3. Context-free Grammars and Syntactic Analysis

Input: **token sequence**

Tasks:

Parsing: construct a derivation according to the **concrete syntax**,
Tree construction: build a structure tree according to the **abstract syntax**,
Error handling: detection of an error, message, recovery

Result: **abstract program tree**

Compiler module parser:

deterministic stack automaton, augmented by actions for tree construction

top-down parsers: leftmost derivation; tree construction top-down or bottom-up

bottom-up parsers: rightmost derivation backwards; tree construction bottom-up

Abstract program tree (condensed derivation tree):
represented by a

- **data structure in memory** for the translation phase to operate on,
- linear **sequence of nodes on a file** (costly in runtime),
- **sequence of calls** of functions of the translation phase.

Lecture Programming Languages and Compilers WS 2013/14 / Slide 301

Objectives:

Relation between parsing and tree construction

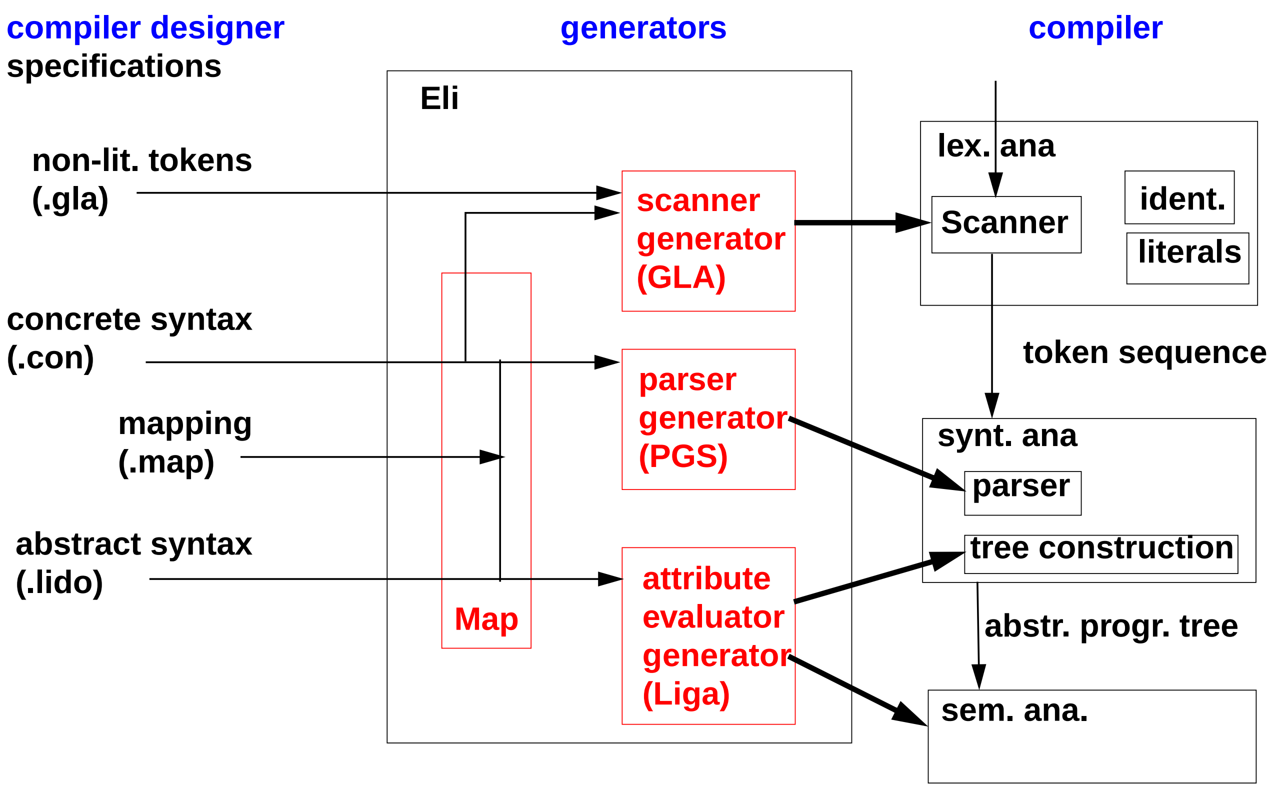
In the lecture:

- Explain the tasks, use example on PLaC-1.3.
- Sources of prerequisites:
- context-free grammars: "Grundlagen der Programmiersprachen (2nd Semester), or "Berechenbarkeit und formale Sprachen" (3rd Semester),
- Tree representation in prefix form, postfix form: "Modellierung" (1st Semester).

Suggested reading:

Kastens / Übersetzerbau, Section 4.1

Generating the structuring phase from specifications (Eli)



Lecture Programming Languages and Compilers WS 2013/14 / Slide 301a

Objectives:

Understand how generators build the structuring phase

In the lecture:

Explain

- the flow of information from the specifications to the generators,
- the generated products in the compiler.

Suggested reading:

Kastens / Übersetzerbau, Section 4.1

3.1 Concrete and abstract syntax

concrete syntax	**abstract syntax**
- context-free grammar	- context-free grammar
- defines the structure of source programs	- defines abstract program trees
- is unambiguous	- is usually ambiguous
- specifies derivation and parser	- translation phase is based on it
- parser actions specify the tree construction --->	- tree construction
- some chain productions have only syntactic purpose `Expr ::= Fact` have no action	no node created
- symbols are mapped `{Expr,Fact}` ->	to one abstract symbol `Exp`

- same action at structural equivalent productions:
`Expr ::= Expr AddOpr Fact &BinEx`
`Fact ::= Fact MulOpr Opd  &BinEx`
	- creates tree nodes

- semantically relevant chain productions, e.g. `ParameterDecl ::= Declaration`	- are kept (tree node is created)
- terminal symbols identifiers, literals, keywords, special symbols	- only semantically relevant ones are kept identifiers, literals
- concrete syntax and symbol mapping specify	- abstract syntax (can be generated)

Lecture Programming Languages and Compilers WS 2013/14 / Slide 302

Objectives:

Distinguish roles and properties of concrete and abstract syntax

In the lecture:

- Use the expression grammar of PLaC-3.3, PLaC-3.4 for comparison.
- Construct abstract syntax systematically.
- Context-free grammars specify trees - not only strings! Is also used in software engineering to specify interfaces.

Suggested reading:

Kastens / Übersetzerbau, Section 4.1

Assignments:

- Generate abstract syntaxes from concrete syntaxes and symbol classes.
- Use Eli for that task. Exercise 10

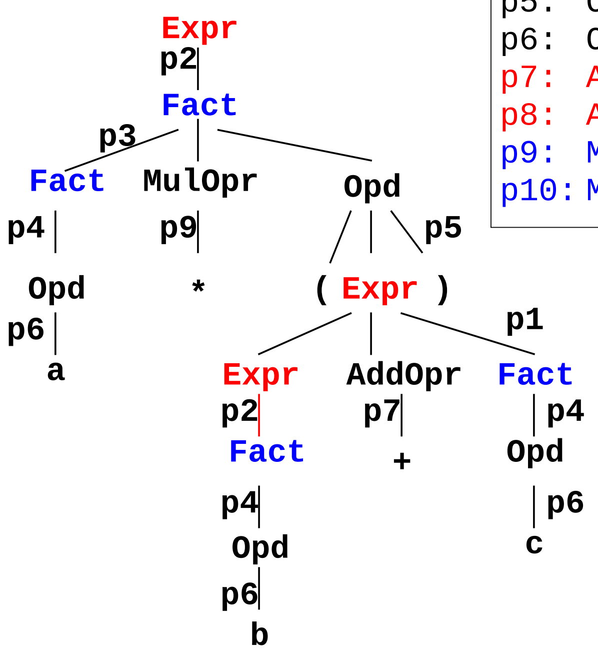
Questions:

- Why is no information lost, when an expression is represented by an abstract program tree?
- Give examples for semantically irrelevant chain productions outside of expressions.
- Explain: XML-based languages are defined by context-free grammars. Their sentences are textual representations of trees.

