Generating Incremental Parsers

Christoph Höger – TU-Berlin

PaGe

- *„An LR(0) item (item for short) of a grammar G is a production of G with a dot at some position of the body.“*
  [Compilers – Principles, Techniques & Tools]

- This may not be the best method to teach LR
- PaGe (**Pa**rser **Ge**nerator) uses „Actions“:

$$A \rightarrow w \Rightarrow A \rightarrow w \,①$$

PaGe

- PaGe transforms the input Grammar

- 1st Target: „Normalform“:

$$A \rightarrow \underline{t}\, Z_m$$

$$\ldots$$

(L(G) is kept unchanged)

$$Z_n \rightarrow ①$$

PaGe

- 2nd Target: Erase original productions:

$$A \rightarrow \underline{t}\, Z_m \quad \ldots \quad B \rightarrow A\, Z_k$$

$$A \rightarrow \underline{t}\, Z_m \quad \ldots \quad B \rightarrow \underline{A}\, Z_k \quad B \rightarrow \underline{t}\, Z_m Z_k$$

- Note the Pseudo Terminal $\underline{A}$
- A production can be deleted!

PaGe

- 3rd Target: Remove Left recursion, leftfactor common start symbols
- L(G) has not been changed yet!

$$Z_n \rightarrow \underline{A}\, Z_k$$
$$Z_n \rightarrow \underline{t}\, Z_m Z_k$$
$$Z_n \rightarrow ①$$

All rules start with (Pseudo-) Terminals (*shift*) or Actions (*reduce*)

- This implies a LR Parse table

Incremental Parsing - What?

- Incremental Parser
  - Input: An AST and a diff describing the changed input tokens
  - Output: An AST which reuses as much Nodes as possible from the input AST
- Works well for most real life grammars, but yields no general advantage over batch parsers!

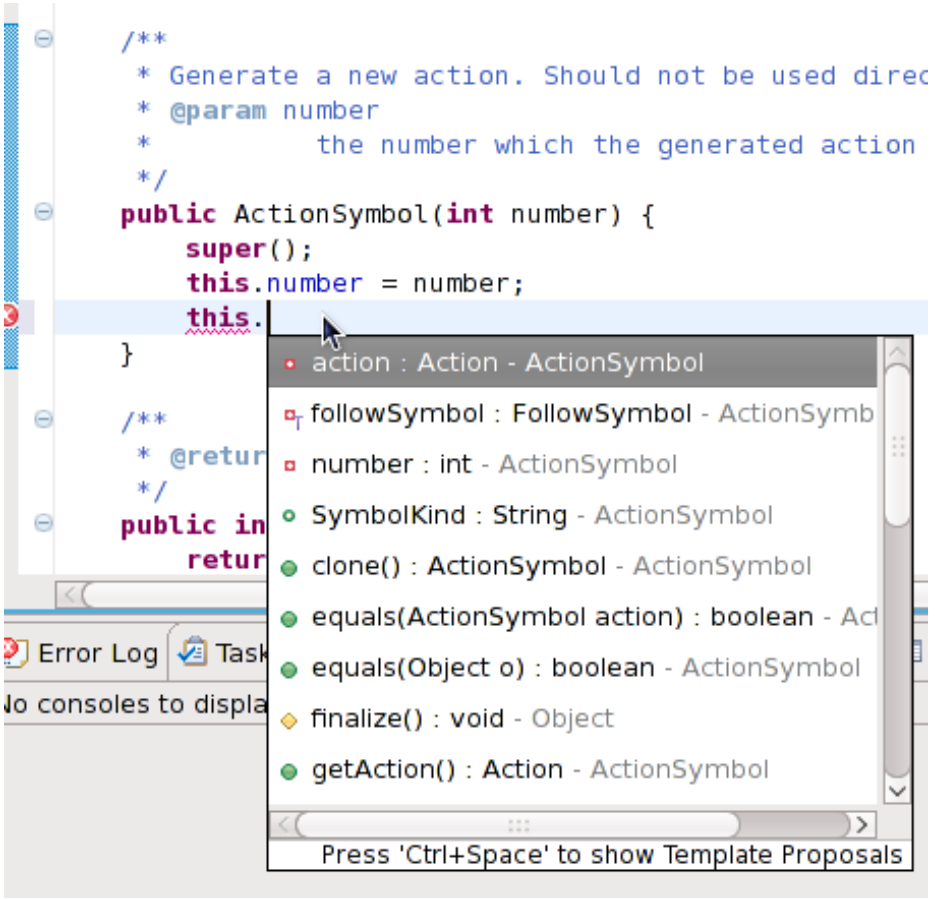
Incremental Parsing - Why?

