcal{E} \times \textbf{Str} \to \textbf{Option}(\mathcal{E} \times \textbf{Str})$ is defined recursively, as follows.

Given $(pt, x) \in \mathcal{E} \times \textbf{Str}$, **parELoop** proceeds as follows.

- If $x = \langle \text{plus} \rangle y$ for some y, then **parELoop** evaluates **parT** y.
  - If this results in **none**, then **parELoop** returns **none**.
  - Otherwise, it results in $\textbf{some}(pt', z)$ for some $pt' \in \mathcal{T}$ and $z \in \textbf{Str}$, and **parELoop** returns $\textbf{parELoop}(E(pt, \langle \text{plus} \rangle, pt'), z)$.
- Otherwise, **parELoop** returns

Parsing

Given a string w, **parE** operates as follows. Because all elements of $\mathcal{E}$ have yields beginning with the yield of an element of $\mathcal{T}$, it starts by evaluating **parT** w. If this results in **none**, it returns **none**. Otherwise, it results in **some**(pt, x), for some $pt \in \mathcal{T}$ and $x \in$ **Str**, in which case **parE** returns **parELoop**$(E(pt), x)$, where **parELoop** $\in \mathcal{E} \times$ **Str** $\to$ **Option**$(\mathcal{E} \times$ **Str**$)$ is defined recursively, as follows.

Given $(pt, x) \in \mathcal{E} \times$ **Str**, **parELoop** proceeds as follows.

- If $x = \langle \text{plus} \rangle y$ for some y, then **parELoop** evaluates **parT** y.
  - If this results in **none**, then **parELoop** returns **none**.
  - Otherwise, it results in **some**(pt', z) for some $pt' \in \mathcal{T}$ and $z \in$ **Str**, and **parELoop** returns **parELoop**$(E(pt, \langle \text{plus} \rangle, pt'), z)$.
- Otherwise, **parELoop** returns **some**(pt, x).

Parsing

Given a string w, **parE** operates as follows. Because all elements of $\mathcal{E}$ have yields beginning with the yield of an element of $\mathcal{T}$, it starts by evaluating **parT** w. If this results in **none**, it returns **none**. Otherwise, it results in **some**(pt, x), for some $pt \in \mathcal{T}$ and $x \in$ **Str**, in which case **parE** returns **parELoop**$(E(pt), x)$, where **parELoop** $\in \mathcal{E} \times$ **Str** $\to$ **Option**$(\mathcal{E} \times$ **Str**$)$ is defined recursively, as follows.

Given $(pt, x) \in \mathcal{E} \times$ **Str**, **parELoop** proceeds as follows.

- If $x = \langle\mathsf{plus}\rangle y$ for some y, then **parELoop** evaluates **parT** y.
  - If this results in **none**, then **parELoop** returns **none**.
  - Otherwise, it results in **some**(pt', z) for some $pt' \in \mathcal{T}$ and $z \in$ **Str**, and **parELoop** returns **parELoop**$(E(pt, \langle\mathsf{plus}\rangle, pt'), z)$.
- Otherwise, **parELoop** returns **some**(pt, x).

The function **parT** operates analogously.

Parsing

Given a string w, **parF** proceeds as follows.

- If $w = \langle \text{id} \rangle x$ for some x, then it returns

Parsing

Given a string w, **parF** proceeds as follows.

- If $w = \langle \mathsf{id} \rangle x$ for some x, then it returns $\mathbf{some}(F(\langle \mathsf{id} \rangle), x)$.

Parsing

Given a string w, **parF** proceeds as follows.

- If $w = \langle \text{id} \rangle x$ for some x, then it returns $\textbf{some}(F(\langle \text{id} \rangle), x)$.
- Otherwise, if $w = \langle \text{openPar} \rangle x$, then **parF** evaluates $\textbf{parE}\, x$.
  - If this results in **none**, it returns **none**.

Parsing

Given a string w, **parF** proceeds as follows.

- If $w = \langle \text{id} \rangle x$ for some x, then it returns $\textbf{some}(F(\langle \text{id} \rangle), x)$.
- Otherwise, if $w = \langle \text{openPar} \rangle x$, then **parF** evaluates $\textbf{parE}\, x$.
  - If this results in **none**, it returns **none**.
  - Otherwise, this results in $\textbf{some}(pt, y)$ for some $pt \in \mathcal{E}$ and $y \in \textbf{Str}$.
    - If $y = \langle \text{closPar} \rangle z$ for some z, then **parF** returns

Parsing

Given a string w, **parF** proceeds as follows.

- If $w = \langle \text{id} \rangle x$ for some x, then it returns $\textbf{some}(F(\langle \text{id} \rangle), x)$.
- Otherwise, if $w = \langle \text{openPar} \rangle x$, then **parF** evaluates $\textbf{parE}\, x$.
  - If this results in **none**, it returns **none**.
  - Otherwise, this results in $\textbf{some}(pt, y)$ for some $pt \in \mathcal{E}$ and $y \in \textbf{Str}$.
    - If $y = \langle \text{closPar} \rangle z$ for some z, then **parF** returns $\textbf{some}(F(\langle \text{openPar} \rangle, pt, \langle \text{closPar} \rangle), z)$.

Parsing

Given a string w, **parF** proceeds as follows.

- If $w = \langle \text{id} \rangle x$ for some x, then it returns $\textbf{some}(F(\langle \text{id} \rangle), x)$.
- Otherwise, if $w = \langle \text{openPar} \rangle x$, then **parF** evaluates $\textbf{parE}\, x$.
  - If this results in **none**, it returns **none**.
  - Otherwise, this results in $\textbf{some}(pt, y)$ for some $pt \in \mathcal{E}$ and $y \in \textbf{Str}$.
    - If $y = \langle \text{closPar} \rangle z$ for some z, then **parF** returns $\textbf{some}(F(\langle \text{openPar} \rangle, pt, \langle \text{closPar} \rangle), z)$.
    - Otherwise, **parF** returns **none**.

Parsing

Given a string w, **parF** proceeds as follows.

- If $w = \langle \text{id} \rangle x$ for some x, then it returns $\textbf{some}(F(\langle \text{id} \rangle), x)$.
- Otherwise, if $w = \langle \text{openPar} \rangle x$, then **parF** evaluates $\textbf{parE}\, x$.
  - If this results in **none**, it returns **none**.
  - Otherwise, this results in $\textbf{some}(pt, y)$ for some $pt \in \mathcal{E}$ and $y \in \textbf{Str}$.
    - If $y = \langle \text{closPar} \rangle z$ for some z, then **parF** returns $\textbf{some}(F(\langle \text{openPar} \rangle, pt, \langle \text{closPar} \rangle), z)$.
    - Otherwise, **parF** returns **none**.
- Otherwise **parF** returns **none**.

Parsing

Given a string w to parse, the algorithm evaluates **parE** w. If the result of this evaluation is:

- **none**, then the algorithm reports failure;

Parsing

Given a string w to parse, the algorithm evaluates **parE** w. If the result of this evaluation is:

- **none**, then the algorithm reports failure;
- **some**$(pt, \%)$, then the algorithm returns pt;

Parsing

Given a string w to parse, the algorithm evaluates **parE** w. If the result of this evaluation is:

- **none**, then the algorithm reports failure;
- **some**$(pt, \%)$, then the algorithm returns pt;
- **some**(pt, y), where $y \neq \%$, then the algorithm reports failure, because not all of the input could be parsed.