Script **generated by TTT**

Title: Petter: Compilerbau (11.05.2017)

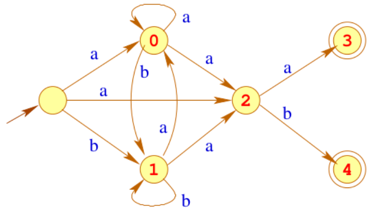
Date: Thu May 11 14:18:10 CEST 2017

Duration: 90:19 min

Pages: 34

Berry-Sethi Approach

... for example:

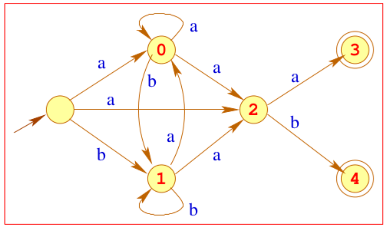


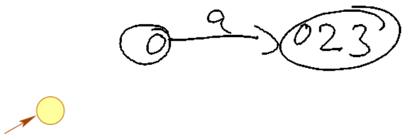
Remarks:

- 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

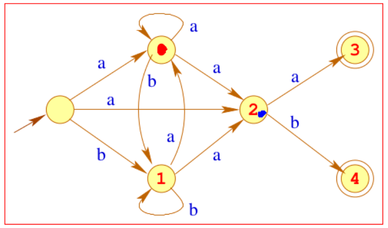
... for example:

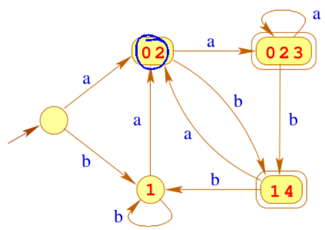




Powerset Construction

... for example:





Powerset Construction

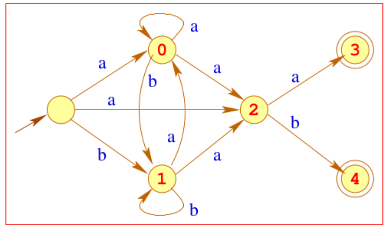
Theorem:

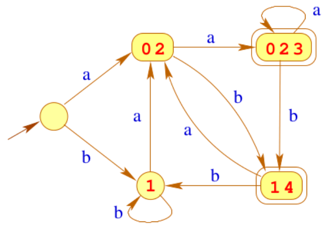
For every non-deterministic automaton $A = (Q, \Sigma, \delta, I, F)$ we can compute a deterministic automaton $\mathcal{P}(A)$ with

$$\mathcal{L}(A) = \mathcal{L}(\mathcal{P}(A))$$

Powerset Construction

... for example:





Powerset Construction

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

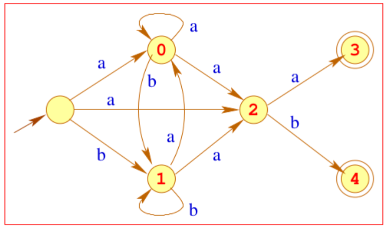
Observation:

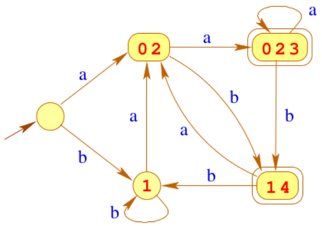
There are exponentially many powersets of Q

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

Powerset Construction

... for example:





Powerset Construction

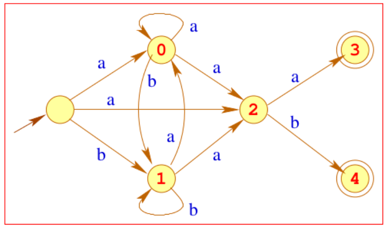
Observation:
There are exponentially many powersets of Q

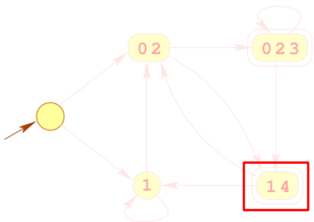
- Idea: Consider only contributing powersets. Starting with the set $Q_{\mathcal{P}} = \{I\}$ we only add further states by need ...
- i.e., whenever we can reach them from a state in $Q_{\mathcal{P}}$
- However, the resulting automaton can become enormously huge ... which is (sort of) not happening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

Powerset Construction

... for example:

a	b	a	b





Remarks:

- For an input sequence of length n, maximally $\mathcal{O}(n)$ 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.

integer

Chapter 5:
Scanner design

Implementation:

Idea:

- Create the DFA $\mathcal{P}(A_e) = (Q, \Sigma, \delta, q_0, F)$ for the expression $e = \boxed{e_1 \mid \ldots \mid e_k}$;
- Define the sets:

$$
\begin{array}{lll}
\boxed{F_1} & = & \{\boxed{q \in F} \mid q \cap \boxed{\mathsf{last}[e_1]} \neq \emptyset\} \\
\boxed{F_2} & = & \{\boxed{q \in (F \backslash F_1)} \mid q \cap \boxed{\mathsf{last}[e_2]} \neq \emptyset\} \\
 & \ldots & \\
F_k & = & \{q \in (F \backslash (F_1 \cup \ldots \cup F_{k-1})) \mid q \cap \mathsf{last}[e_k] \neq \emptyset\}
\end{array}
$$

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

Implementation:

Idea (cont'd):

- The scanner manages two pointers $\langle A, B\rangle$ and the related states $\langle q_A, q_B\rangle$...
- 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.

s	t	d	o	u	t	.	w	r	i	t	e	l	n		(	"	H	a	l	l	o	"	)	;

4

A	B

Extension: States

while // while

- 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:

```
⟨state⟩ {  e1      { action1   yybegin(state1); }
           e2      { action2   yybegin(state2); }
               ...
           ek      { actionk   yybegin(statek); }
        }
```

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

... for example:

```
⟨YYINITIAL⟩    "/*"    { yybegin(COMMENT); }
⟨COMMENT⟩   {  "*/"    { yybegin(YYINITIAL); }
               . | \n  {  }
            }
```


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:

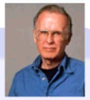
Chapter 1:

Basics of Contextfree 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...