- Why not use it for `make`?
  - Loading the last AST from disk will probably take longer than parsing from scratch.
  - Creating the diff is non-trivial

- So, why should one write an incremental parser?

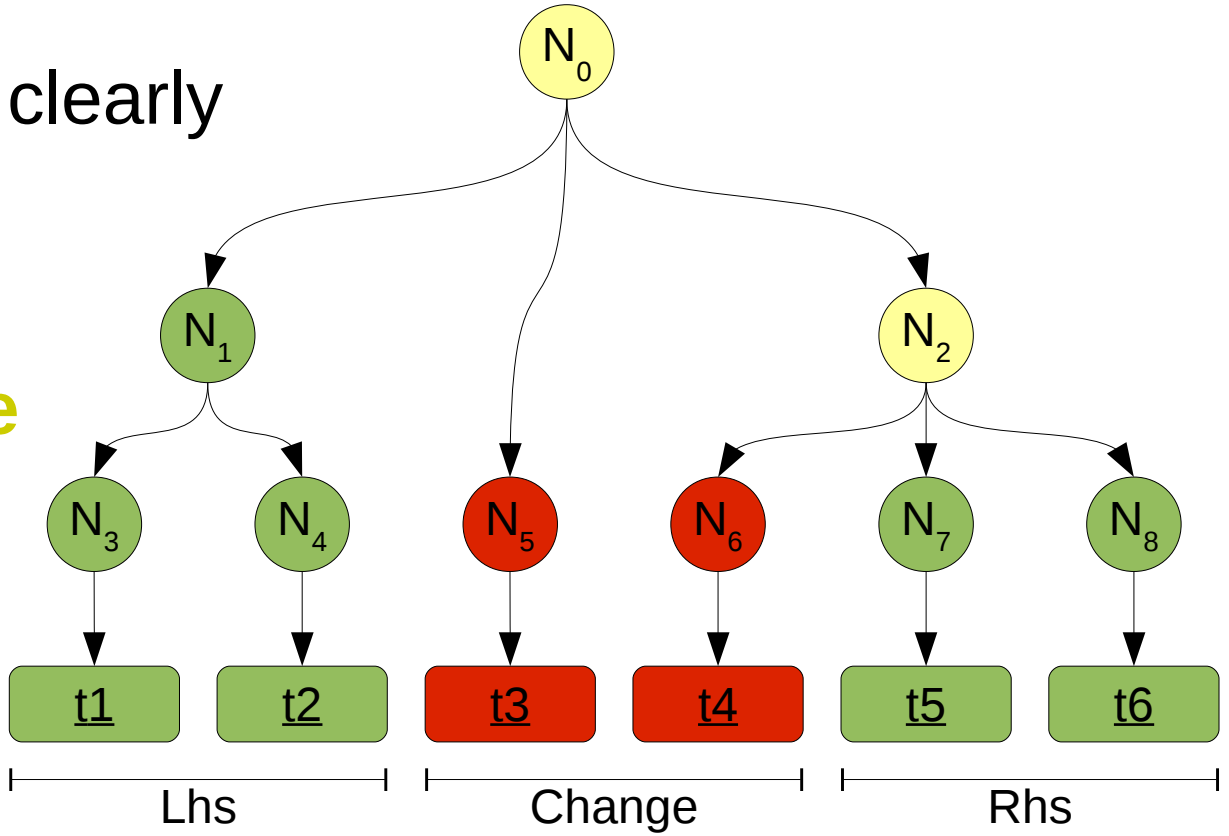
Incremental Parsing - Why?

