Compiler Construction

Parsing & Lexing

Hannes Saffrich
Peter Thiemann

University of Freiburg
Department of Computer Science
Programming Languages

28. April 2025

Parser

- A parser checks if a word is part of a language.
- An alphabet Σ is a finite set.
- The elements $a \in \Sigma$ are called letters or *terminal symbols*.
- A word w is a list of letters, i.e., $w \in \Sigma^*$.
- A language $\mathcal{L}$ is a set of words, i.e., $\mathcal{L} \subseteq \Sigma^*$.
- We consider languages described by *context-free grammars*.
- Information provided by a parser:
  - if a word is in the language, it produces one (or more) syntax trees;
  - if a word is not in the language, it produces an error message describing why it is not in the language.

Lexer

- A lexer is an optional preprocessing step for a parser.
- It translates words from one alphabet to words of another alphabet

$$\text{lexer} : \Sigma^* \to \Delta^*$$

- Typically, $\Sigma =$ set of (unicode) characters.
- The output letters $t \in \Delta$ are called *Tokens*.
- A lexer serves two purposes:
  - Allow the parser to be based on a more readable grammar, e.g., by removing whitespace and comments, and treating numbers or variable names as single letters;
  - Increase performance as some tasks can be done more efficiently in a lexer.
- We describe a lexer by a mapping from regular expressions to tokens.

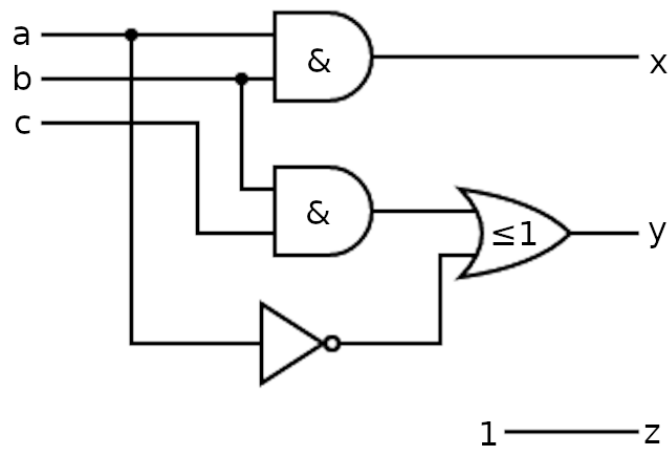
Running Example: A Circuit Description Language

$$\langle prog \rangle ::= (\langle expr \rangle \texttt{;})^*$$

$$\begin{aligned}\langle expr \rangle ::= & \texttt{True} \\ & \mid \texttt{False} \\ & \mid \langle var \rangle \\ & \mid \texttt{!}\langle expr \rangle \\ & \mid \langle expr \rangle \texttt{\&} \langle expr \rangle \\ & \mid \langle expr \rangle \texttt{|} \langle expr \rangle\end{aligned}$$

Example Program:

```
a & b;
!a | b & c;
True;
```



Lexer Example

Input String	Output Tokens
`"`	`[`
`a & b;`	`Id('a'), And(), Id('b'), Semicolon(),`
`!a | b & c;`	`Not(), Id('a'), Or(), Id('b'), And(), Id('c'), Semicolon(),`
`True;`	`True(), Semicolon(),`
`"`	`]`

Lexer Description (Semi-Formal)

```
Id        = "[a-zA-Z][a-zA-Z0-9]*"
True      = "True"
False     = "False"
And       = "&"
Or        = "\|"
Not       = "!"
Semicolon = ";"
Ignore1   = "[ \t\n]+"
Ignore2   = "#[^\n]*\n"
```

Lexer Implementation

- Let's write a naive lexer!
- ... or to be more precise: an interpreter for a lexer description!

