Berry-Sethi Approach: (sophisticated version)

Construction (sophisticated version):
Create an automaton based on the syntax tree’s new attributes:

- **States:** 
  \[
  \{c\} \cup \{i \mid i \text{ a leaf}\}
  \]
- **Start state:** 
  \[
  c
  \]
- **Final states:** 
  \[
  \{c\} \cup \text{last}[e] \quad \text{if empty}[e] = f
  \]
  \[
  \{c\} \cup \text{last}[e] \quad \text{otherwise}
  \]
- **Transitions:** 
  \[
  (c, a, i) \quad \text{if } i \in \text{first}[e] \text{ and } i \text{ labeled with } a.
  \]
  \[
  (i, a, i') \quad \text{if } i' \in \text{next}[i] \text{ and } i' \text{ labeled with } a.
  \]

We call the resulting automaton \( A_c \).

Berry-Sethi Approach

... for example:

![Diagram](image1)

Remarks:
- This construction is known as Berry-Sethi- or Glushkov-construction.
- It is used for XML to define Content Models
- The result may not be, what we had in mind...

Powerset Construction

... for example:

![Diagram](image2)
Powerset Construction

... for example:

```
0 -> a
1 -> b
2 -> a
3 -> a
4 -> b
```

Theorem:
For every non-deterministic automaton \( A = (Q, \Sigma, \delta, I, F) \) we can compute a deterministic automaton \( \mathcal{P}(A) \) with

\[
\mathcal{L}(A) = \mathcal{L}(\mathcal{P}(A))
\]

Powerset Construction

Observation:
There are exponentially many powersets of \( Q \)

- Idea: Consider only contributing powersets. Starting with the set \( Q_P = \{ I \} \) we only add further states by need ...
- i.e., whenever we can reach them from a state in \( Q_P \)
- However, the resulting automaton can become enormously huge ...
  which is (sort of) not happening in practice
Remarks:

- For an input sequence of length \( n \), maximally \( O(n) \) sets are generated.
- Once a set/edge of the DFA is generated, they are stored within a hash-table.
- Before generating a new transition, we check this table for already existing edges with the desired label.

Summary:

**Theorem:**

For each regular expression \( e \) we can compute a deterministic automaton \( A = \mathcal{P}(A_e) \) with

\[ \mathcal{L}(A) = [e] \]

---

Lexical Analysis

Chapter 5: Scanner design

---

**Powerset Construction**

... for example:

![Diagram with states and transitions labeled a, b, a, b]
Berry-Sethi Approach

... for example:

\[(a|b)^* a (a|b)\]

Berry-Sethi Approach: (sophisticated version)

Construction (sophisticated version):
Create an automaton based on the syntax tree's new attributes:

States: \{\(\varepsilon\)\} \cup \{i \ast i \mid a \text{ leaf}\}
Start state: \(\varepsilon\)
Final states: last[\(\varepsilon\)] if empty[\(\varepsilon\)] = \(f\)
\{\(\varepsilon\)\} \cup last[\(\varepsilon\)] otherwise
Transitions: (\(\varepsilon, a, i \ast i\)) if \(i \in \text{first}[\(\varepsilon\)]\) and \(i\) labeled with \(a\).
(i \ast i, a, i' \ast i') if \(i' \in \text{next}[i]\) and \(i'\) labeled with \(a\).

We call the resulting automaton \(A_c\).