Example: concrete expression grammar

name	production	action
p1:	`Expr ::= Expr AddOpr Fact`	*BinEx*
p2:	`Expr ::= Fact`	
p3:	`Fact ::= Fact MulOpr Opd`	*BinEx*
p4:	`Fact ::= Opd`	
p5:	`Opd ::= '(' Expr ')'`	
p6:	`Opd ::= Ident`	*IdEx*
p7:	`AddOpr ::= '+'`	*PlusOpr*
p8:	`AddOpr ::= '-'`	*MinusOpr*
p9:	`MulOpr ::= '*'`	*TimesOpr*
p10:	`MulOpr ::= '/'`	*DivOpr*

derivation tree for `a * (b + c)`



`+, - lower precedence`
`*, / higher precedence`

Lecture Programming Languages and Compilers WS 2013/14 / Slide 303

Objectives:

Illustrate comparison of concrete and abstract syntax

In the lecture:

- Repeat concepts of "GdP" (slide GdP-2.5): Grammar expresses operator precedences and associativity.
- The derivation tree is constructed by the parser - not necessarily stored as a data structure.
- Chain productions have only one non-terminal symbol on their right-hand side.

Suggested reading:

Kastens / Übersetzerbau, Section 4.1

Suggested reading:

slide GdP-2.5

Questions:

- How does a grammar express operator precedences and associativity?
- What is the purpose of the chain productions in this example.
- What other purposes can chain productions serve?

Patterns for expression grammars

Expression grammars are **systematically** constructed,
such that **structural properties** of expressions are defined:

one level of precedence, **binary** operator, left-associative:

```
A ::= A Opr B
A ::= B
```

one level of precedence, **binary** operator, right-associative:

```
A ::= B Opr A
A ::= B
```

one level of precedence, **unary Operator**, prefix:

```
A ::= Opr A
A ::= B
```

one level of precedence, **unary Operator**, postfix:

```
A ::= A Opr
A ::= B
```

Elementary operands: **only derived** from the nonterminal of the **highest precedence** level (be H here):

```
H ::= Ident
```

Expressions in parentheses: only derived from the nonterminal of the **highest precedence** level (assumed to be H here); **contain** the nonterminal of the **lowest precedence level** (be A here):

```
H ::= '(' A ')'
```


Lecture Programming Languages and Compilers WS 2013/14 / Slide 303a

Objectives:

Be able to apply the patterns

In the lecture:

Explain the patterns

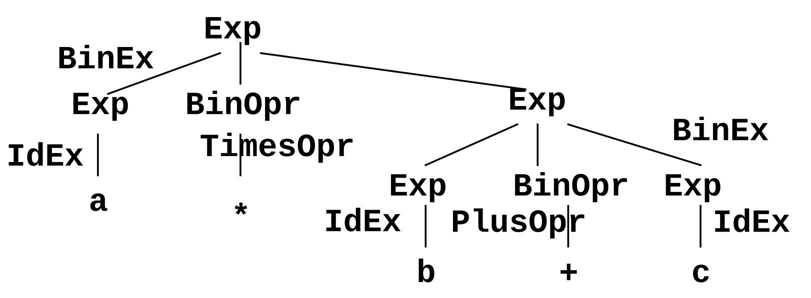
Assignments:

Apply the patterns to understand given and construct new expression grammars.

Example: abstract expression grammar

name	production
`BinEx:`	`Exp ::= Exp BinOpr Exp`
`IdEx:`	`Exp ::= Ident`
`PlusOpr:`	`BinOpr ::= '+'`
`MinusOpr:`	`BinOpr ::= '-'`
`TimesOpr:`	`BinOpr ::= '*'`
`DivOpr:`	`BinOpr ::= '/'`

abstract program tree for `a * (b + c)`



symbol classes:

```
Exp = { Expr, Fact, Opd }
BinOpr = { AddOpr, MulOpr }
```

Actions of the concrete syntax: **productions** of the abstract syntax to create tree nodes for **no action** at a concrete chain production: **no tree node** is created

Lecture Programming Languages and Compilers WS 2013/14 / Slide 304

Objectives:

Illustrate comparison of concrete and abstract syntax

In the lecture:

- Repeat concepts of "GdP" (slide GdP-2.9):
- Compare grammars and trees.
- Actions create nodes of the abstract program tree.
- Symbol classes shrink node pairs that represent chain productions into one node

Suggested reading:

Kastens / Übersetzerbau, Section 4.1

Suggested reading:

slide GdP-2.9

Questions:

- Is this abstract grammar unambiguous?
- Why is that irrelevant?

3.2 Design of concrete grammars

Objectives

The concrete grammar for **parsing**

- is parsable: fulfills the **grammar condition** of the chosen parser generator;
- specifies the **intended language -** or a small super set of it;
- is provably related to the **documented grammar**;
- can be **mapped to** a suitable **abstract grammar**.

Lecture Programming Languages and Compilers WS 2013/14 / Slide 304a

Objectives:

Guiding objectives

In the lecture:

The objectives are explained.

A strategy for grammar development

1. **Examples**: Write at least one example for every intended language construct. Keep the examples for checking the grammar and the parser.
2. **Sub-grammars**: Decompose a non-trivial task into topics covered by sub-gammars, e.g. statements, declarations, expressions, over-all structure.
3. **Top-down**: Begin with the start symbol of the (sub-)grammar, and refine each nonterminal according to steps 4 - 7 until all nonterminals of the (sub-)grammar are refined.
4. **Alternatives**: Check whether the language construct represented by the current nonterminal, say Statement, shall occur in structurally different alternatives, e.g. while-statement, if-statement, assignment. Either introduce chain productions, like
   `Statement ::= WhileStatement | IfStatement | Assignment.`
   or apply steps 5 - 7 for each alternative separately.
5. **Consists of**: For each (alternative of a) nonterminal representing a language construct explain its immediate structure in words, e.g. „A Block is a declaration sequence followed by a statement sequence, both enclosed in curly braces." Refine only one structural level. Translate the description into a production. If a sub-structure is not yet specified introduce a new nonterminal with a speaking name for it, e.g.
   `Block ::= ’{’ DeclarationSeq StatementSeq ’}’.`
6. **Natural structure**: Make sure that step 5 yields a „natural" structure, which supports notions used for static or dynamic semantics, e.g. a range for valid bindings.
7. **Useful patterns**: In step 5 apply patterns for description of sequences, expressions, etc.

Lecture Programming Languages and Compilers WS 2013/14 / Slide 304aa

Objectives:

Develop CFGs systematically

In the lecture:

- Apply the strategy for a little task.
- Apply the strategy in context of the running project.
- Apply the patterns of slides GPS-2.10, GPS-2.10, 12, 14, 15.
- The strategy is applicable for the concrete and the abstract syntax.

Suggested reading:

Kastens / Übersetzerbau, Section 4.1

Suggested reading:

slide GdP-2.10ff

Grammar design for an existing language

- Take the grammar of the **language specification literally**.
- Only **conservative modifications** for parsability or for mapping to abstract syntax.
- **Describe all modifications**.
(see ANSI C Specification in the Eli system description
http://www.uni-paderborn.de/fachbereich/AG/agkastens/eli/examples/eli_cE.html)

- **Java** language specification (1996):
Specification grammar is not LALR(1).
5 problems are described and how to solve them.
- **Ada** language specification (1983):
Specification grammar is LALR(1)
- requirement of the language competition
- **ANSI C, C++**:
several ambiguities and LALR(1) conflicts, e.g.
„**dangling else**“,
„**typedef problem**“:
`A (*B);`
is a declaration of variable **`B`**, if **`A`** is a type name,
otherwise it is a call of function **`A`**

Lecture Programming Languages and Compilers WS 2013/14 / Slide 304b

Objectives:

Avoid document modifications

In the lecture:

- Explain the conservative strategy.
- Java gives a solution for the dangling else problem.
- For typedef problem see PLaC-2.3.