Context-free Grammars: Definition

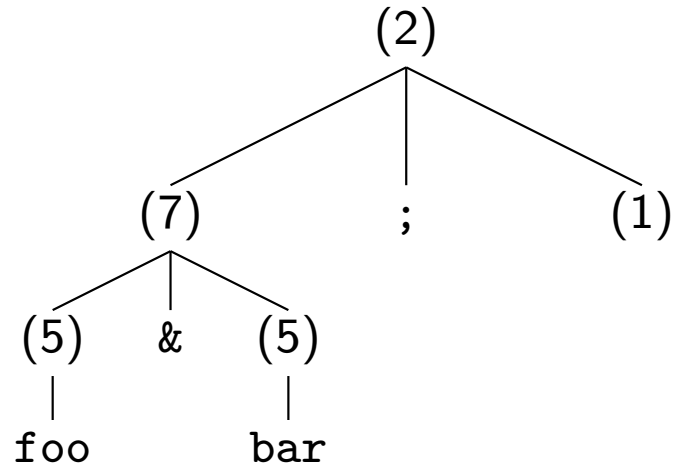
- Formally, a context-free grammar is a tuple (N, Σ, S, P) where
  - N is a finite set of non-terminal symbols
  - Σ is a set of terminal symbols
  - $S \in N$ is the start symbol
  - $P \subseteq N \times (N \cup \Sigma)^*$ are the production rules
- Example:
  - $N = \{\langle prog \rangle, \langle expr \rangle, \langle var \rangle\}$
  - $\Sigma = \{\texttt{True}, \texttt{False}, \texttt{;}, \texttt{\&}, \texttt{|}, \texttt{a}, \texttt{b}, \ldots\}$
  - $S = \langle prog \rangle$ is the start symbol
  - P contains the following rules:

$$\langle prog \rangle \rightarrow \qquad \langle expr \rangle \rightarrow \texttt{!}\langle expr \rangle$$
$$\langle prog \rangle \rightarrow \langle expr \rangle\texttt{;}\langle prog \rangle \qquad \langle expr \rangle \rightarrow \langle expr \rangle\texttt{\&}\langle expr \rangle$$
$$\langle expr \rangle \rightarrow \texttt{True} \qquad \langle expr \rangle \rightarrow \langle expr \rangle\texttt{|}\langle expr \rangle$$
$$\langle expr \rangle \rightarrow \texttt{False} \qquad \langle var \rangle \rightarrow \texttt{Var(\_)}$$
$$\langle expr \rangle \rightarrow \langle var \rangle$$

Grammars: Language

- The language of a grammar is the set of words that can be derived from the start symbol by applying production rules
- Example:

$$\begin{aligned}
\langle prog \rangle &\Rightarrow \langle expr \rangle ; \langle prog \rangle \\
&\Rightarrow \langle expr \rangle ; \\
&\Rightarrow \langle expr \rangle \& \langle expr \rangle ; \\
&\Rightarrow \langle var \rangle \& \langle expr \rangle ; \\
&\Rightarrow \texttt{foo\&}\langle expr \rangle ; \\
&\Rightarrow \texttt{foo\&}\langle var \rangle ; \\
&\Rightarrow \texttt{foo\&bar;}
\end{aligned}$$



- Rules:

$$\begin{aligned}
\langle prog \rangle &\rightarrow && (1) \\
\langle prog \rangle &\rightarrow \langle expr \rangle ; \langle prog \rangle && (2) \\
\langle expr \rangle &\rightarrow \texttt{True} && (3) \\
\langle expr \rangle &\rightarrow \texttt{False} && (4) \\
\langle expr \rangle &\rightarrow \langle var \rangle && (5) \\
\langle expr \rangle &\rightarrow \texttt{!}\langle expr \rangle && (6) \\
\langle expr \rangle &\rightarrow \langle expr \rangle \& \langle expr \rangle && (7) \\
\langle expr \rangle &\rightarrow \langle expr \rangle \texttt{|} \langle expr \rangle && (8) \\
\langle var \rangle &\rightarrow \texttt{Var(_)} && (9)
\end{aligned}$$

Grammars: EBNF Notation

- The *Extended Backus–Naur form (EBNF)* is a concise, but equivalent notation for writing down the production rules of a grammar.
- Multiple rules with the same left side are combined by writing the right side of the rule as a regular expression over the alphabet $N \cup \Sigma$.
- Our example grammar in EBNF notation:

$$\langle prog \rangle ::= (\langle expr \rangle \texttt{;})^*$$
$$\langle expr \rangle ::= \texttt{True} \mid \texttt{False} \mid \langle var \rangle \mid \texttt{!}\langle expr \rangle \mid \langle expr \rangle \texttt{\&} \langle expr \rangle \mid \langle expr \rangle \texttt{|} \langle expr \rangle$$
$$\langle var \rangle ::= \texttt{Var(\_)}$$

