Chapter 5: Scanner design

Input (simplified): a set of rules:

\[ e_1 \quad \{ \text{action}_1 \} \]
\[ e_2 \quad \{ \text{action}_2 \} \]
\[ \cdots \]
\[ e_k \quad \{ \text{action}_k \} \]

Output: a program,

\[ \cdots \quad \text{reading a maximal prefix } w \quad \text{from the input, that satisfies} \]
\[ e_1 \mid \cdots \mid e_k; \]
\[ \text{determining the minimal } i, \text{ such that } w \in [e_i]; \]
\[ \text{executing } \{ \text{action}_i \} \text{ for } w. \]
Implementation:

Idea:

- Create the DFA $\mathcal{P}(A_e) = (Q, \Sigma, \delta, q_0, F)$ for the expression $e = (e_1 | \ldots | e_k)$;
- Define the sets:
  
  $F_1 = \{ q \in F | q \cap \text{last}[e_1] \neq \emptyset \}$
  
  $F_2 = \{ q \in (F \setminus F_1) | q \cap \text{last}[e_2] \neq \emptyset \}$
  
  $\ldots$
  
  $F_k = \{ q \in (F \setminus (F_1 \cup \ldots \cup F_{k-1})) | q \cap \text{last}[e_k] \neq \emptyset \}$

- For input $w$ we find: $\delta^*(q_0, w) \in F_k$ iff the scanner must execute action $\text{action}$ for $w$

Implementation:

Idea (cont’d):

- The scanner manages two pointers $(A, B)$ and the related states $(q_A, q_B)$...
- Pointer $A$ points to the last position in the input, after which a state $q_A \in F$ was reached;
- Pointer $B$ tracks the current position.

```plaintext
stdout, writeln ("Hello")
```

```
A B
0 0
```

Implementation:

Idea (cont’d):

- The current state being $q_B = \emptyset$, we consume input up to position $A$ and reset:

```plaintext
B := A; A := \perp;
q_B := q_0; q_A := \perp
```

```plaintext
writeln ("Hello")
```

```
A B
0 0
```
Implementation:

Idea (cont’d):

- The current state being \( q_B = \emptyset \), we consume input up
  position \( A \) and reset:

\[
B := A; \quad A := \perp;
q_B := q_0; \quad q_A := \perp
\]

Extension: States

- Now and then, it is handy to differentiate between particular
  scanner states.
- In different states, we want to recognize different token classes
  with different precedences.
- Depending on the consumed input, the scanner state can be
  changed.

Example: Comments

Within a comment, identifiers, constants, comments, ... are ignored

Input (generalized): a set of rules:

\[
\langle \text{state} \rangle \{ \begin{array}{l}
\quad e_1 \quad \{ \text{action}_1 \ \text{yybegin(state)_1);} \\
\quad e_2 \quad \{ \text{action}_2 \ \text{yybegin(state)_2);} \\
\quad \quad \vdots \\
\quad e_k \quad \{ \text{action}_k \ \text{yybegin(state)_k);} \\
\end{array} \}
\]

- The statement \( \text{yybegin(state)_i;} \) resets the current state
  to \( \text{state}_i \).
- The start state is called (e.g. flex JFlex) \( \text{YYINITIAL} \).

... for example:

\[
\langle \text{YYINITIAL} \rangle \quad "/\*" \quad \{ \text{yybegin(COMMENT);} \}
\langle \text{COMMENT} \rangle \quad "/\*" \quad \{ \text{yybegin(YYINITIAL);} \\
\quad \{ \text{yybegin(COMMENT);} \}
\quad \} \quad \} \\
\langle \text{ COMMENT } \rangle \quad . \quad \| \quad \text{n} \quad \{ \} \\
\]
Remarks:
- "." matches all characters different from "\n".
- For every state we generate the scanner respectively.
- Method `yybegin (STATE);` switches between different scanners.
- Comments might be directly implemented as (admittedly overly complex) token-class.
- Scanner-states are especially handy for implementing preprocessors, expanding special fragments in regular programs.

Syntactic Analysis

- Token-Stream → Parser → Syntaxtree

Syntactic analysis tries to integrate Tokens into larger program units.

Discussion:

In general, parsers are not developed by hand, but generated from a specification:

- Specification → Generator → Parser
Discussion:

In general, parsers are not developed by hand, but generated from a specification:

\[ E \rightarrow E \{ \text{op} \} E \]

**Generator**

**Specification of the hierarchical structure:** contextfree grammars

**Generated implementation:** Pushdown automata + X

---

**Basics:** Context-free Grammars

- Programs of programming languages can have arbitrary numbers of tokens, but only finitely many Token-classes.
- This is why we choose the set of Token-classes to be the finite alphabet of terminals $T$.
- The nested structure of program components can be described elegantly via context-free grammars...