Grammar design together with language design

Read grammars before writing a new grammar.

Apply **grammar patterns systematically** (cf. GPS-2.5, GPS-2.8)

- repetitions
- optional constructs
- precedence, associativity of operators

Syntactic structure should reflect semantic structure:

E. g. a range in the sense of scope rules should be represented by a single subtree of the derivation tree (of the abstract tree).

Violated in Pascal:

functionDeclaration ::= functionHeading block
functionHeading ::= ‘function‘ identifier formalParameters ‘:‘ resultType ‘;‘

formalParameters together with block form a range,
but identifier does not belong to it

Lecture Programming Languages and Compilers WS 2013/14 / Slide 304c

Objectives:

Grammar design rules

In the lecture:

- Refer to GdP slides.
- Explain semantic structure.
- Show violation of the example.

Syntactic restrictions versus semantic conditions

Express a restriction **syntactically** only if
it can be **completely covered with reasonable complexity**:

- **Restriction can not be decided syntactically**:
  e.g. type check in expressions:
  BoolExpression ::= IntExpression ‘<‘ IntExpression

- **Restriction can not always be decided syntactically**:
  e. g. disallow array type to be used as function result
  Type ::= ArrayType | NonArrayType | Identifier
  ResultType ::= NonArrayType
  If a type identifier may specify an array type,
  a semantic condition is needed, anyhow

- **Syntactic restriction is unreasonably complex**:
  e. g. distinction of compile-time expressions from ordinary expressions requires duplication of the expression syntax.

Lecture Programming Languages and Compilers WS 2013/14 / Slide 304d

Objectives:

How to express restrictions

In the lecture:

- Examples are explained.
- Semantic conditions are formulated with attribute grammar concepts, see next chapter.

Assignments:

Discuss further examples for restrictions.

Eliminate ambiguities

unite syntactic constructs - distinguish them semantically

Examples:

- Java:
```
ClassOrInterfaceType  ::= ClassType | InterfaceType
InterfaceType         ::= TypeName
ClassType             ::= TypeName
```
  replace first production by
```
ClassOrInterfaceType  ::= TypeName
```
  semantic analysis distinguishes between class type and interface type

- Pascal:
```
factor              ::= variable | ... | functionDesignator
variable            ::= entireVariable | ...
entireVariable      ::= variableIdentifier
variableIdentifier  ::= identifier                               (**)
functionDesignator  ::= functionIdentifier                        (*)
                     |  functionIdentifer '(' actualParameters ')'
functionIdentifier  ::= identifier
```
  eliminate marked (*) alternative
  semantic analysis checks whether (**) is a function identifier

Lecture Programming Languages and Compilers WS 2013/14 / Slide 304e

Objectives:

Typical ambiguities

In the lecture:

- Same notation with different meanings;
- ambiguous, if they occur in the same context.
- Conflicting notations may be separated by several levels of productions (Pascal example)

Questions:

Unbounded lookahead

The decision for a **reduction** is determined by a **distinguishing token** that may be **arbitrarily far to the right**:

Example, **forward** declarations as could have been defined in Pascal:

functionDeclaration ::=
‘function‘ forwardIdent formalParameters ‘:‘ resultType ‘;‘ ‘forward‘

| ‘function‘ functionIdent formalParameters ‘:‘ resultType ‘;‘ block

The distinction between **forwardIdent** and **functionIdent** would require to see the **forward** or the **begin** token.

Replace **forwardIdent** and **functionIdent** by the same nonterminal; distinguish semantically.

Lecture Programming Languages and Compilers WS 2013/14 / Slide 304f

Objectives:

Typical situation

In the lecture:

Explain the problem and the solution using the example

Questions:

3.3 Recursive descent parser

top-down (construction of the **derivation** tree), **predictive** method

Systematic transformation of a context-free grammar into a set of functions:

non-terminal symbol X	function X
alternative productions for X	branches in the function body
decision set of production p_i	decision for branch p_i
non-terminal occurrence X ::= ... Y ...	function call Y()
terminal occurrence X ::= ... t ...	accept a token t and read the next token

Productions for `Stmt`:

```
p1: Stmt ::=
     Variable ':=' Expr

p2: Stmt ::=
     'while' Expr 'do' Stmt
```

```
void Stmt ()
{  switch (CurrSymbol)
   {
      case decision set for p1:
         Variable();
         accept(assignSym);
         Expr();
         break;
      case decision set for p2:
         accept(whileSym);
         Expr();
         accept(doSym);
         Stmt();
         break;
   default: Fehlerbehandlung();
}  }
```


Lecture Programming Languages and Compilers WS 2013/14 / Slide 305

Objectives:

Understand the construction schema

In the lecture:

Explanation of the method:

- Demonstrate the construction of a left-derivation and the top-down construction of a derivation tree by this animation.
- Relate grammar constructs to function constructs.
- Each function plays the role of an acceptor for a symbol.
- accept function for reading and checking of the next token (scanner).
- Computation of decision sets on PLaC-3.6.
- Decision sets must be pairwise disjoint!

Suggested reading:

Kastens / Übersetzerbau, Section 4.2

Questions:

- A parser algorithm is based on a stack automaton. Where is the stack of a recursive descent parser? What corresponds to the states of the stack automaton?
- Where can actions be inserted into the functions to output production sequences in postfix or in prefix form?

Grammar conditions for recursive descent

Definition: A context-free grammar is **strong LL(1)**, if for any pair of **productions** that have the **same symbol on their left-hand sides**, A ::= u and A ::= v, the **decision sets are disjoint**:

DecisionSet (A ::= u) $\cap$ DecisionSet (A ::= v) = $\emptyset$

with

DecisionSet (A ::= u) := if nullable (u) then **First (u)** $\cup$ **Follow (A)** else **First (u)**

nullable (u) holds iff a derivation u $\Rightarrow^*$ ε exists

First (u) := { t $\in$ T | v $\in$ V* exists and a derivation u $\Rightarrow^*$ t v }

Follow (A):= { t $\in$ T | u,v $\in$ V* exist, A $\in$ N and a derivation S $\Rightarrow^*$ u A t v }

Example:

		production	DecisionSet
p1:	Prog	::= Block #	**begin**
p2:	Block	::= **begin** Decls Stmts **end**	**begin**
p3:	Decls	::= Decl **;** Decls	**new**
p4:	Decls	::=	Ident **begin**
p5:	Decl	::= **new** Ident	**new**
p6:	Stmts	::= Stmts **;** Stmt	**begin** Ident
p7:	Stmts	::= Stmt	**begin** Ident
p8:	Stmt	::= Block	**begin**
p9:	Stmt	::= Ident **:=** Ident	Ident

non-terminal

X	First (X)	Follow (X)
Prog	**begin**	
Block	**begin**	**# ; end**
Decls	**new**	Ident **begin**
Decl	**new**	**;**
Stmts	**begin** Ident	**; end**
Stmt	**begin** Ident	**; end**

Lecture Programming Languages and Compilers WS 2013/14 / Slide 306

Objectives:

Strong LL(1) can easily be checked

In the lecture:

- Explain the definitions using the example.
- First set: set of terminal symbols, which may begin some token sequence that is derivable from u.
- Follow set: set of terminal symbols, which may follow an A in some derivation.
- Disjoint decision sets imply that decisions can be made deterministically using the next input token.
- For k=1: Strong LL(k) is equivalent to LL(k).

Suggested reading:

Kastens / Übersetzerbau, Section 4.2, LL(k) conditions, computation of First sets and Follow sets

Questions:

The example grammar is not strong LL(1).