- Programmers need help!
  - This needs an AST to walk on.
  - User expects no delay.
  - Parsing needs to be in „realtime“



Incremental Parsing - How?

- Tree input:
  - Some nodes clearly **reusable**
  - Some **not**
  - Some **maybe**

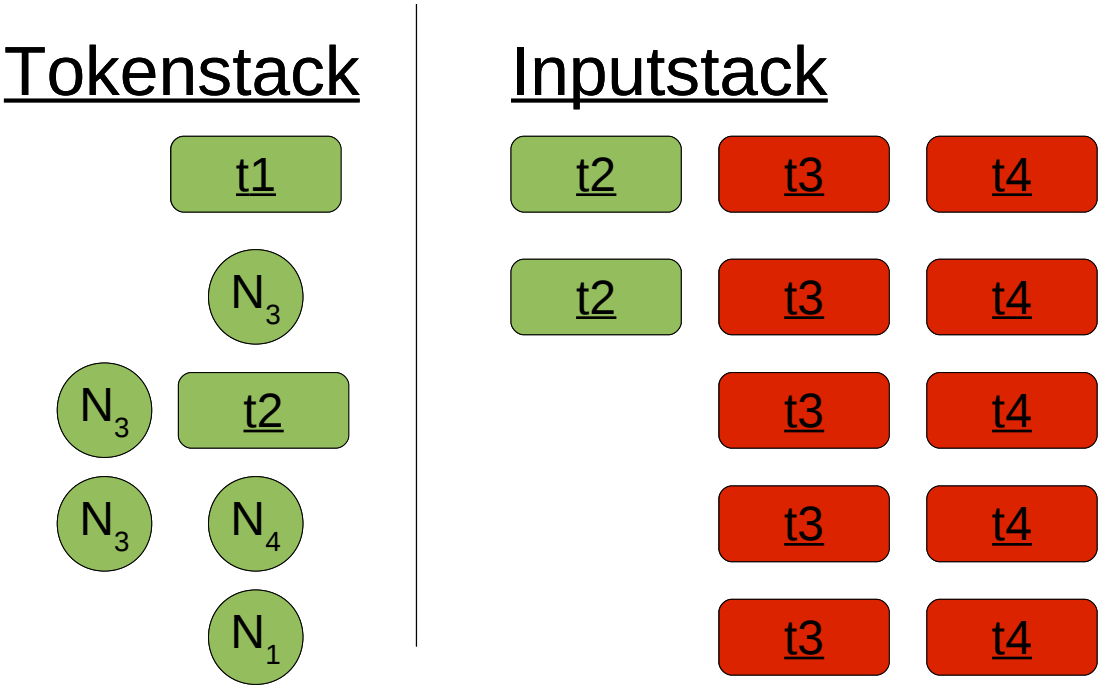


Incremental Parsing - How?

- We will not simply save Parser configurations (although that would be trivial)
- Basic idea: „Short circuiting the parser“
  - We know what the parser did for a given input
- Two Phase design.
  - First phase handle left hand side AST
  - Second phase: handle change spot and right hand side

Incremental Parsing - How?