**Basics:** Context-free Grammars

- Programs of programming languages can have arbitrary numbers of tokens, but only finitely many Token-classes.
- This is why we choose the set of Token-classes to be the finite alphabet of terminals $T$.
- The nested structure of program components can be described elegantly via context-free grammars...

**Definition: Context-Free Grammar**

A context-free grammar (CFG) is a 4-tuple $G = (N, T, P, S)$ with:

- $N$ the set of nonterminals,
- $T$ the set of terminals,
- $P$ the set of productions or rules, and
- $S \in N$ the start symbol.
The rules of context-free grammars take the following form:

\[ A \rightarrow \alpha \quad \text{with} \quad A \in N, \quad \alpha \in (N \cup T)^* \]

... for example:

\[ S \rightarrow aSb \]
\[ S \rightarrow \epsilon \]

Specified language: \( \{a^n b^n \mid n \geq 0\} \)

... a practical example:

\[ S \rightarrow \langle\text{stmt}\rangle \]
\[ \langle\text{stmt}\rangle \rightarrow \langle\text{if}\rangle \mid \langle\text{while}\rangle \mid \langle\text{exp}\rangle \]
\[ \langle\text{if}\rangle \rightarrow \text{if} (\langle\text{exp}\rangle) \langle\text{stmt}\rangle \text{ else } \langle\text{stmt}\rangle \]
\[ \langle\text{while}\rangle \rightarrow \text{while} (\langle\text{exp}\rangle) \langle\text{stmt}\rangle \]
\[ \langle\text{exp}\rangle \rightarrow \text{int} \mid \langle\text{exp}\rangle \mid \langle\text{exp}\rangle = \langle\text{exp}\rangle \]

Conventions:

In examples, we specify nonterminals and terminals in general implicitly:
- nonterminals are: \( A, B, C, \ldots, \langle\text{stmt}\rangle, \ldots \)
- terminals are: \( a, b, c, \ldots, \text{name}, \ldots \)
... a practical example:

\[
S \rightarrow \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle \rightarrow (\text{if}) \mid (\text{while}) \mid (\text{exp}) \\
(\text{if}) \rightarrow \text{if} (\langle \text{exp} \rangle) (\langle \text{stmt} \rangle) \text{ else } (\langle \text{stmt} \rangle) \\
(\text{while}) \rightarrow \text{while} (\langle \text{exp} \rangle) (\langle \text{stmt} \rangle) \\
(\text{exp}) \rightarrow \text{int} \mid (\langle \text{exp} \rangle) \mid (\langle \text{exp} \rangle) = (\langle \text{exp} \rangle) \mid ... \\
(\langle \text{exp} \rangle) \rightarrow \text{name} \mid ... \\
\]

More conventions:
- For every nonterminal, we collect the right hand sides of rules and list them together.
- The \( j \)-th rule for \( A \) can be identified via the pair \((A, j)\) (with \( j \geq 0 \)).

Pair of grammars:

<table>
<thead>
<tr>
<th>Production</th>
<th>( E \rightarrow E + E )</th>
<th>( E + E )</th>
<th>(( E ))</th>
<th>name</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow E + T )</td>
<td>( T )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T \rightarrow T * F )</td>
<td>( F )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( F \rightarrow (E) )</td>
<td>name</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Both grammars describe the same language.

Pair of grammars:

<table>
<thead>
<tr>
<th>Production</th>
<th>( E \rightarrow E + E )</th>
<th>( E + E )</th>
<th>(( E ))</th>
<th>name</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow E + T )</td>
<td>( T )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T \rightarrow T * F )</td>
<td>( F )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( F \rightarrow (E) )</td>
<td>name</td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Both grammars describe the same language.

Derivation:

Grammars are term rewriting systems. The rules offer feasible rewriting steps. A sequence of such rewriting steps \( \alpha_0 \rightarrow \ldots \rightarrow \alpha_m \) is called derivation.

... for example:

\[
E
\]
Pair of grammars:

<table>
<thead>
<tr>
<th>E</th>
<th>E+E</th>
<th>E*E</th>
<th>(E)</th>
<th>name</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E+E</td>
<td>E*E</td>
<td>(E)</td>
<td>name</td>
<td>int</td>
</tr>
<tr>
<td>T</td>
<td>T+F</td>
<td>F</td>
<td>name</td>
<td>int</td>
<td></td>
</tr>
</tbody>
</table>

Both grammars describe the same language

Derivation

Grammars are term rewriting systems. The rules offer feasible rewriting steps. A sequence of such rewriting steps $\alpha_0 \rightarrow \ldots \rightarrow \alpha_m$ is called derivation.