- Show where the condition is violated.
- Explain the reason for the violation.
- What would happen if we constructed a recursive descent parser although the condition is violated?

Computation rules for nullable, First, and Follow

Definitions:

nullable(u) holds iff a derivation $u \Rightarrow^* \varepsilon$ exists

First(u):= $\{ t \in T \mid v \in V^*$ exists and a derivation $u \Rightarrow^* t\ v \}$

Follow(A):= $\{ t \in T \mid u,v \in V^*$ exist, $A \in N$ and a derivation $S \Rightarrow^* u\ A\ v$ such that $t \in$ First(v) $\}$

with $G = (T, N, P, S)$; $V = T \cup N$; $t \in T$; $A \in N$; $u,v \in V^*$

Computation rules:

nullable(ε) = true; nullable(t) = false; nullable(uv) = nullable(u) $\wedge$ nullable(v);
nullable(A) = true iff $\exists$ A::=u $\in$ P $\wedge$ nullable(u)

First(ε) = $\emptyset$; First(t) = {t};
First(uv) = if nullable(u) then First(u) $\cup$ First(v) else First(u)
First(A) = First(u_1) $\cup \ldots \cup$ First(u_n) for all A::=$u_i \in$ P

Follow(A):
 if A=S then # $\in$ Follow(A)
 if Y::=uAv $\in$ P then First(v) $\subseteq$ Follow(A) and if nullable(v) then Follow(Y) $\subseteq$ Follow(A)

Lecture Programming Languages and Compilers WS 2013/14 / Slide 306a

Objectives:

Compute First- and Follow-sets

In the lecture:

- Explain and apply computation rules

Suggested reading:

Kastens / Übersetzerbau, Section 4.2, LL(k) conditions, computation of First sets and Follow sets

Grammar transformations for LL(1)

Consequences of strong LL(1) condition:
A strong LL(1) grammar can not have

Simple **grammar transformations** that keep the defined **language invariant**:

- **alternative productions that begin with the same symbols:**

left-factorization:

non-LL(1) productions	transformed
A ::= v u	A ::= v X
A ::= v w	X ::= u
	X ::= w

- **productions that are directly or indirectly left-recursive:**

elimination of direct recursion:

A ::= A u	A ::= v X
A ::= v	X ::= u X
	X ::=

special case empty v:

A ::= A u	A ::= u A
A ::=	A ::=

u, v, w ∈ V*
X ∈ N does not occur in the original grammar

Lecture Programming Languages and Compilers WS 2013/14 / Slide 307

Objectives:

Understand transformations and their need

In the lecture:

- Argue why strong LL(1) grammars can not have such productions.
- Show why the transformations remove those problems.
- Replacing left-recursion by right recursion would usually distort the structure.
- There are more general rules for indirect recursion.

Questions:

- Apply recursion elimination for expression grammars.

LL(1) extension for EBNF constructs

EBNF constructs can avoid violation of strong LL(1) condition:

EBNF construct:	**Option [u]**	**Repetition (u)***
Production:	A ::= v [u] w	A ::= v (u)* w

additional LL(1)-condition:

if nullable(w)
then $First(u) \cap (First(w) \cup Follow(A)) = \emptyset$
else $First(u) \cap First(w) = \emptyset$

in recursive descent parser:

Option	Repetition
v if (CurrToken in First(u)) { u } w	v while (CurrToken in First(u)) { u } w

Repetition (u)+ left as exercise

Lecture Programming Languages and Compilers WS 2013/14 / Slide 307a

Objectives:

Understand transformations and their need

In the lecture:

- Show EBNF productions in recursive descent parsers.

Questions:

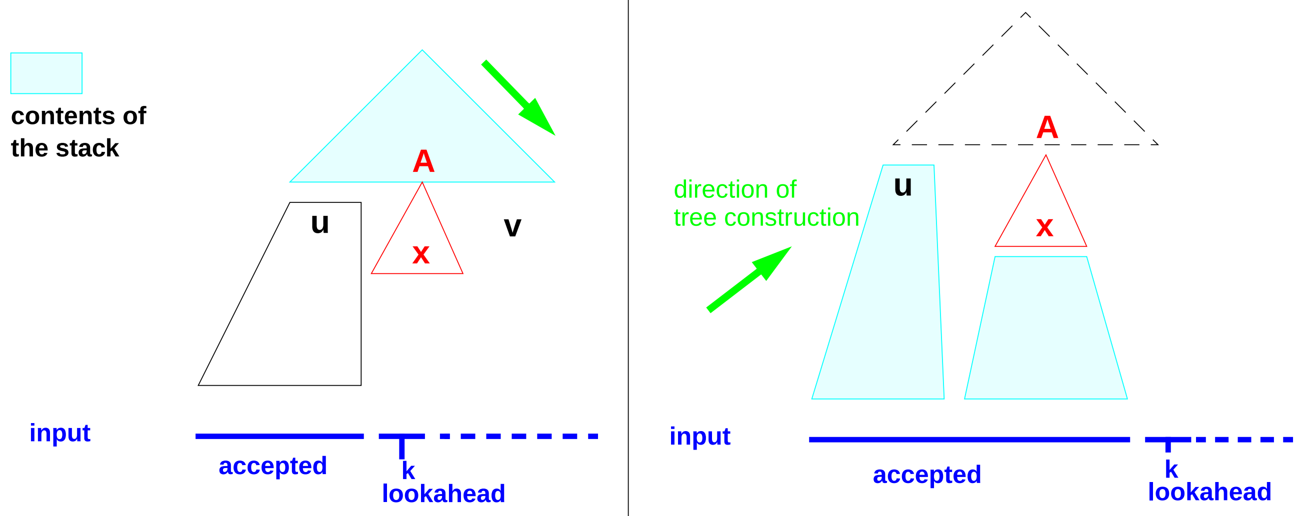
- Write a strong LL(1) expression grammar using EBNF.

Comparison: top-down vs. bottom-up

Information a stack automaton has when it decides to apply production A ::= x :

top-down, predictive
leftmost derivation

bottom-up
rightmost derivation backwards



A bottom-up parser has seen more of the input when it decides to apply a production.

Consequence: **bottom-up** parsers and their grammar classes are more **powerful**.

Lecture Programming Languages and Compilers WS 2013/14 / Slide 308

Objectives:

Understand the decision basis of the automata

In the lecture:

Explain the meaning of the graphics:

- role of the stack: contains states of the automaton,
- accepted input: will not be considered again,
- lookahead: the next k symbols, not yet accepted
- leftmost derivation: leftmost non-terminal is derived next; rightmost correspondingly,
- consequences for the direction of tree construction,

Abbreviations

- LL: (L)eft-to-right, (L)eftmost derivation,
- LR: (L)eft-to-right, (R)ightmost derivation,
- LALR: (L)ook(A)head LR