- Batch parsing:



Phase 1

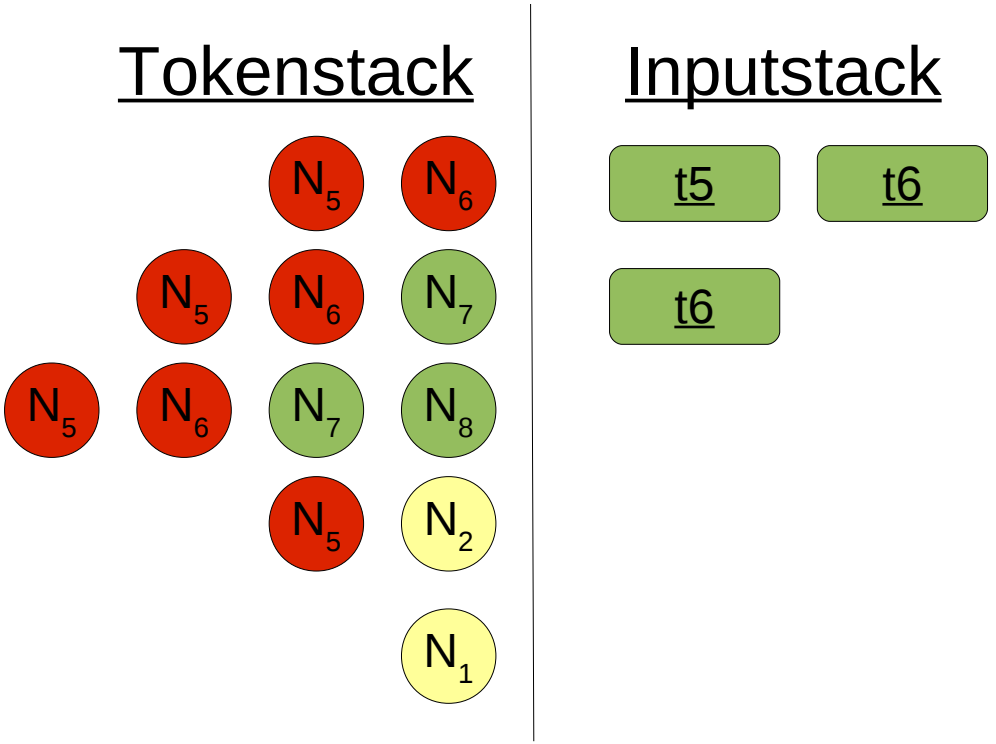
- We can directly calculate the final configuration by looking at N_1, because:
  - It has no changed children
  - The parser is deterministic
  - It is a valid reduction in the current state
- Phase 1 is:
  - Descend the tree
  - If the Node is unchanged, simulate its reduction
  - else test check its children

Phase 2

- Phase 2 is a little bit trickier:
  - Use parent references (new terminals have none)
  - Check for the highest parent node that can be reused
  - (The parse table allows us to check this)
  - This allows reusage of nodes even in new subtrees (!)
  - Use the batch parse steps when needed

Phase 2

- Phase 2:



Runtime

- We assume a balanced AST with n nodes
- Runtime O(|change| +log²(n)) :
  - Phase 1 descends the tree: O(log(n))
  - The change is parsed in linear time O(|change|)
  - Phase 2 ascends at least one step after checking O(log(n)) parent nodes: O(log²(n))
- Space: None (except parent links, but you'll need them anyway)

Optimizing Phase 2

- $O(\log^2(n))$ can be dropped to $O(\log(n))$:
  - After reusage directly check for sibling reusage
  - If no sibling, check parents sibling
  - This eliminates unnecessary ascensions
  - Needs some kind of sibling references
  - Sounds cool, but in fact can lead to performance regressions (worst case tree is very unlikely)

Object Reusage

- Remember the yellow nodes?
  - Why recreate them?
  - We construct bottom up: If one child node already has a parent, reuse it!
  - This would take O(k) time, where k is the maximum amount of child nodes.
  - In PaGe we use Lists, k is not constant
  - But: Only the first and last child need to be checked, since there is only one change

Conclusion

- Incremental Parsers are valuable in IDEs
- For changes with constant length, one can reconcile the AST in O(log(n)) and with no additional space cost
- This is practical realtime for normal source files (< 64kb)