... for example: $E \rightarrow E + T$
Derivation

Grammars are term rewriting systems. The rules offer feasible
rewriting steps. A sequence of such rewriting steps $\alpha_0 \rightarrow \ldots \rightarrow \alpha_m$
is called derivation.

... for example:

\[
\begin{align*}
E & \rightarrow E + T \\
& \rightarrow T + T \\
& \rightarrow T \cdot E + T \\
& \rightarrow E \cdot \text{int} + T \\
& \rightarrow \text{name} \cdot \text{int} + T \\
& \rightarrow \text{name} \cdot \text{int} + E \\
& \rightarrow \text{name} \cdot \text{int} + \text{int}
\end{align*}
\]

Definition

The derivation relation $\rightarrow$ is a relation on words over $N \cup T$, with

$\alpha \rightarrow \alpha'$ iff $\alpha = \alpha_1 \beta_1 \alpha_2$ and $\alpha' = \alpha_1 \beta_2 \alpha_2$ for an $\alpha_1 \beta \in P$. 

... for example:

\[
\begin{align*}
E & \rightarrow E + T \\
& \rightarrow T + T \\
& \rightarrow T \cdot E + T \\
& \rightarrow T \cdot \text{int} + T \\
& \rightarrow \text{name} \cdot \text{int} + T \\
& \rightarrow \text{name} \cdot \text{int} + E \\
& \rightarrow \text{name} \cdot \text{int} + \text{int}
\end{align*}
\]
Derivation

Grammars are term rewriting systems. The rules offer feasible rewriting steps. A sequence of such rewriting steps \( \alpha_0 \rightarrow \ldots \rightarrow \alpha_m \) is called derivation.

... for example:

\[
\begin{align*}
E & \rightarrow E + T \\
& \rightarrow T + T \\
& \rightarrow T * E + T \\
& \rightarrow E * int + T \\
& \rightarrow name * int + T \\
& \rightarrow name * int + E \\
& \rightarrow name * int + int
\end{align*}
\]

Definition

The derivation relation \( \rightarrow \) is a relation on words over \( N \cup T \), with

\[
\alpha \rightarrow \alpha' \iff \alpha = \alpha_1 A \alpha_2 \land \alpha' = \alpha_1 \beta \alpha_2 \text{ for an } A \rightarrow \beta \in P
\]

The reflexive and transitive closure of \( \rightarrow \) is denoted as: \( \rightarrow^* \)

Remarks:

- The relation \( \rightarrow \) depends on the grammar
- In each step of a derivation, we may choose:
  * a spot, determining where we will rewrite.
  * a rule, determining how we will rewrite.
- The language, specified by \( G \) is:

\[
\mathcal{L}(G) = \{ w \in T^* \mid S \rightarrow^* w \}
\]

Attention:

The order, in which disjunct fragments are rewritten is not relevant.

Derivation Tree

Derivations of a symbol are represented as derivation trees:

... for example:

\[
\begin{align*}
E & \rightarrow^0 E + T \\
& \rightarrow^1 T + T \\
& \rightarrow^0 T * E + T \\
& \rightarrow^2 T * int + T \\
& \rightarrow^1 E * int + T \\
& \rightarrow^1 name * int + T \\
& \rightarrow^1 name * int + E \\
& \rightarrow^2 name * int + int
\end{align*}
\]

A derivation tree for \( A \in N \):

inner nodes: rule applications
root: rule application for \( A \)
leaves: terminals or \( \epsilon \)

The successors of \((B, i)\) correspond to right hand sides of the rule
Special Derivations

Attention:
In contrast to arbitrary derivations, we find special ones, always rewriting the leftmost or rather rightmost occurrence of a nonterminal.

- These are called leftmost (or rather rightmost) derivations and are denoted with the index $L$ or $R$, respectively).
- Leftmost (or rightmost) derivations correspond to a left-to-right (or right-to-left) preorder-DFS-traversal of the derivation tree.
- Reverse rightmost derivations correspond to a left-to-right postorder-DFS-traversal of the derivation tree.

... for example:

Leftmost derivation: $(E, 0) (E, 1) (T, 0) (T, 1) (F, 1) (F, 2) (T, 1) (F, 2)$
Rightmost derivation: $(E, 0) (T, 1) (F, 2) (E, 1) (T, 0) (F, 2) (T, 1) (F, 1)$
Special Derivations

... for example:

Leftmost derivation: \((E, 0) (E, 1) (T, 0) (T, 1) (F, 1) (F, 2) (T, 1) (F, 2)\)
Rightmost derivation: \((E, 0) (T, 1) (F, 2) (E, 1) (T, 0) (F, 2) (T, 1) (F, 1)\)
Reverse rightmost derivation: \((F, 1) (T, 1) (F, 2) (T, 0) (E, 1) (F, 2) (T, 1) (E, 0)\)

Unique Grammars

The concatenation of leaves of a derivation tree \(t\) are often called \(\text{yield}(t)\).