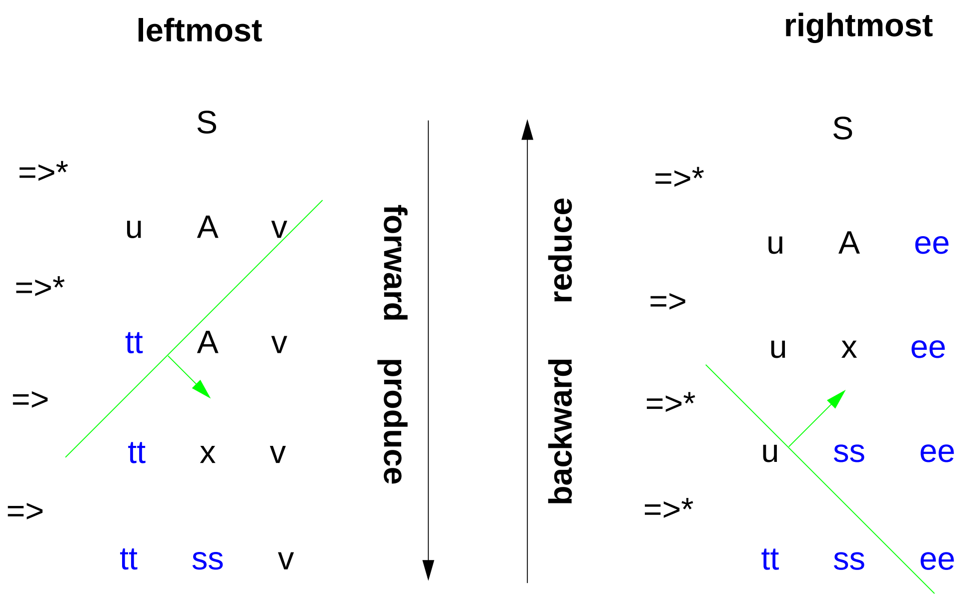
Suggested reading:

Kastens / Übersetzerbau, Section Text zu Abb. 4.2-1, 4.3-1

Questions:

Use the graphics to explain why a bottom-up parser without lookahead (k=0) is reasonable, but a top-down parser is not.

Leftmost and rightmost derivations



$u, v, x \in V^*$

$tt, ss, ee \in T^*$

$A \in N$

Lecture Programming Languages and Compilers WS 2013/14 / Slide 309

Objectives:

Understand rightmost derivation backward

In the lecture:

- Explain the two derivation patterns.

Derivation tree: top-down vs. bottom-up construction

```
p0:   P  ::= D
P1:   D  ::= FF
P2:   D  ::= FB
P3:   FF ::= 'fun' FI '(' Ps ')' 'fwd'
P4:   FB ::= 'fun' FI '(' Ps ')' B
P5:   Ps ::= Ps PI
P6:   Ps ::=
p7:   B  ::= '{' '}'
p8:   FI ::= Id
p9:   PI ::= Id
```

```
     P
p0   D
p1   FF
p3   fun FI ( Ps       ) fwd
p8       Id
p5            Ps    PI
p5            Ps PI
p6
p9               Id
p9                  Id

     fun Id (    Id Id ) fwd
```

```
P                                   p0
D                                   p1
FF                                  p3
        Ps             ) fwd        p5
                  PI                p9
        Ps        Id                p5
               PI                   p9
        Ps     Id                   p6
    FI (                            p8
fun id

fun Id (       Id Id ) fwd
```


Lecture Programming Languages and Compilers WS 2013/14 / Slide 309a

Objectives:

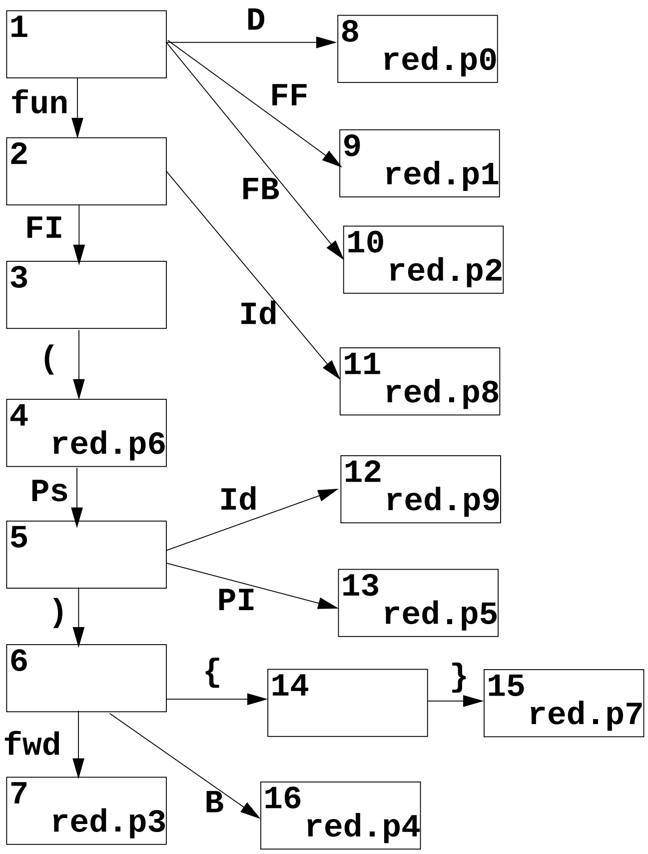
Understand derivation tree construction

In the lecture:

Use this animation to explain

- On the left: construction of a left-derivation.
- The magenta production names indicate that the decision can not be made on the base of the derivation so far and the next input tokens.
- On the right: construction of a derivation backward (bottom-up).
- No decision problem occurs.
- It is a right-derivation constructed backward.

LR(0) -Automaton



reduction	stack	input
	1	fun Id(Id Id)fwd
	1 2	Id(Id Id)fwd
p8	1 2 11	(Id Id)fwd
	1 2 3	(Id Id)fwd
p6	1 2 3 4	Id Id)fwd
	1 2 3 4 5	Id Id)fwd
p9	1 2 3 4 5 12	Id)fwd
p5	1 2 3 4 5 13	Id)fwd
	1 2 3 4 5	Id)fwd
p9	1 2 3 4 5 12	)fwd
p5	1 2 3 4 5 13	)fwd
	1 2 3 4 5	)fwd
	1 2 3 4 5 6	fwd
p3	1 2 3 4 5 6 7	#
p1	1 9	#
p0	1 8	#

Lecture Programming Languages and Compilers WS 2013/14 / Slide 309b

Objectives:

Understand understand how LR automata work

In the lecture:

- See PLaC-3.12 for explanations of the operations shift and reduce.
- Execute the automaton.

3.4 LR parsing

LR(k) grammars introduced 1965 by Donald Knuth; non-practical until subclasses were defined.

LR parsers construct the derivation tree **bottom-up,** a right-derivation backwards.

LR(k) grammar condition can not be checked directly, but
a context-free grammar is LR(k), iff the (canonical) **LR(k) automaton is deterministic**.

We consider only **1 token lookahead: LR(1).**

Comparison of LL and LR states:

The **stacks** of LR(k) and LL(k) automata **contain states**.

The construction of LR and LL states is based on the notion of **items** (see next slide).

Each **state** of an automaton represents **LL: one item** **LR: a set of items**
An LL item corresponds to a position in a case branch of a recursive function.

Lecture Programming Languages and Compilers WS 2013/14 / Slide 310

Objectives:

Introduction

In the lecture:

- Explain the comparison.

LR(1) items

An **item** represents the progress of analysis with respect to one production:

[A ::= u . v   R]   **e. g. [B ::= (. D ; S)   {#}]**

. marks the position of analysis: *accepted and reduced* . *to be accepted*

R expected **right context:**
a **set of terminals** which may follow in the input
when the complete production is accepted.
(general k>1: R contains sequences of terminals not longer than k)

Items can distinguish different right contexts: [A ::= u . v R] and [A ::= u . v R']

Reduce item:

[A ::= u v .   R]   **e. g. [B ::= (D ; S) .   {#}]**

characterizes a reduction using this production if the next input token is in R.

The automaton uses R only for the decision on reductions!

A **state** of an LR automaton represents **a set of items**

Lecture Programming Languages and Compilers WS 2013/14 / Slide 311

Objectives:

Fundamental notions of LR automata

In the lecture:

Explain

- items are also called situations,
- meaning of an item,
- lookahead in the input and right context in the automaton.
- There is no right context set in case of an LR(0) automaton.