- Our example grammar in rule notation:

$$\langle prog \rangle \to$$
$$\langle prog \rangle \to \langle expr \rangle \texttt{;} \langle prog \rangle$$
$$\langle expr \rangle \to \texttt{True}$$
$$\langle expr \rangle \to \texttt{False}$$
$$\langle expr \rangle \to \langle var \rangle$$
$$\langle expr \rangle \to \texttt{!}\langle expr \rangle$$
$$\langle expr \rangle \to \langle expr \rangle \texttt{\&} \langle expr \rangle$$
$$\langle expr \rangle \to \langle expr \rangle \texttt{|} \langle expr \rangle$$
$$\langle var \rangle \to \texttt{Var(\_)}$$

Grammars: Ambiguity

- A grammar is *ambiguous*, if a word can be derived in multiple ways.
- Our example grammar is ambiguous ...

$$\langle prog \rangle ::= (\langle expr \rangle \texttt{;})^*$$
$$\langle expr \rangle ::= \texttt{True} \mid \texttt{False} \mid \langle var \rangle \mid \texttt{!}\langle expr \rangle \mid \langle expr \rangle \texttt{\&} \langle expr \rangle \mid \langle expr \rangle \texttt{|} \langle expr \rangle$$
$$\langle var \rangle ::= \texttt{Var(\_)}$$

- The word `x & y & z;` can be derived as `(x & y) & z;` and `x & (y & z);`
- Even worse, the word `x & y | z;` can be derived as `(x & y) | z;` and `x & (y | z);` where only the first one is valid ("and" binds stronger than "or")
- No problem if the grammar is used to describe abstract syntax tree data types, but it matters for parsing *concrete syntax*.
- We amend the grammar to do so.

Grammars: Ambiguity (Fix 1)

- Grammar for abstract syntax (ambiguous):

$$\langle prog \rangle ::= (\langle expr \rangle \texttt{;})^*$$
$$\langle expr \rangle ::= \texttt{True} \mid \texttt{False} \mid \langle var \rangle \mid \texttt{!}\langle expr \rangle \mid \langle expr \rangle \texttt{\&} \langle expr \rangle \mid \langle expr \rangle \texttt{|} \langle expr \rangle$$
$$\langle var \rangle ::= \texttt{Var(\_)}$$

- Grammar for concrete syntax (unambiguous):

$$\langle prog \rangle ::= (\langle expr \rangle \texttt{;})^*$$
$$\langle expr \rangle ::= \texttt{True} \mid \texttt{False} \mid \langle var \rangle \mid \texttt{(!}\langle expr \rangle \texttt{)}n \mid \texttt{(}\langle expr \rangle \texttt{\&} \langle expr \rangle \texttt{)} \mid \texttt{(}\langle expr \rangle \texttt{|} \langle expr \rangle \texttt{)}$$
$$\langle var \rangle ::= \texttt{Var(\_)}$$

- This forbids `!a & b | c;`, but requires us to write `(((!a) & b) | c);`

Grammars: Ambiguity

- What we actually want is:
  - `&` binds stronger than `|`, e.g. `x & y | z` means `(x & y) | z`
  - `!` binds stronger than `&`, e.g. `!x & y` means `(!x) & y`
  - `&` is left-associative, e.g. `x & y & z` means `(x & y) & z`
  - `|` is left-associative, e.g. `x | y | z` means `(x | y) | z`
  - `!` is right-associative, e.g. `!!x` means `! (! x)`
- As `&` and `|` are associative operators, making them left-associative is an arbitrary choice, and we could just as well make them right-associative.

Grammars: Ambiguity (Fix 2)

- Grammar for abstract syntax (ambiguous):

$$\langle prog \rangle ::= (\langle expr \rangle \texttt{;})^*$$
$$\langle expr \rangle ::= \texttt{True} \mid \texttt{False} \mid \langle var \rangle \mid \texttt{!}\langle expr \rangle \mid \langle expr \rangle \texttt{\&} \langle expr \rangle \mid \langle expr \rangle \texttt{|} \langle expr \rangle$$
$$\langle var \rangle ::= \texttt{Var(\_)}$$

- Grammar for concrete syntax (unambiguous):

