Topic: Syntactic Analysis

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:

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

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

Derivation

The rewriting relation $\rightarrow$ is a relation on words over $N \cup T$, with
$$\alpha \rightarrow \alpha' \iff \alpha = A_1 A_2 \ldots A_n \land \alpha' = A_1 A_2 \ldots A_n$$
for an $A_i \rightarrow \beta \in P$.

The reflexive and transitive closure of $\rightarrow$ is denoted as $\rightarrow^*$.
A derivation tree for \( A \in X \):

- Inner nodes: rule applications
- Leaves: terminal or \( \epsilon \)
- The successors of \((B,i)\) correspond to right hand sides of the rule

**Leftmost derivation:**
- Inner nodes: rule application for \( \epsilon \)
- Leaves: terminals or \( \epsilon \)

The first one is ambiguous, the second one is unique

**Special Derivations**

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

**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.

**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.

**Definition: Pushdown Automaton**

A pushdown automaton (PDA) is a tuple \( \mathcal{M} = (Q,T,E,Q_0,F) \) with:

- \( Q \) a finite set of states;
- \( T \) an input alphabet;
- \( E \in Q \) the start state;
- \( F \subseteq Q \) the set of final states and
- \( \delta \subseteq Q \times (T \cup \{\epsilon\}) \times Q^* \) a finite set of transitions

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

\[ \langle q, w \rangle \in Q \times T^* \]

consisting of the pushdown content and the remaining input.
The item pushdown automaton shifts the bullet around the derivation tree ...

(0, ϵ, a bbb) ⊢ (1, a bbb)
(1, a bbb) ⊢ (11, bbb)
(11, bbb) ⊢ (112, kb)
(112, kb) ⊢ (12, k)
(12, k) ⊢ (2, ϵ)

Definition: Deterministic Pushdown Automaton
The pushdown automaton \( M \) is deterministic, if every configuration has maximally one successor configuration.

This is exactly the case if for distinct transitions \( (\gamma, x, \gamma'), (\gamma'', x, \gamma') \in \delta \) we can assume:
Is \( \gamma \) a suffix of \( \gamma' \), then \( x \neq x' \) and \( x \neq x' \) is valid.

... for example:

<table>
<thead>
<tr>
<th>States: 0, 1, 2</th>
<th>Final states: 0, 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, a b bbb)</td>
<td>(0, a b bbb)</td>
</tr>
<tr>
<td>(1, a b bbb)</td>
<td>(1, a b bbb)</td>
</tr>
<tr>
<td>(11, bbb)</td>
<td>(11, bbb)</td>
</tr>
<tr>
<td>(112, kb)</td>
<td>(112, kb)</td>
</tr>
<tr>
<td>(12, k)</td>
<td>(12, k)</td>
</tr>
<tr>
<td>(2, ϵ)</td>
<td>(2, ϵ)</td>
</tr>
</tbody>
</table>

Syntactic Analysis

Chapter 3:
Top-down Parsing

Item Pushdown Automaton – Example

Our example:

\[ S \rightarrow AB^0 \quad A \rightarrow a^2 \quad B \rightarrow b^2 \]

Item Pushdown Automaton

The item pushdown automaton \( M \) has three kinds of transitions:

Expansions:
\[ [A \rightarrow \alpha \quad B \beta, x] [A \rightarrow \alpha \quad B \beta, \gamma] : \quad \text{for} \]
\[ A \rightarrow \alpha \quad B \beta, \quad \gamma \in \delta \]

Shifts:
\[ [A \rightarrow \alpha \quad B \beta, \gamma] : \quad \text{for} \quad A \rightarrow \alpha \quad B \beta, \quad \gamma \in \delta \]

Reduces:
\[ [A \rightarrow \alpha \quad B \beta, \gamma] : \quad \text{for} \quad A \rightarrow \alpha \quad B \beta, \quad \gamma \in \delta \]

Items of the form: \( [A \rightarrow \alpha \quad \bullet] \) are also called complete

Item Pushdown Automaton

Discussion:

- The expansions of a computation form a leftmost derivation
- Unfortunately, the expansions are chosen nondeterministically
- For proving correctness of the construction, we show that for every item the following holds:
  \[ ([A \rightarrow \alpha \quad B \beta, w] \gamma) ^+ [A \rightarrow \alpha \quad B \beta, x] \quad \text{if} \quad B \rightarrow \gamma w \]

Item Pushdown Automaton

Pushdown Automata

Theorem:
For each context free grammar \( G = (N, T, P, S) \) a pushdown automaton \( M \) with \( L(M) = L(G) \) can be built.