Suggested reading:

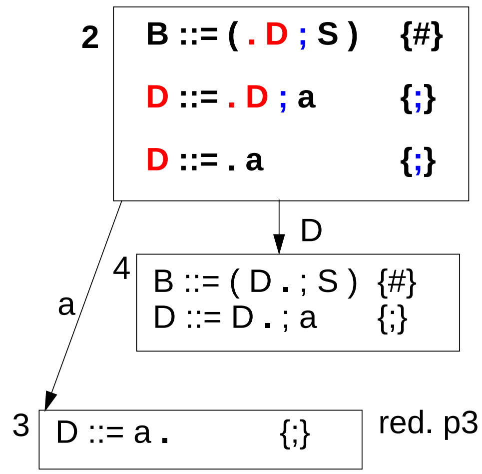
Kastens / Übersetzerbau, Section 4.3

Questions:

- What contains the right context set in case of a LR(3) automaton?

LR(1) states and operations

A **state of an LR automaton represents a set of items**
Each item represents a way in which analysis may proceed from that state.



A shift transition is made under
a **token read** from input or
a **non-terminal** symbol
obtained from a **preceding reduction.**
The state is pushed.

A **reduction** is made according to a reduce item.
n states are popped for a production of length n.

Operations:	**shift**	read and push the next state on the stack
	reduce	reduce with a certain production, pop n states from the stack
	error	error recognized, report it, recover
	stop	input accepted

Lecture Programming Languages and Compilers WS 2013/14 / Slide 312

Objectives:

Understand LR(1) states and operations

In the lecture:

Explain

- Sets of items,
- shift transitions,
- reductions.

Suggested reading:

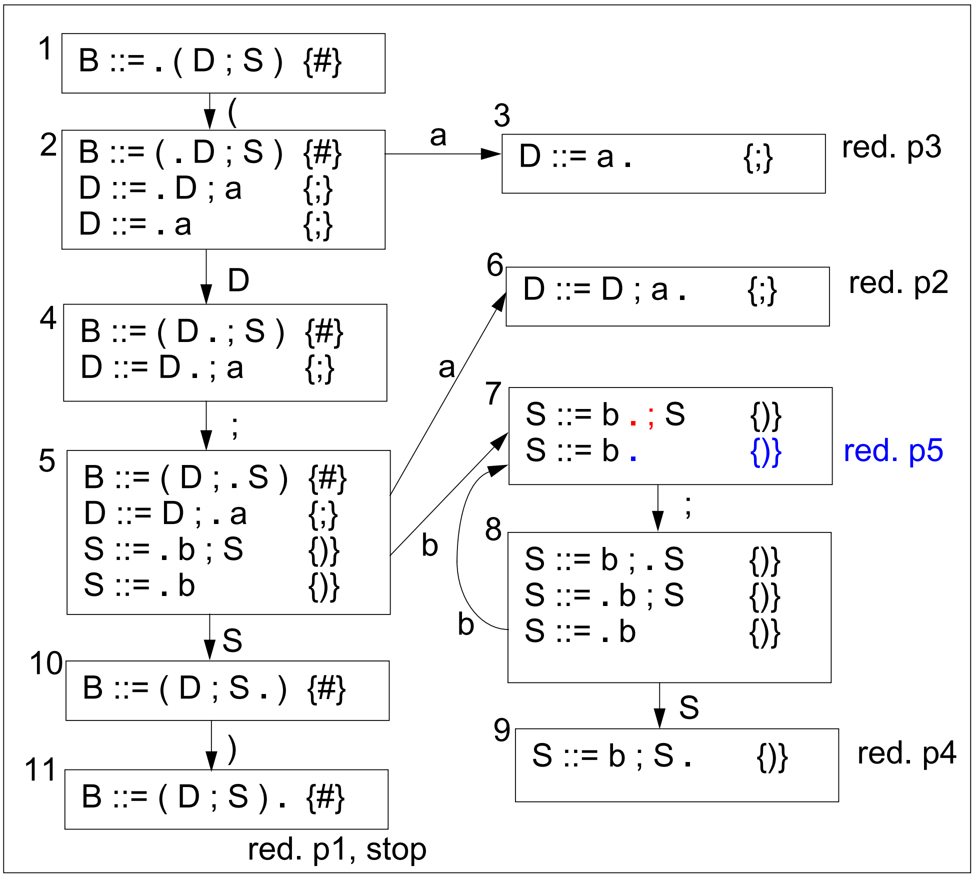
Kastens / Übersetzerbau, Section 4.3

Questions:

- Explain: A state is encoded by a number. A state represents complex information which is important for construction of the automaton.

Example for a LR(1) automaton

Grammar:
p1 B ::= (D ; S)
p2 D ::= D ; a
p3 D ::= a
p4 S ::= b ; S
p5 S ::= b



In state 7 a decision is required on next input:

- if ; then shift
- if) then reduce p5

In states 3, 6, 9, 11 a decision is not required:

- reduce on any input

Lecture Programming Languages and Compilers WS 2013/14 / Slide 313

Objectives:

Example for states, transitions, and automaton construction

In the lecture:

Use the example to explain

- the start state,
- the creation of new states,
- transitions into successor states,
- transitive closure of item set,
- push and pop of states,
- consequences of left-recursive and right-recursive productions,
- use of right context to decide upon a reduction,

erläutern.

Suggested reading:

Kastens / Übersetzerbau, Section 4.3

Questions:

- Describe the subgraphs for left-recursive and right-recursive productions. How do they differ?
- How does a LR(0) automaton decide upon reductions?

Construction of LR(1) automata

Algorithm:
1. Create the start state.
2. For each created state compute the transitive closure of its items.
3. Create transitions and successor states as long as new ones can be created.

Transitive closure is to be applied to each state q:

Consider all items in q with the analysis position before a non-terminal B:

[A_1 ::= u_1 . B v_1 R_1] ... [A_n ::= u_n . B v_n R_n],

then for each production **B ::= w**

[B ::= . w First (v_1 R_1)∪...∪First (v_n R_n)]

has to be added to state q.

before: 2
B ::= (. D ; S) {#}

after: 2
B ::= (. D ; S) {#}
D ::= . D ; a {;}∪{;}
D ::= . a {;}∪{;}

Start state:

Closure of **[S ::= . u {#}]**

S ::= u is the **unique start production**,

is an (**artificial) end symbol** (eof)

1
B ::= . (D ; S) {#}

Successor states:

For each **symbol x** (terminal or non-terminal), which occurs in some items **after the analysis position**, a **transition** is created **to a successor state**. That contains corresponding items with the **analysis position advanced behind the x** occurrence.

2
B ::= (. D ; S) {#}
D ::= . D ; a {;}
D ::= . a {;}

2 $\xrightarrow{D}$ 4
B ::= (D . ; S) {#}
D ::= D . ; a {;}

2 $\xrightarrow{a}$ 3
D ::= a . {;}

Lecture Programming Languages and Compilers WS 2013/14 / Slide 314

Objectives:

Understand the method

In the lecture:

Explain using the example on PLaC-3.13:

- transitive closure,
- computation of the right context sets,
- relation between the items of a state and those of one of its successor

Suggested reading:

Kastens / Übersetzerbau, Section 4.3

Questions:

- Explain the role of the right context.
- Explain its computation.

