Compiler Construction

Parsing & Lexing

Hannes Saffrich

University of Freiburg
Department of Computer Science
Programming Languages

26. April 2024

Parser

- A parser checks if a word is part of a language
- An alphabet Σ is a 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. $\mathcal{L} \subseteq \Sigma^*$
- A word w is part of the language $\mathcal{L}$ if $w \in \mathcal{L}$
- Typically, a language is described by a grammar
- Typically, a parser produces additional information:
  - 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

$$\Sigma^* \to \Delta^*$$

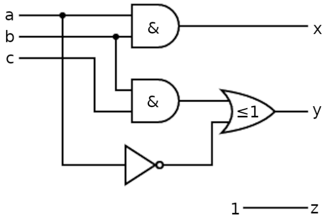
- Typically, the input words are actual strings, i.e. $\Sigma^* = str$
- The output letters $t \in \Delta$ are called *Tokens*
- A lexer serves two purposes:
  - Allow the parser to be based on a more readable grammar, e.g. by removing whitespace and comments, and treating numbers or variable names as single letters
  - Increase performance as some parsing tasks can be done more efficiently in a lexer
- Typically, a lexer is described by a mapping from regular expressions to tokens

Running Example: A Circuit Description Language

$$
\begin{aligned}
\langle \mathit{prog} \rangle &::= (\langle \mathit{expr} \rangle \,\texttt{;}\,)^* \\
\langle \mathit{expr} \rangle &::= \texttt{True} \\
&\mid \texttt{False} \\
&\mid \langle \mathit{var} \rangle \\
&\mid \texttt{!}\langle \mathit{expr} \rangle \\
&\mid \langle \mathit{expr} \rangle \texttt{\&} \langle \mathit{expr} \rangle \\
&\mid \langle \mathit{expr} \rangle \texttt{|} \langle \mathit{expr} \rangle
\end{aligned}
$$

Example Program:

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



Lexer Example

Input String

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

True;
"
```

Output Tokens

```
[
    Id('a'), And(), Id('b'), Semicolon(),
    Not(), Id('a'), Or(), Id('b'), And(),
        Id('c'), Semicolon(),
    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 lexer!
- ... or to be more precise: a lexer library!

Grammars: Definition

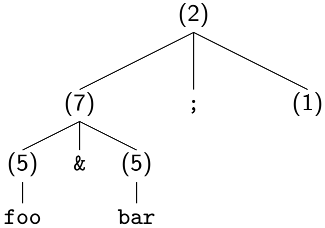
- Typically, languages are specified as grammars
- Formally, a grammar is a tuple (N, Σ, S, P) where
  - N is a finite set of non-terminal symbols
  - Σ is a set of terminal symbols
  - $S \in N$ is the start symbol
  - $P \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ are the production rules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

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

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:

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



- Rules:

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

Grammars: EBNF Notation

- The *Extended Backus–Naur form (EBNF)* is a more concise 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:

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

- Our example grammar in rule notation:

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

Grammars: Ambiguity

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

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

- 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")
- This is no problem if the grammar only used to describe abstract syntax tree data types, but it matters for parsing
- We can fix this by slightly adjusting the grammar to capture the *concrete syntax*

Grammars: Ambiguity (Fix 1)

- Grammar for abstract syntax (ambiguous):

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

- Grammar for concrete syntax (unambiguous):

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

- This forbids `!a & b | c;` and 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):

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

- Grammar for concrete syntax (unambiguous):

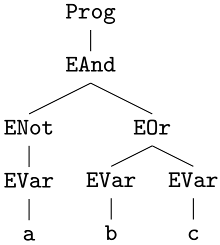
$$
\begin{aligned}
\langle prog \rangle &::= (\langle expr \rangle\,\texttt{;})^* \\
\langle expr \rangle &::= \langle expr \rangle\texttt{|}\langle expr1 \rangle \mid \langle expr1 \rangle \\
\langle expr1 \rangle &::= \langle expr1 \rangle\texttt{\&}\langle expr2 \rangle \mid \langle expr2 \rangle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