Noam Chomsky



John Backus

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:

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

Conventions

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

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

... for example:

$$S \to a\,S\,b$$
$$S \to \epsilon$$

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

Conventions:

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

- nonterminals are: $A, B, C, \ldots, \langle\text{exp}\rangle, \langle\text{stmt}\rangle, \ldots$;
- terminals are: $a, b, c, \ldots$, int, name, ...;

... a practical example:

$$
\begin{array}{lcl}
S & \to & \langle\text{stmt}\rangle \\
\langle\text{stmt}\rangle & \to & \langle\text{if}\rangle \mid \langle\text{while}\rangle \mid \langle\text{rexp}\rangle; \\
\langle\text{if}\rangle & \to & \text{if } (\langle\text{rexp}\rangle) \ \langle\text{stmt}\rangle \text{ else } \langle\text{stmt}\rangle \\
\langle\text{while}\rangle & \to & \text{while } (\langle\text{rexp}\rangle) \ \langle\text{stmt}\rangle \\
\langle\text{rexp}\rangle & \to & \text{int} \mid \langle\text{lexp}\rangle \mid \langle\text{lexp}\rangle = \langle\text{rexp}\rangle \mid \ldots \\
\langle\text{lexp}\rangle & \to & \text{name} \mid \ldots
\end{array}
$$

Stmt → <if>
stmt → <while>
stmt → <rexp>

... a practical example:

S	$\to$	⟨stmt⟩
$\boxed{\text{⟨stmt⟩}}$	$\to$	⟨if⟩ 0 | $\boxed{\text{⟨while⟩}}$ 1 | ⟨rexp⟩ 2
⟨if⟩	$\to$	if (⟨rexp⟩) ⟨stmt⟩ else ⟨stmt⟩
⟨while⟩	$\to$	while (⟨rexp⟩) ⟨stmt⟩
⟨rexp⟩	$\to$	int | ⟨lexp⟩ | ⟨lexp⟩ = ⟨rexp⟩ | ...
⟨lexp⟩	$\to$	name | ...

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 $j \geq 0$).

(⟨stmt⟩, 1)

Pair of grammars:

E	$\to$	$E{+}E$ | $E{*}E$ | $(\ E\ )$ | name | int
E	$\to$	$E{+}T$ | T
T	$\to$	$T{*}F$ | F
F	$\to$	$(\ E\ )$ | name | int

Both grammars describe the same language

Derivation

Remarks:

- The relation $\to$ 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:

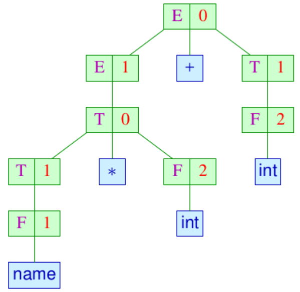
$$\mathcal{L}(G) = \{\boxed{w \in T^*} \mid \boxed{S \to^* w}\}$$

Derivation Tree

Derivations of a symbol are represented as derivation trees:

... for example:

$$E \xrightarrow{0} E+T \xrightarrow{1} T+T \xrightarrow{0} T*\boxed{F}+T \xrightarrow{2} \boxed{T}*\text{int}+T \xrightarrow{1} F*\text{int}+T \xrightarrow{1} \text{name}*\text{int}+T \xrightarrow{1} \text{name}*\text{int}+F \xrightarrow{2} \text{name}*\text{int}+\text{int}$$



A derivation tree for $A \in N$:
inner nodes: rule applications
root: rule application for A
leaves: terminals or ϵ
The successors of (B, i) correspond to right hand sides of the rule

Special Derivations

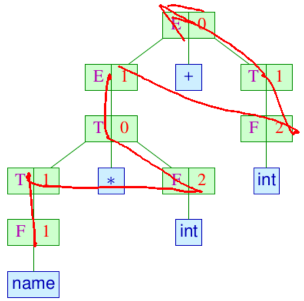
Attention:

In contrast to arbitrary derivations, we find special ones, always rewriting the leftmost (or rather rightmost) occurance 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 correspondt 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

Special Derivations

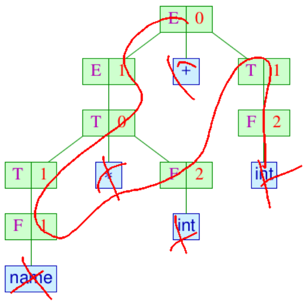
... for example:



Special Derivations

LL

... for example:



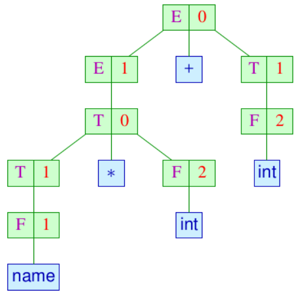
LR

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 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 $\mathsf{yield}(t) = w$.

... in our example:

E	$\to$	$E{+}E$ [0]	|	$E{*}E$ [1]	|	$(\,E\,)$ [2]	| name [3]	| int [4]

E	$\to$	$E{+}T$ [0]	|	T [1]		
T	$\to$	$T{*}F$ [0]	|	F [1]		
F	$\to$	$(\,E\,)$ [0]	|	name [1]	|	int [2]

E + E + E

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.

LL

LR

Chapter 2:
Basics of Pushdown Automata