Operations of LR(1) automata

shift x (terminal or non-terminal):
from current state q
under x into the **successor state q‘** ,
push q‘

reduce p:
apply production p B ::= u ,
pop as many states,
as there are **symbols in u**, from the
new current state make a **shift with B**

error:
the current state has no transition
under the next input token,
issue a **message** and **recover**

stop:
reduce start production,
see # in the input

Example:

stack	input	reduction
1	(a ; a ; b ; b) #	
1 2	a ; a ; b ; b) #	
1 2 3	; a ; b ; b) #	p3
1 2	; a ; b ; b) #	
1 2 4	; a ; b ; b) #	
1 2 4 5	a ; b ; b) #	
1 2 4 5 6	; b ; b) #	p2
1 2	; b ; b) #	
1 2 4	; b ; b) #	
1 2 4 5	b ; b) #	
1 2 4 5 7	; b) #	
1 2 4 5 7 8	b) #	
1 2 4 5 7 8 7	) #	p5
1 2 4 5 7 8	) #	
1 2 4 5 7 8 9	) #	p4
1 2 4 5	) #	
1 2 4 5 10	) #	
1 2 3 5 10 11	#	p1
1	#	

Lecture Programming Languages and Compilers WS 2013/14 / Slide 315

Objectives:

Understand how the automaton works

In the lecture:

Explain operations

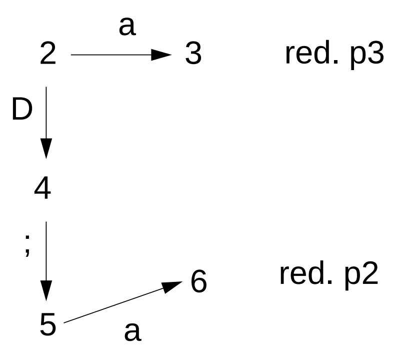
Questions:

- Why does the automaton behave differently on a-sequences than on b-sequences?
- Which behaviour is better?

Left recursion versus right recursion

left recursive productions:

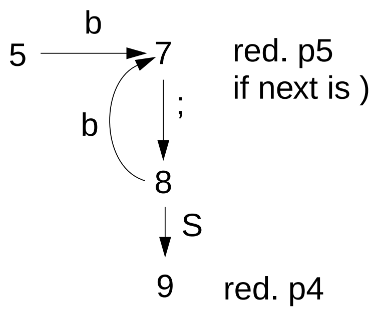
p2: D ::= D ; a
p3: D ::= a



reduction immediately after each **; a** is accepted

right recursive productions:

p4: S ::= b ; S
p5: S ::= b



the states for all **; b** are pushed before the first reduction

Lecture Programming Languages and Compilers WS 2013/14 / Slide 316

Objectives:

Understand the difference

In the lecture:

Explain

- why right recursion fills the stack deeply,
- why left recursion is advantagous.

LR conflicts

An **LR(1) automaton that has conflicts is not deterministic**.
Its **grammar is not LR(1)**;
correspondingly defined for any other LR class.

2 kinds of conflicts:

reduce-reduce conflict:
A state contains two reduce items, the **right context sets** of which are **not disjoint**:

```
...
A ::= u .   R1
B ::= v .   R2
...
```

R1, R2 not disjoint

shift-reduce conflict:
A state contains a **shift item** with the **analysis position in front of a t** and a **reduce item with t in its right context set**.

```
...
A ::= u .t v   R1
B ::= w .      R2
...
```

t $\in$ **R2**

Lecture Programming Languages and Compilers WS 2013/14 / Slide 317

Objectives:

Understand LR conflicts

In the lecture:

Explain: In certain situations the given input token t can not determine

- Reduce-reduce: which reduction is to be taken;
- Shift-reduce: whether the next token is to be shifted, a reduction is to be made.

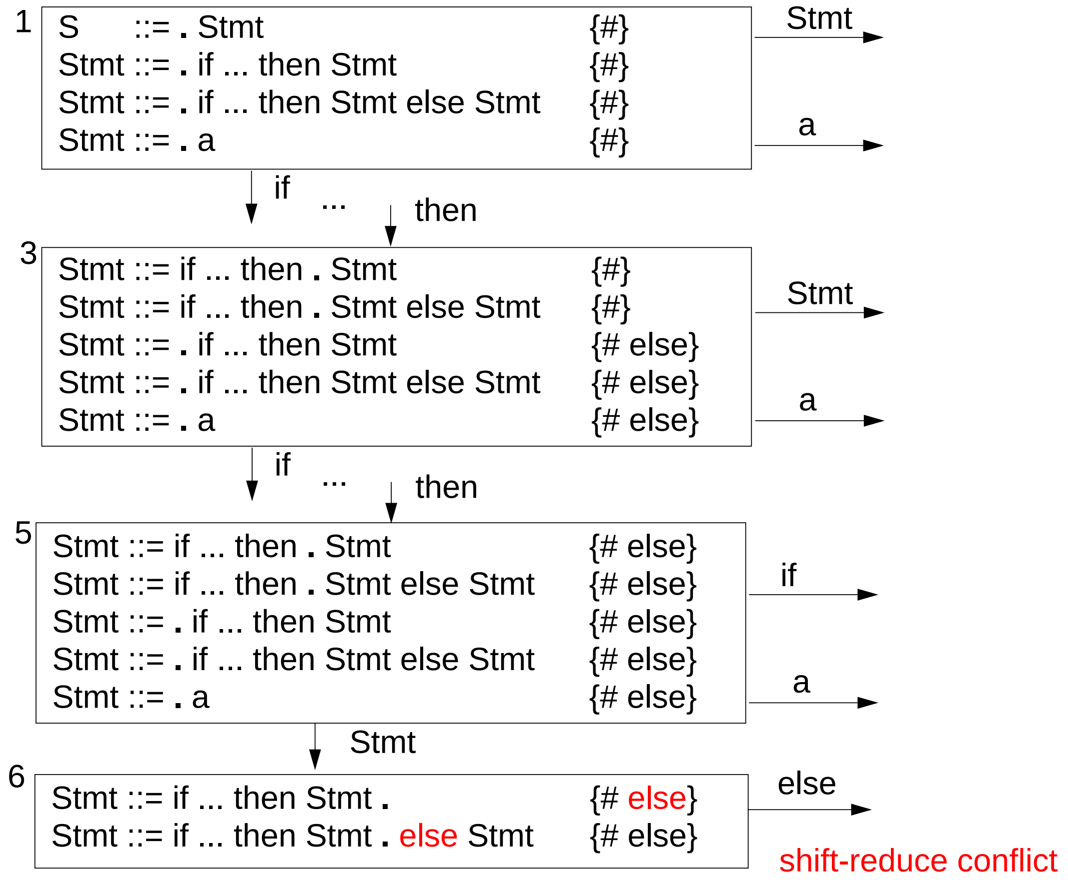
Suggested reading:

Kastens / Übersetzerbau, Section 4.3

Questions:

- Why can a shift-shift conflict not exist?
- In LR(0) automata items do not have a right-context set. Explain why a state with a reduce item may not contain any other item.

Shift-reduce conflict for „dangling else“ ambiguity



Lecture Programming Languages and Compilers WS 2013/14 / Slide 318

Objectives:

See a conflict in an automaton

In the lecture:

Explain

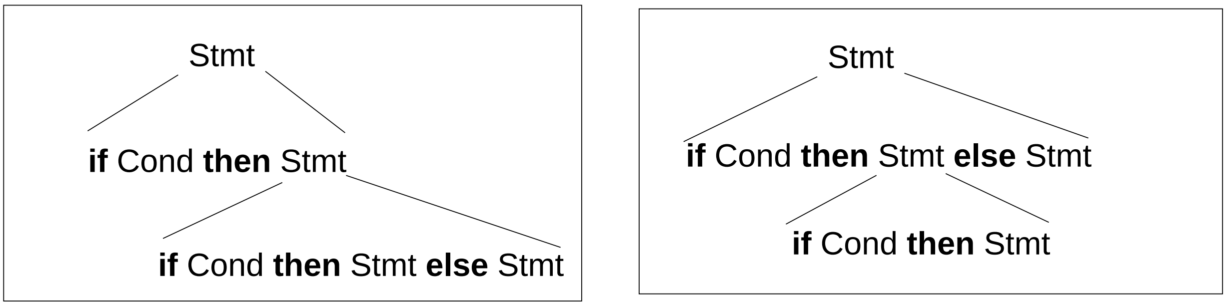
- the construction
- a solution of the conflict: The automaton can be modified such that in state 6, if an else is the next input token, it is shifted rather than a reduction is made. In that case the ambiguity is solved such that the else part is bound to the inner if. That is the structure required in Pascal and C. Some parser generators can be instructed to resolve conflicts in this way.

