Syntactical analysis

- Natural languages are ambiguous (unlike LL(k) or LR(k)).
- Disambiguation should take place on a semantical basis

Alternatives

- Backtrack parser:
  - Only non left recursive grammars
  - Time: $O(n^k), n = \text{length of input}$
  - Use of First information can speedup parsing
  - Error messages and error recovery are difficult

- Generalized LR:
  - Try alternatives in parallel
  - Share as much as possible

Tomita’s algorithm:

- Generalisation of LR parsing
- Allows arbitrary context-free grammars
- Time complexity:
  - $O(n^3)$ (general case)
  - $O(n^2)$ (non ambiguous grammars)
  - $O(n)$ (LL(1) and LR(1) grammars)

A backtrack parser tries all alternatives one after the other for each non-terminal:

- this causes problems in case of left recursive grammars
- a lot of work has to be redone
- Main idea: *Try all alternatives in parallel*
- No double work needs to be done
- In case of syntax errors none of the parses can proceed
Syntactical analysis

Tomita's parser

Input: \[ S_0 \]

Stack:

- \[ S_0 \rightarrow X_m \]
- \[ S_0 \rightarrow X_m \]

Tomita parser

Output

Action table

Goto table

Tomita's method is a generalization of LR parsing

- The entries in the action table may have multiple elements: (shift/reduce, reduce/reduce conflicts)
- At each multiple entry in the action table the stack is split into two. It is not necessary to split the entire stack, the parts which are not affected need not to be split
- All parallel parsers are synchronized on the shift actions
- The parse stacks have the form of a graph, graph structured stack (GSS)

Syntactical analysis

- To describe Tomita's algorithm in more detail a different stack format is introduced.
- LR stack is represented as a graph (without cycles).
- States on the stack are represented by circles:

  The tops of the Tomita stacks are represented by double circles, they are called active states: \[ S_0 \]

  The grammar symbols on the stack are represented by boxes: \[ A \]

  This format gives insight in sharing and splitting of the stack.

Syntactical analysis

Tomita's algorithm:

1. Construct a SLR table
2. The initial stack consists of one node labeled with the start state
3. Repeat the next actions as long as there are active states for which a shift or reduce action is possible
   a) As long as there is a reduce action in one of the active nodes possible, select one of them:
      - Suppose rule \[ w_1 \ldots w_n \rightarrow A \] must be reduced
Syntactical analysis

• Then remove, starting with the active node, in all possible ways $w_1, \ldots, w_n$ from the stack together with intermediate state nodes. The state nodes on the tops of the stacks become active.

Example:

\[
\text{Action: action}[s_5, a] = \text{reduce } w_1w_2 \rightarrow A
\]

Syntactical analysis

• Perform for each of the active states $s_i$ the \text{goto}[$s_i, A$]. However, combine those which yield the same state.

Example:

\[
\text{Action: goto}[s_2, A] = s_5 \text{ and goto}[s_3, A] = s_5
\]

Syntactical analysis

• If the \text{goto}[$s_j, A$] yields an active state which already existed on one of the other stacks and this already existing active state is also the result of a \text{goto}[$s_j, A$] then these are combined as well.

Example:

\[
\text{Action: goto}[s_2, A] = s_4 \text{ and goto}[s_3, A] = s_4
\]

b) If in all active nodes only \textit{shift} actions are possible, perform the shift.

• Combine those \text{action}[$s_1, b$] which yield the same state.

Example:

\[
\text{Action: action}[s_1, b] = s_5, \text{ action}[s_2, b] = s_5 \text{ and action}[s_3, b] = s_4
\]
4. If there are no more active nodes in which a reduce or shift action is possible, the parser is finished. If in one of the active nodes an accept action is possible, then the parse was successful.

- SLR(1) table for

<table>
<thead>
<tr>
<th></th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s2</td>
<td>s3</td>
</tr>
<tr>
<td>1</td>
<td>r0</td>
<td>r1</td>
</tr>
<tr>
<td>2</td>
<td>s2</td>
<td>s3</td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>r3</td>
</tr>
<tr>
<td>4</td>
<td>s2</td>
<td>s3</td>
</tr>
<tr>
<td>5</td>
<td>r2</td>
<td>r1</td>
</tr>
<tr>
<td>6</td>
<td>r2</td>
<td>r1</td>
</tr>
<tr>
<td>7</td>
<td>acc</td>
<td></td>
</tr>
</tbody>
</table>

- Input sentence: t | f & f

- Syntactical analysis

```
0  t  r2  t f & t $
0  B  s5  t f & t $
0  B  s3  f & t $
0  B  s3  f & t $
0  B  s4/r2  & t $
0  B  s4  & t $
0  B  s4  & t $
```

- Syntactical analysis
Syntactical analysis

• This generalization of a deterministic parser (in this case LR) can also be applied to LL parsers
• The construction of parse trees with Generalized LR parsers is more complex because the parser yields a forest of parse trees
• These forests may become big. The number of parses can be reduced by taking semantic values into consideration during parsing (research issue!). This reduces also the size of the parse forest.