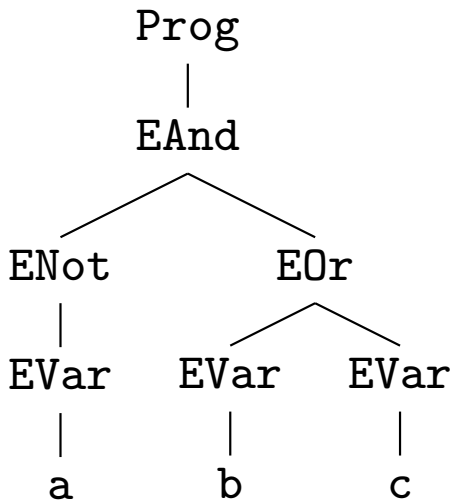
$$\begin{aligned}
\langle prog \rangle &::= (\langle expr \rangle \texttt{;})^* \\
\langle expr \rangle &::= \langle expr \rangle \texttt{|} \langle expr1 \rangle \mid \langle expr1 \rangle \\
\langle expr1 \rangle &::= \langle expr1 \rangle \texttt{\&} \langle expr2 \rangle \mid \langle expr2 \rangle \\
\langle expr2 \rangle &::= \texttt{!}\langle expr2 \rangle \mid \langle expr3 \rangle \\
\langle expr3 \rangle &::= \texttt{True} \mid \texttt{False} \mid \langle var \rangle \mid \texttt{(}\langle expr \rangle\texttt{)} \\
\langle var \rangle &::= \texttt{Var(_)}
\end{aligned}$$

- This change forces `!a & b | c;` to be parsed as `(((!a) & b) | c);`

Grammars: Abstract Syntax vs Concrete Syntax

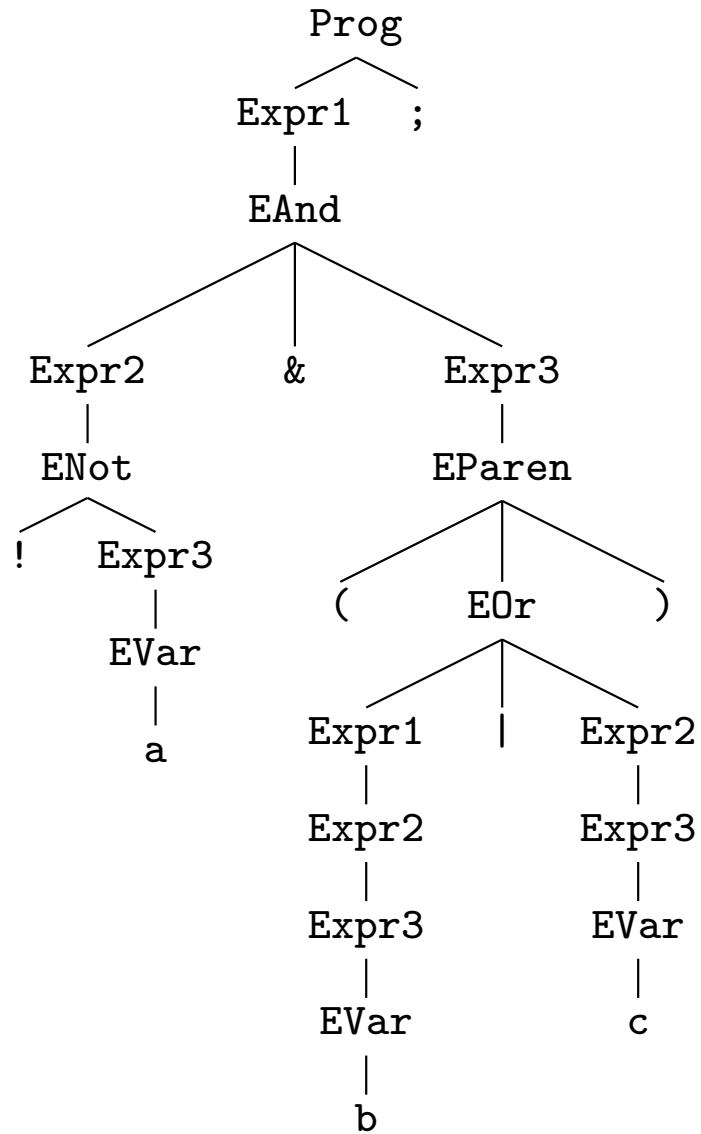
Example word: `!a & (b | c);`

Abstract Syntax Tree:



In most scenarios, we parse the concrete syntax, but let the parser generate an abstract syntax tree.

Concrete Syntax Tree:



Grammars: Classification

The *Chomsky Hierarchy* classifies languages by the kind of production rules that are required to describe them.