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

Grammars: Abstract Syntax vs Concrete Syntax

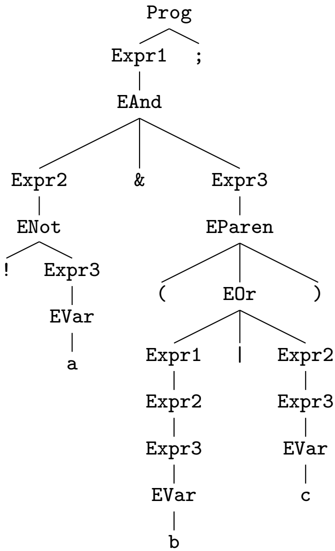
Example word: `!a & (b | c);`

Abstract Syntax Tree:



In most scenarios, we want to parse according to the concrete syntax, but have the parser generate an abstract syntax tree → attribute grammars

Concrete Syntax Tree:



Grammars: Classification

- The *Chomsky Hierarchy* classifies languages by the kind of production rules that are required to describe them:
  - *Recursively enumerable* languages can be described by rules of the form:

    $$\alpha \to \beta \qquad \forall \alpha, \beta \in (N \cup \Sigma)^*, \alpha \text{ not empty}$$

    - Parsing is semi-decidable
  - *Context-sensitive* languages can be described by rules of the form:

    $$\alpha A \gamma \to \alpha \beta \gamma \qquad \forall A \in N, \alpha, \beta, \gamma \in (N \cup \Sigma)^*, \beta \text{ not empty}$$

    - Parsing is PSPACE-complete
  - *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
  - *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

Grammars: Classification

- We are only concerned with regular and context free languages
  - Regular languages for lexing
  - Context-free languages for parsing
- The majority of programming languages can be described by a context-free grammar*
- There are many subclasses of context-free languages, which are more restrictive, but have better time and space complexity
  - e.g. the language classes LL, LR, LALR can be parsed in linear time, but are still powerful enough to parse Java
  - writing grammars in these subclasses can be more annoying, because less shapes of production rules are allowed

*: Some require minor preprocessing, like Python ("semantic whitespace")

The Earley Parser

- Can parse arbitrary context-free grammars
- 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)
- Can be reused as it is basically a grammar interpreter
- Nice for prototyping as it supports general context-freee grammars
- Probably a bit slow for single files with millions of tokens
- Produces multiple syntax trees when used with ambigous grammars

The Earley Parser: Basic Principle

- Loops once over each token of the input word
- Tracks which production rules made how much progress in a chart
- A *chart* maps each token 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 token index at which it was first added to the chart
- `chart[0]` is initialized with dotted rules derivable from the start symbol
- For each token `t` we compute `chart[i]` from `chart[i-1]` by checking, which rule expects a `t` after the dot and moving their 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:

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

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

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

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

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

The Earley Parser: Example (2 / 7)

- Input word: `"True & False ;"`
- Step 4: The first input token is `True`, so search for dotted rules in `chart[0]`, which have an `True` after the dot:

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

- Move the dot after the matched symbol:

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

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

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

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:

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

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

$$(0, \langle \mathit{expr} \rangle \rightarrow \bullet\langle \mathit{expr} \rangle \& \langle \mathit{expr} \rangle)$$
$$(0, \langle \mathit{expr} \rangle \rightarrow \bullet\langle \mathit{expr} \rangle \,|\, \langle \mathit{expr} \rangle)$$

- Move the dot after $\langle \mathit{expr} \rangle$ and add them to `chart[1]`:

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

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.
- This is already the case.

The Earley Parser: Example (5 / 7)

- Input word: `"True & False ;"`
- Step 6: The next input token is & so check for rules in `chart[1]` with an & after the dot:

$$(0, \langle \mathit{expr} \rangle \rightarrow \langle \mathit{expr} \rangle \bullet \& \langle \mathit{expr} \rangle)$$

- Move the • to the right and add to `chart[2]`:

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

- Step 7: Repeat Step 5 for `chart[2]` until it doesn't change anymore:

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

The Earley Parser: Example (6 / 7)

- Input word: `"True & False ;"`
- Step 8: The next input token is `False` so check for rules in `chart[2]` with an & after the dot:

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

- Move the $\bullet$ to the right and add to `chart[3]`:

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

- Step 9: Repeat Step 5 for `chart[3]` until it doesn't change anymore:

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

The Earley Parser: Example (7 / 7)

- Input word: `"True & False ;"`
- Step 10: The next input token is `;` so check for rules in `chart[3]` with an `;` after the dot:

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

- Move the $\bullet$ to the right and add to `chart[4]`:

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

- Step 11: Repeat Step 5 for `chart[4]` until it doesn't change anymore:

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

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