Berry-Sethi Approach

In general:

- Input is only consumed at the leaves.
- Navigating the tree does not consume input \(\Rightarrow \varepsilon\)-transitions
- For a formal construction we need identifiers for states.
- For a node \(n's\) identifier we take the subexpression, corresponding to the subtree dominated by \(n\).
- There are possibly identical subexpressions in one regular expression.

\[\Rightarrow \quad \text{we enumerate the leaves} \ldots\]

Berry-Sethi Approach

... for example:

Berry-Sethi Approach

Remarks:

- This construction is known as Berry-Sethi- or Glushkov-construction.
- It is used for XML to define Content Models
- The result may not be, what we had in mind...
Implementation:

Idea:

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

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

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

Implementation (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("Hallo");
```

Input (generalized):

A set of rules:

\[
\langle \text{state} \rangle \{ \begin{array}{ll}
\text{e}_1 & \{ \text{action}_1 \text{yybegin(state}_1); \} \\
\text{e}_2 & \{ \text{action}_2 \text{yybegin(state}_2); \}
\end{array}
\ldots
\begin{array}{ll}
\text{e}_k & \{ \text{action}_k \text{yybegin(state}_k); \}
\end{array}\}
\]

- The statement `yybegin(state_i);` resets the current state to `state_i`.
- The start state is called (e.g. `flex JFlex`) `YYINITIAL`.

... for example:

```plaintext
\langle YYINITIAL \rangle
\langle COMMENT \rangle \{ \\n"/*" \\n\{ \\n\{ yybegin(COMMENT); \} \\
\} \\
\{ yybegin(YYINITIAL); \}
```

...
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.

Discussion:

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

![Diagram](generated.png)

**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

Conventions

The rules of context-free grammars take the following form:

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

... a practical example:

$$
\begin{align*}
S & \rightarrow (\text{stmt}) \\
(\text{stmt}) & \rightarrow (\text{if}) | (\text{while}) | (\text{rexp}) \\
(\text{if}) & \rightarrow \text{if} (\text{rexp}) (\text{stmt}) \text{else} (\text{stmt}) \\
(\text{while}) & \rightarrow \text{while} (\text{rexp}) (\text{stmt}) \\
(\text{rexp}) & \rightarrow \text{int} | (\text{rexp}) | (\text{rexp}) = (\text{rexp}) \\
(\text{expr}) & \rightarrow \text{name} | ... \\
\end{align*}
$$

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

Conventions:

In examples, we specify nonterminals and terminals in general implicitly:

- nonterminals are: $A, B, C, ..., (\text{expr}), (\text{stmt}), ...$
- terminals are: $a, b, c, ..., \text{int}, \text{name}, ...$
... a practical example:

\[
\begin{align*}
S & \rightarrow \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle & \rightarrow \langle \text{if} \rangle \mid \langle \text{while} \rangle \mid \langle \text{rexp} \rangle \\
\langle \text{if} \rangle & \rightarrow \text{if} \langle \text{rexp} \rangle \langle \text{stmt} \rangle \text{else} \langle \text{stmt} \rangle \\
\langle \text{while} \rangle & \rightarrow \text{while} \langle \text{rexp} \rangle \langle \text{stmt} \rangle \\
\langle \text{rexp} \rangle & \rightarrow \langle \text{rexp} \rangle \mid \langle \text{lexp} \rangle \mid \langle \text{lexp} \rangle = \langle \text{rexp} \rangle \mid \ldots \\
\langle \text{lexp} \rangle & \rightarrow \text{name} \mid \ldots
\end{align*}
\]

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 \(j \geq 0\)).

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:

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

Definition

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

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

Pair of grammars:

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

Both grammars describe the same language

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:

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

Derivations of a symbol are represented as derivation trees:

... for example:

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

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.

Unique Grammars

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

... for example:

```
gives rise to the concatenation: name * int + int.
```
Unique Grammars

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

... in our example:

\[
\begin{align*}
E & \rightarrow E + E^0 \mid E * E^1 \mid (E)^2 \mid \text{name}^3 \mid \text{int}^4 \\
E & \rightarrow E + T^0 \mid T^1 \\
T & \rightarrow T * F^0 \mid F^1 \\
F & \rightarrow (E)^0 \mid \text{name}^1 \mid \text{int}^2
\end{align*}
\]

The first one is ambiguous, the second one is unique

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.

Syntactic Analysis

Chapter 2:
Basics of Pushdown Automata