- *Context-free* languages can be described by rules of the form:

$$A \to \alpha \qquad \forall A \in N, \alpha \in (N \cup \Sigma)^*$$

  - Parsing with $O(n^3)$ worst-case time complexity.
  - There are subsets (LL, LR, LALR) that place further restrictions on the grammar so that it can be parsed in $O(n)$ time. These languages are all deterministic and hence unambiguous.
  - Sufficiently powerful to parse realistic languages.
- *Regular languages* can be described by rules of the form:

$$A \to a \qquad A \to aB \qquad \forall A, B \in N, a \in \Sigma$$

  - Parsing with $O(n)$ worst-case time complexity.

The Earley Parser

- Classical parsing algorithm for arbitrary context-free grammars.
  Jay Earley. 1970. An efficient context-free parsing algorithm. Commun. ACM 13, 2 (Feb 1970), 94–102. https://doi.org/10.1145/362007.362035
- Worst-case time complexity:
  - $O(n^3)$ for ambiguous grammars (we don't care about those)
  - $O(n^2)$ for unambiguous grammars
- Relatively simple (170 lines of Python)
- Basically a grammar interpreter
- Nice for prototyping as it supports general context-free grammars
- Probably too slow for large files
- Produces multiple syntax trees when used with ambigous grammars

The Earley Parser: Basic Principle

- Loops once over each symbol of the input word
- Tracks which production rules made how much progress in a chart
- A *chart* maps each symbol index to a set of dotted rules
- A *dotted rule* is a production rule, which
  - has a marker (dot) in its right side that denotes how much of that rule was matched by the input so far
  - is annotated with the symbol index at which it was first added to the chart
- `chart[0]` is initialized with dotted rules derivable from the start symbol
- For each symbol `t` we compute `chart[i+1]` from `chart[i]` by checking, which rule expects a `t` after the dot and moving its dot one symbol to the right [some details omitted]
- A word is accepted, if the last entry of `chart` contains a rule with the start symbol on the left side and a dot at the end of its right side

The Earley Parser: Example (1 / 7)

Input word: `True & False ;"`

- Step 1: Add rules with the start symbol to the chart:

$$\texttt{chart[0]} = \{\ (0, \langle prog \rangle \to \bullet),\ (0, \langle prog \rangle \to \bullet \langle expr \rangle\, ; \langle prog \rangle)\ \}$$

- Step 2: For each dotted rule in `chart[0]`, also add the rules for all non-terminals which come immediately after a dot:

$$\begin{aligned}\texttt{chart[0]} = \texttt{chart[0]} \cup \{\ & (0, \langle expr \rangle \to \bullet \texttt{True}),\\ & (0, \langle expr \rangle \to \bullet \texttt{False}),\\ & (0, \langle expr \rangle \to \bullet \langle var \rangle),\\ & (0, \langle expr \rangle \to \bullet\, !\, \langle expr \rangle),\\ & (0, \langle expr \rangle \to \bullet \langle expr \rangle \& \langle expr \rangle),\\ & (0, \langle expr \rangle \to \bullet \langle expr \rangle\, |\, \langle expr \rangle)\ \}\end{aligned}$$

- Step 3: Repeat until the set doesn't change anymore:

$$\texttt{chart[0]} = \texttt{chart[0]} \cup \{\ (0, \langle var \rangle \to \bullet \texttt{Var(\_)})\ \}$$

The Earley Parser: Example (2 / 7)

Input word: `True & False ;"`

- Step 4: The first input symbol is `True`, so search for dotted rules in `chart[0]` with `True` after the dot:

$$(0, \langle expr \rangle \to \bullet \texttt{True})$$

- Shift the dot after the matched symbol:

$$(0, \langle expr \rangle \to \texttt{True} \bullet)$$

- Add the modified rules to `chart[1]`:

$$\texttt{chart[1]} = \{ (0, \langle expr \rangle \to \texttt{True} \bullet) \}$$

The Earley Parser: Example (3 / 7)

Input word: `True & False ;"`

- Step 5: Check for completed rules in `chart[1]`, i.e. rules which have the dot at the end:

$$(0, \langle expr \rangle \rightarrow \texttt{True}\bullet)$$

- This rule was created at the beginning (0), so search for rules in `chart[0]`, which have $\langle expr \rangle$ after the dot:

$$(0, \langle expr \rangle \rightarrow \bullet\langle expr \rangle \& \langle expr \rangle)$$
$$(0, \langle expr \rangle \rightarrow \bullet\langle expr \rangle \mid \langle expr \rangle)$$

- Shift the dot after $\langle expr \rangle$ and add them to `chart[1]`:

$$\texttt{chart[1] = chart[1]} \cup \{\ (0, \langle expr \rangle \rightarrow \langle expr \rangle \bullet \& \langle expr \rangle)$$
$$(0, \langle expr \rangle \rightarrow \langle expr \rangle \bullet \mid \langle expr \rangle)\ \}$$

The Earley Parser: Example (4 / 7)

Input word: `True & False ;"`

- Check if there are rules in `chart[1]` where the dot is in front of a non-terminal, and add the rules for the non-terminal also to `chart[1]` (same as for `chart[0]` in the beginning).
- There are no such rules.
- Repeat Step 5 until the `chart[1]` doesn't change anymore.
- No further additions required for this case.