Syntactical analysis

- Scannerless: in a traditional compiler lexical syntax is implemented by a scanner and context-free syntax by a parser. SGLR: scanner and parser are integrated
  - makes resulting parser more expressive
  - simplifies the implementation
  - disadvantages:
    - more stack operations for every character
    - explosion of parse table in case of Unicode

- Scanners have problems with common programming languages:
  - Have implicit lexical disambiguation rules (longest match, keywords first, ...)
  - Parse context independent decisions, or...
  - Complex interfaces with a parser
  - Example in PL1:
    IF THEN THEN THEN = ELSE;
    ELSE ELSE ELSE = THEN;
Syntactical analysis

• Scannerless parsing:
  • No implicit lexical disambiguation rules
  • Context-free power for lexical syntax
  • No interfacing between scanner and parser
• GLR gives arbitrary look ahead
• Design solutions for frequently occurring ambiguities

Syntactical analysis

• Longest match
  • Keywords
  • Priority conflicts
  • Associativity
  • Choice between productions

• Follow restrictions
  • Reject productions
  • Priority declarations
  • Associativity attributes

Syntactical analysis

• Follow restrictions (longest match)
  Syntax:
  \[ \text{Id}^+ \rightarrow \text{List} \]
  \[ [a-z]^+ \rightarrow \text{Id} \]
  Input sentence:
  “foo” = [f,o,o] v[fo,o] v[f,oo] v[foo]
  Restriction:
  \[ \text{Id} \not\rightarrow [a-z] \]
  • Id can not immediately be followed by [a-z]
  • Longest match for a specific non-terminal

Syntactical analysis

• Reject productions (reserved keywords)
  Syntax:
  \[ \text{Id}^+ \rightarrow \text{List} \]
  \[ "\text{begin}" \text{Id}^* "\text{end}" \rightarrow \text{List} \]
  Input sentence:
  “begin foo end” = [begin,foo,end] v [foo]
  Reject production:
  “begin” |"end" \rightarrow \text{Id} (reject)
  • “begin” and “end” can never be derived as Id
Syntactical analysis

- Implementation:
  - Generalized: basically Lang's & Tomita's GLR
    - stack forks when there is a conflict
    - stacks merge as soon as possible
  - Scannerless: each character is a token
  - Disambiguation: Filtering

  
  Grammar → Generator → Table → Parser → Forest → Tree Filters → Tree

  - Rule of thumb: the sooner a filter is applied the faster the parser is

Syntactical analysis

- Implementing filters:

  Follow restrictions (1) Reject productions Preference attributes
  Priority rules Follow restrictions (>1)
  Associativity attributes

Syntactical analysis

- Implementation of follow restrictions

  (1) character restrictions:
  - Shifts of characters that are in the restriction set are removed from the parse table
  - (n)-character restrictions:
  - Reductions are not performed during parsing
  - Intersect the next n characters of the input with the characters in the follow restriction

Syntactical analysis

- Implementation: reject productions

  Grammar → Generator → Table → Parser → Forest → Tree Filters → Tree

  - Recognize reject productions when parsing
  - Wait until all stacks have been merged
  - Stacks with more derivations are ambiguity clusters
  - If one derivation has a rejected production on top the entire cluster is dead
  - Because "begin" may not be recognized in any way as an Id
Syntactical analysis

• Priority and associativity
  Id -> Exp
  Exp “+” Exp -> Exp {left}
  Exp “*” Exp -> Exp {left}
  "a + b * c" = (a+b)*c v a+(b*c)
  "a + b + c" = (a+b)+c v a+(b+c)

  %priorities
  "*" > "+
  (left) forbids "+" to have itself as a right child
  >" forbids "*" to have "+" as a child

• Priority/associativity are filtered during parse table construction
  • productions are not predicted if conflicting
  • transitions are not added if conflicting
  • Priority/associativity ignore chain rules
  • filtering the parse forest

Syntactical analysis

• Preference attributes
  Id -> Exp
  "if" Exp "then" Exp -> Exp {prefer}
  "if" Exp "then" Exp "else" Exp -> Exp

  "if a then if b then c else d" =
  "if a then (if b then c else d)" v
  "if a then (if b then c) else d

  {prefer} selects a derivation among alternatives
  {avoid} is the dual of {prefer}

• Filtering of preferences the parse forest
• Ambiguous derivations are collected locally in the parse forest
• Compare the top productions of each derivation
• Preferred derivations are selected
• Avoided derivations are removed