... for example:

Leftmost derivation:
- \(E\)
- \(+\)
- \(T\)
- \(*\)
- \(F\)
- \(int\)

Rightmost derivation:
- \(T\)
- \(+\)
- \(F\)
- \(*\)
- \(T\)
- \(E\)

Reverse rightmost derivation:
- \(F\)
- \(*\)
- \(E\)
- \(+\)
- \(T\)

Gives rise to the concatenation: \(\text{name} + \text{int} + \text{int}\).

Definition:
Grammar \(G\) is called **unique**, if for every \(w \in T^*\) there is maximally one derivation tree \(t\) of \(S\) with \(\text{yield}(t) = w\).

Unique grammars

... in our example:

Conclusion:

- A derivation tree represents a possible hierarchical structure of a word.
- For programming languages, only those grammars with a unique structure are of interest.
- Derivation trees are one-to-one corresponding with leftmost derivations as well as (reverse) rightmost derivations.

- **Leftmost derivations** correspond to a **top-down** reconstruction of the syntax tree.
- **Reverse rightmost derivations** correspond to a **bottom-up** reconstruction of the syntax tree.

The first one is ambiguous, the second one is unique.
Chapter 2: Basics of Pushdown Automata

Basics of Pushdown Automata

Languages, specified by context-free grammars are accepted by Pushdown Automata:

The pushdown is used e.g. to verify correct nesting of braces.

Example:

**States:** 0, 1, 2
**Start state:** 0
**Final states:** 0, 2

<table>
<thead>
<tr>
<th>State</th>
<th>Symbol</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>

Example:

**States:** 0, 1, 2
**Start state:** 0
**Final states:** 0, 2

<table>
<thead>
<tr>
<th>State</th>
<th>Symbol</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>

**Conventions:**
- We do not differentiate between pushdown symbols and states.
- The rightmost / upper pushdown symbol represents the state.
- Every transition consumes / modifies the upper part of the pushdown.
Definition: Pushdown Automaton

A pushdown automaton (PDA) is a tuple $M = (Q, T, \delta, q_0, F)$ with:
- $Q$ a finite set of states;
- $T$ an input alphabet;
- $q_0 \in Q$ the start state;
- $F \subseteq Q$ the set of final states and
- $\delta \subseteq Q^+ \times (T \cup \{ \varepsilon \}) \times Q^*$ a finite set of transitions

We define computations of pushdown automata with the help of transitions; a particular computation state (the current configuration) is a pair:

$$(\gamma, w) \in Q^* \times T^*$$

consisting of the pushdown content and the remaining input.

... for example:

<table>
<thead>
<tr>
<th>States:</th>
<th>0, 1, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start state:</td>
<td>0</td>
</tr>
<tr>
<td>Final states:</td>
<td>0, 2</td>
</tr>
</tbody>
</table>

$$(0, \ a\ a\ a\ b\ b\ b)$$

<table>
<thead>
<tr>
<th>States:</th>
<th>0, 1, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start state:</td>
<td>0</td>
</tr>
<tr>
<td>Final states:</td>
<td>0, 2</td>
</tr>
</tbody>
</table>

$$(0, \ a\ a\ a\ b\ b\ b) \downarrow (11, \ a\ b\ b\ b)$$
... for example:

States: 0, 1, 2
Start state: 0
Final states: 0, 2

\[
\begin{array}{c|cc}
0 & a & 11 \\
1 & a & 11 \\
11 & b & 2 \\
12 & b & 2 \\
\end{array}
\]

\[
(0, \ aabb) \vdash (11, \ aabbb) \\
\vdash (111, \ abb) \\
\vdash (1111, \ bbb) \\
\vdash (112, \ bb) \\
\vdash (2, \ c)
\]

... for example:

\[
\{a^n b^n | n \geq 0\}
\]
A computation step is characterized by the relation \( \vdash \subseteq (Q^* \times T^*)^2 \)
with
\[
(\alpha \gamma, \left[ w \right]) \vdash (\alpha \gamma', \left[ w \right]) \quad \text{for} \quad (\gamma, \left[ x \right], \gamma') \in \delta
\]

Remarks:

- The relation \( \vdash \) depends on the pushdown automaton \( M \)
- The reflexive and transitive closure of \( \vdash \) is denoted by \( \vdash^* \)
- Then, the language accepted by \( M \) is

\[
\mathcal{L}(M) = \{ w \in T^* | \exists f \in F : (q_0, w) \vdash^* (f, \varepsilon) \}
\]

We accept with a final state together with empty input.