The Earley Parser: Example (5 / 7)

Input word: `True & False ;"`

- Step 6: The next input symbol at position 1 is & so check for rules in `chart[1]` with & after the dot:

$$(0, \langle expr \rangle \to \langle expr \rangle \bullet \,\&\langle expr \rangle)$$

- Shift the dot to the right and add to `chart[2]`:

$$\texttt{chart[2]} = \{\ (0, \langle expr \rangle \to \langle expr \rangle \,\&\, \bullet \langle expr \rangle)\ \}$$

- Step 7: Add all reachable dotted rules to `chart[2]`:

$$\begin{aligned}\texttt{chart[2]} = \texttt{chart[2]} \cup \{\ & (2, \langle expr \rangle \to \bullet\texttt{True}),\\ & (2, \langle expr \rangle \to \bullet\texttt{False}),\\ & (2, \langle expr \rangle \to \bullet\langle var \rangle),\\ & (2, \langle expr \rangle \to \bullet\,!\langle expr \rangle),\\ & (2, \langle expr \rangle \to \bullet\langle expr \rangle\&\langle expr \rangle),\\ & (2, \langle expr \rangle \to \bullet\langle expr \rangle\,|\,\langle expr \rangle),\\ & (2, \langle var \rangle \to \bullet\texttt{Var(\_)})\ \}\end{aligned}$$

- There are no complete rules in `chart[2]`.

The Earley Parser: Example (6 / 7)

Input word: `True & False ;"`

- Step 8: The next input symbol at position 2 is `False` so check for rules in `chart[2]` with `False` after the dot:

$$(2, \langle expr \rangle \to \bullet \texttt{False})$$

- Shift the dot to the right and add to `chart[3]`:

$$\texttt{chart[3]} = \{\ (2, \langle expr \rangle \to \texttt{False}\bullet)\ \}$$

- This dotted rule is complete, so shift `chart[2]` on $\langle expr \rangle$.
- Step 9: Add all reachable dotted rules to `chart[3]`:

$$\begin{aligned}\texttt{chart[3]} = \texttt{chart[3]} \cup \{\ & (0, \langle expr \rangle \to \langle expr \rangle \& \langle expr \rangle \bullet), \\ & (2, \langle expr \rangle \to \langle expr \rangle \bullet \& \langle expr \rangle), \\ & (2, \langle expr \rangle \to \langle expr \rangle \bullet \mid \langle expr \rangle), \\ & (0, \langle prog \rangle \to \langle expr \rangle \bullet ; \langle prog \rangle)\ \}\end{aligned}$$

The Earley Parser: Example (7 / 7)

Input word: `True & False ;"`

- Step 10: The next input symbol at position 3 is `;` so check for rules in `chart[3]` with `;` after the dot:

$$(0, \langle prog \rangle \to \langle expr \rangle \bullet \texttt{;} \langle prog \rangle)$$

- Shift the dot to the right and add to `chart[4]`:

$$\texttt{chart[4]} = \{\ (0, \langle prog \rangle \to \langle expr \rangle \texttt{;} \bullet \langle prog \rangle)\ \}$$

- Step 11: Add al reachable dotted rules to `chart[4]`:

$$\begin{aligned}\texttt{chart[4]} = \texttt{chart[4]} \cup \{\ & (4, \langle prog \rangle \to \bullet), \\ & (4, \langle prog \rangle \to \bullet \langle expr \rangle \texttt{;} \langle prog \rangle), \\ & (4, \langle expr \rangle \to \bullet \texttt{True}), \\ & \ldots \\ & (0, \langle prog \rangle \to \langle expr \rangle \texttt{;} \langle prog \rangle \bullet)\ \}\end{aligned}$$

- `chart[4]` contains a completed rule from the beginning (0) with the start symbol on the left side, hence the input word is in the language.