The theorem is so important for us, that we take a look at two constructions for automata, motivated by both of the special derivations:
- \( LL \) to build Leftmost derivations
- \( LR \) to build Rightmost derivations

Item Pushdown Automaton

Construction: Item Pushdown Automaton \( M \)
- Reconstruct a Leftmost derivation.
- Expand nonterminals using a rule.
- Verify successively, that the chosen rule matches the input.

We accept with a final state together with empty input.
Item Pushdown Automaton

Example: $S' \rightarrow S \delta \quad S \rightarrow \epsilon | aSb$

The transitions of the according Item Pushdown Automaton:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>${S',S}$</th>
<th>${S}$</th>
<th>${\epsilon}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S' \rightarrow S \delta$</td>
<td>${S',S}$</td>
<td>${S}$</td>
<td>${\epsilon}$</td>
</tr>
<tr>
<td>$S \rightarrow \epsilon</td>
<td>aSb$</td>
<td>${\epsilon}$</td>
<td>${\epsilon}$</td>
</tr>
</tbody>
</table>

Conflicts arise between the transitions $(0,1)$ and $(3,4)$, resp.

Topdown Parsing

Problem:
Conflicts between the transitions prohibit an implementation of the item pushdown automation as deterministic pushdown automation.

Idea 1: GLL Parsing
For each conflict, we create a virtual copy of the complete configuration and continue computing in parallel.

Idea 2: Recursive Descent & Backtracking
Depth-first search for an appropriate derivation.

Idea 3: Recursive Descent & Lookahead
Conflicts are resolved by considering a lookup of the next input symbols.

Lookahead Sets

Definition: First₁-Sets
For a set $L \subseteq T^*$ we define:

$$\text{First}_1(L) = \{ \epsilon | \epsilon \in L \} \cup \{ u \in T | \exists v \in T^* : uv \in L \}$$

Example: $S \rightarrow \epsilon | aSb$

Lookahead Sets

Arithmetics:
For $\alpha \in (N \cup T)^*$ we are interested in the set:

$$\text{First}_1(\alpha) = \text{First}_1(\{ w \in T^* | \alpha \rightarrow^* w \})$$

Idea: Treat $\epsilon$ separately: $\text{First}_1(\epsilon) = \{ \epsilon \}$

Definition: 1-concatenation
Let $L_1, L_2 \subseteq T \cup \{ \epsilon \}$ with $L_1 \neq \emptyset \neq L_2$. Then:

$$L_1 \sqcap L_2 = \begin{cases} L_1 \cup L_2 & \text{if } \epsilon \notin L_1 \\ L_1 \cup L_2 & \text{otherwise} \end{cases}$$

If all rules of $G$ are productive, then all sets $\text{First}_1(A)$ are non-empty.

Lookahead Sets

Fast Computation of Lookahead Sets

Observation:
The form of each inequality of these systems is:

$$x \sqcup y \text{ resp. } x \sqcap d$$

for variables $x, y$ and $d \in D$

Such systems are called pure unification problems
Such problems can be solved in linear space/time.

for example:

$$\begin{align*}
x_1 \geq 1 & \quad (a) \\
x_2 \geq 2 & \quad (b) \\
x_3 \geq 3 & \quad (c) \\
x_3 \geq 2 & \quad (d)
\end{align*}$$
Fast Computation of Lookahead Sets

... for our example grammar:
First_1:

Left Recursion

Theorem:

Proof:

Case 1: \( \beta \Rightarrow^* \epsilon \) — Contradiction
Case 2: \( \beta \Rightarrow^* \epsilon \) — Contradiction

Idea 1: Rewrite the rules from \( G \) to \( \tilde{G} \):

Example: Arithmetic Expressions (cont'd)

Fast Computation of Lookahead Sets

Proceeding:

- Create the Variable Dependency Graph for the inequality system,
- Within a Strongly Connected Component (\(-\) Tarjan) all variables have the same value,
- Is there no incoming edge for an SCC, its value is computed via the smallest upper bound of all values within the SCC,
- In case of incoming edges, their values are also to be considered for the upper bound

Item Pushdown Automaton as LL(1)-Parser

Is \( G \) an LL(1)-grammar, we can index a lookahead-table with items and nonterminals:

LL(1)-Lookahead Table

We set \( M[u,v] = i \) with \( B \rightarrow v \) if \( u \in \text{First}_1(\gamma) \cap \text{Follow}_1(\delta) \)

... for example:

\[
S \rightarrow S \; \Rightarrow \; S \rightarrow b \; \mid \; aSb \\
\text{First}_1(S) = \{a, b\} \quad \text{Follow}_1(S) = \{b, \}$
\]

S-rule 0: \( \text{First}_1(S) \cap \text{Follow}_1(S) = \{b, \}$
S-rule 1: \( \text{First}_1(aSb) \cap \text{Follow}_1(S) = \{a\}$

Lookahead table:

Item Pushdown Automaton as LL(1)-Parser

Recurring scheme in programming languages: Lists of sth...

Within a Strongly Connected Component (\(-\) Tarjan) all variables have the same value:

- Is there no incoming edge for an SCC, its value is computed via the smallest upper bound of all values within the SCC,
- In case of incoming edges, their values are also to be considered for the upper bound

Item Pushdown Automaton as LL(1)-Parser

Inequality system for \( \text{Follow}_1(\delta) = \text{First}_1(\gamma) \cap \text{Follow}_1(\delta) \)

Create the Variable Dependency Graph for the inequality system.

Item Pushdown Automaton as LL(1)-Parser

Fast Computation of Lookahead Sets

... for our example grammar:
First_1:

Left Recursion

Theorem:

Proof:

Case 1: \( \beta \Rightarrow^* \epsilon \) — Contradiction
Case 2: \( \beta \Rightarrow^* \epsilon \) — Contradiction

Idea 1: Rewrite the rules from \( G \) to \( \tilde{G} \):

Example: Arithmetic Expressions (cont'd)

Fast Computation of Lookahead Sets

Proceeding:

- Create the Variable Dependency Graph for the inequality system,
- Within a Strongly Connected Component (\(-\) Tarjan) all variables have the same value,
- Is there no incoming edge for an SCC, its value is computed via the smallest upper bound of all values within the SCC,
- In case of incoming edges, their values are also to be considered for the upper bound

Item Pushdown Automaton as LL(1)-Parser

Is \( G \) an LL(1)-grammar, we can index a lookahead-table with items and nonterminals:

LL(1)-Lookahead Table

We set \( M[u,v] = i \) with \( B \rightarrow v \) if \( u \in \text{First}_1(\gamma) \cap \text{Follow}_1(\delta) \)

... for example:

\[
S \rightarrow S \; \Rightarrow \; S \rightarrow b \; \mid \; aSb \\
\text{First}_1(S) = \{a, b\} \quad \text{Follow}_1(S) = \{b, \}$
\]

S-rule 0: \( \text{First}_1(S) \cap \text{Follow}_1(S) = \{b, \}$
S-rule 1: \( \text{First}_1(aSb) \cap \text{Follow}_1(S) = \{a\}$

Lookahead table:

Item Pushdown Automaton as LL(1)-Parser

Recurring scheme in programming languages: Lists of sth...

Within a Strongly Connected Component (\(-\) Tarjan) all variables have the same value:

- Is there no incoming edge for an SCC, its value is computed via the smallest upper bound of all values within the SCC,
- In case of incoming edges, their values are also to be considered for the upper bound

Item Pushdown Automaton as LL(1)-Parser

Inequality system for \( \text{Follow}_1(\delta) = \text{First}_1(\gamma) \cap \text{Follow}_1(\delta) \)

Create the Variable Dependency Graph for the inequality system.

Item Pushdown Automaton as LL(1)-Parser

Fast Computation of Lookahead Sets

... for our example grammar:
First_1:

Left Recursion

Theorem:

Proof:

Case 1: \( \beta \Rightarrow^* \epsilon \) — Contradiction
Case 2: \( \beta \Rightarrow^* \epsilon \) — Contradiction

Idea 1: Rewrite the rules from \( G \) to \( \tilde{G} \):

Example: Arithmetic Expressions (cont'd)

Fast Computation of Lookahead Sets

Proceeding:

- Create the Variable Dependency Graph for the inequality system,
- Within a Strongly Connected Component (\(-\) Tarjan) all variables have the same value,
- Is there no incoming edge for an SCC, its value is computed via the smallest upper bound of all values within the SCC,
- In case of incoming edges, their values are also to be considered for the upper bound

Item Pushdown Automaton as LL(1)-Parser

Is \( G \) an LL(1)-grammar, we can index a lookahead-table with items and nonterminals:

LL(1)-Lookahead Table

We set \( M[u,v] = i \) with \( B \rightarrow v \) if \( u \in \text{First}_1(\gamma) \cap \text{Follow}_1(\delta) \)

... for example:

\[
S \rightarrow S \; \Rightarrow \; S \rightarrow b \; \mid \; aSb \\
\text{First}_1(S) = \{a, b\} \quad \text{Follow}_1(S) = \{b, \}$
\]

S-rule 0: \( \text{First}_1(S) \cap \text{Follow}_1(S) = \{b, \}$
S-rule 1: \( \text{First}_1(aSb) \cap \text{Follow}_1(S) = \{a\}$

Lookahead table:

Item Pushdown Automaton as LL(1)-Parser

Recurring scheme in programming languages: Lists of sth...

Within a Strongly Connected Component (\(-\) Tarjan) all variables have the same value:

- Is there no incoming edge for an SCC, its value is computed via the smallest upper bound of all values within the SCC,
- In case of incoming edges, their values are also to be considered for the upper bound

Item Pushdown Automaton as LL(1)-Parser

Inequality system for \( \text{Follow}_1(\delta) = \text{First}_1(\gamma) \cap \text{Follow}_1(\delta) \)

Create the Variable Dependency Graph for the inequality system.
Idea 2: Recursive Descent RLL Parsers:

For each $A \rightarrow \alpha \in P$, we introduce:

```cpp
void A(){
generate(\alpha)
}
```

with the meta-program `generate` being defined by structural decomposition of $\alpha$:

```cpp
generate(\alpha_1 \ldots \alpha_k) = generate(\alpha_1)
generate(\alpha) = next = scan()
generate(A) = A();
generate(\epsilon) = ;
generate(r_k) = while (next \in F_\epsilon(r_k)) {
generate(r_k)
}
generate(r_1 | \ldots | r_k) = \ldots generate(r_k) break ;
``` 

labels({\alpha_1, \ldots, \alpha_m}) = label(\alpha_1): \ldots label(\alpha_m):

- label(\alpha) = case \alpha
- label(\epsilon) = default

Discussion

Recursive descent RLL(1)-parsers are an alternative to table-driven parsers; apart from the usual function `scan()`, we generate a program frame with the lookahead function `expect()` and the main parsing method `parse()`:

```cpp
int next;
void expect(Set E){
    if ((next) \in E) {
        cerr << "Expected " << E << " found" << next;
        exit(0);
    }
    return ;
}
void parse(){
    next = scan();
    expect(First(\epsilon));
    S();
    expect([EOF]);
}
```

Top-down Parsing

Discussion

- A practical implementation of an RLL(1)-parser via recursive descent is a straight-forward idea.
- However, only a subset of the deterministic context-free languages can be parsed this way.
- As soon as `First()` sets are not disjoint any more,
  - Solution 1: For many accessibility written grammars, the alternation between right hand sides happens too early. Keeping the common prefixes of right hand sides joined and introducing a new production for the actual diverging sentence forms often helps.
  - Solution 2: Introduce `ranked` grammars, and decide conflicting lookahead always in favour of the higher ranked alternative.
- Solution 3: Going from `left to right`.
  - The size of the occurring sets is rapidly increasing with larger $k$.
  - Unfortunately, even `ranked` parsers are not sufficient to accept all deterministic context-free languages.
- In practical systems, this often motivates the implementation of $k = 1$ only ...