Suggested reading:

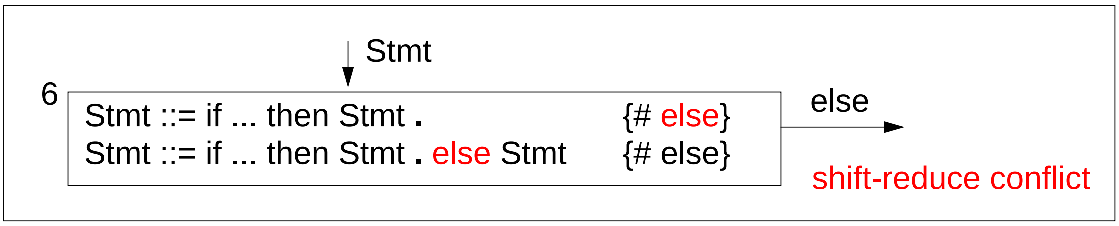
Kastens / Übersetzerbau, Section 4.3

Decision of ambiguity

dangling else ambiguity:



desired solution for Pascal, C, C++, Java



State 6 of the automaton can be modified such that
an input token **else is shifted** (instead of causing a reduction);
yields the desired behaviour.

Some parser generators allow such modifications.

Lecture Programming Languages and Compilers WS 2013/14 / Slide 319

Objectives:

Understand modification of automaton

In the lecture:

Explain why the desired effect is achieved.

Simplified LR grammar classes

LR(1):

too many states for practical use**,** because right-contexts distinguish many states.
Strategy: simplify right-contexts sets; **fewer states**; grammar classes less powerful

LALR(1):

construct LR(1) automaton,
identify LR(1) states if their items
differ only in their right-context sets,
unite the sets for those items;

yields the states of the **LR(0) automaton**
augmented by the "exact" LR(1) right-context.

State-of-the-art parser generators
accept LALR(1)

q

A ::= u . v   R1
B ::= x . y   R2
C ::= z .   R3

r

A ::= u . v   R1'
B ::= x . y   R2'
C ::= z .   R3'

q, r identified:

qr

A ::= u . v $R1 \cup R1'$
B ::= x . y $R2 \cup R2'$
C ::= z . $R3 \cup R3'$

SLR(1):

LR(0) states; in reduce items
use larger right-context sets for decision:
[A ::= u . Follow (A)]

A ::= u . v
B ::= x . y
C ::= z .   **Follow(C)**

LR(0):

all items **without right-context**
Consequence: reduce items only in singleton sets

C ::= z .

Lecture Programming Languages and Compilers WS 2013/14 / Slide 320

Objectives:

Understand relations between LR classes

In the lecture:

Explain:

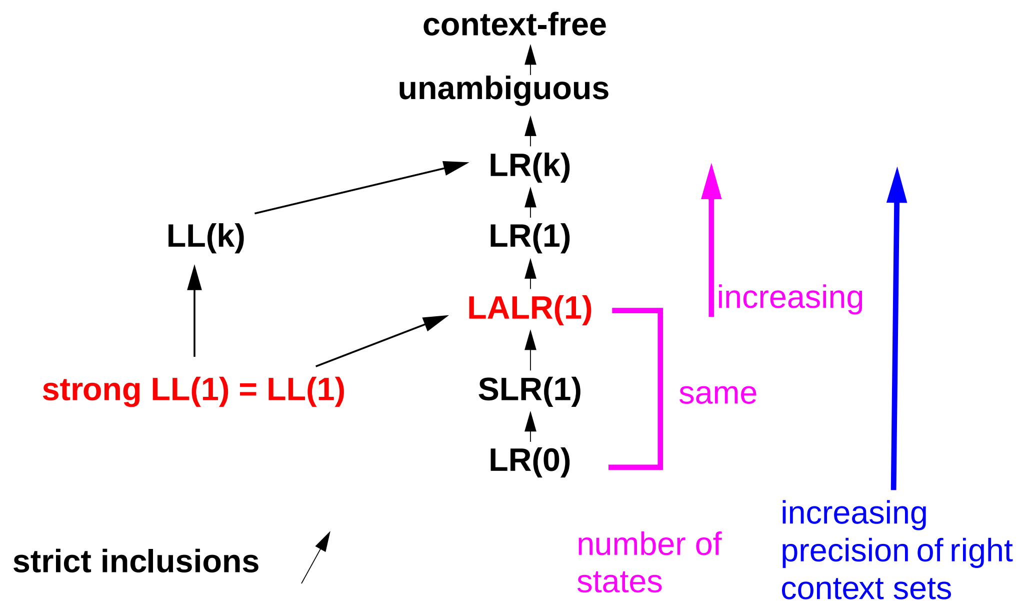
- LALR(1), SLR(1), LR(0) automata have the same number of states,
- compare their states,
- discuss the grammar classes for the example on slide PLaC-3.13.

Suggested reading:

Kastens / Übersetzerbau, Section 4.3

Questions:

Hierarchy of grammar classes



Lecture Programming Languages and Compilers WS 2013/14 / Slide 321

Objectives:

Understand the hierarchy

In the lecture:

Explain:

- the reasons for the strict inclusions,

Suggested reading:

Kastens / Übersetzerbau, Section 4.3

Questions:

- Assume that the LALR(1) construction for a given grammar yields conflicts. Classify the potential reasons using the LR hierarchy.

Reasons for LALR(1) conflicts

Grammar condition does not hold:

context-free		
unambiguous	**ambiguous**	most cases
LR(k)	**unbounded lookahead** needed	
LR(1)	**fixed length lookahead > 1** needed	
LALR(1)	**merge of LR(1) states** introduces **conflicts**	rare cases

LALR(1) parser generator can not distinguish these cases.

Lecture Programming Languages and Compilers WS 2013/14 / Slide 321a

Objectives:

Distinguish cases

In the lecture:

The cases are explained.

LR(1) but not LALR(1)

Identification of LR(1) states causes non-disjoint right-context sets.

Artificial example:

Grammar:
Z ::= S
S ::= A a
S ::= B c
S ::= b A c
S ::= b B a
A ::= d.
B ::= d.

LR(1) states

Z ::= . S	{#}
S ::= . A a	{#}
S ::= . B c	{#}
S ::= . b A c	{#}
S ::= . b B a	{#}
A ::= . d	{a}
B ::= . d	{c}

d →

A ::= d .	{a}
B ::= d .	{c}

b ↓

S ::= b . A c	{#}
S ::= b . B a	{#}
A ::= . d	{c}
B ::= . d	{a}

d →

A ::= d .	{c}
B ::= d .	{a}

identified states

LALR(1) state

A ::= d .	{a, c}
B ::= d .	{a, c}

Avoid the distinction between A and B - at least in one of the contexts.

Lecture Programming Languages and Compilers WS 2013/14 / Slide 321b

Objectives:

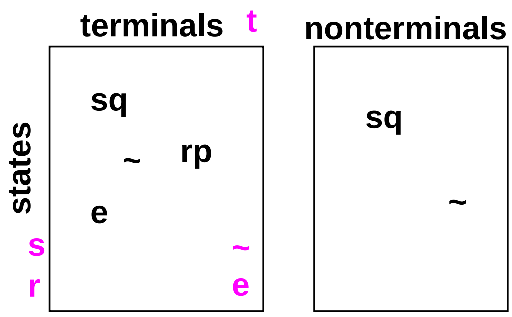
Understand source of conflicts

In the lecture:

Explain the pattern, and why identification of states causes a conflict.

Table driven implementation of LR automata

LR parser tables



sq: shift into state q

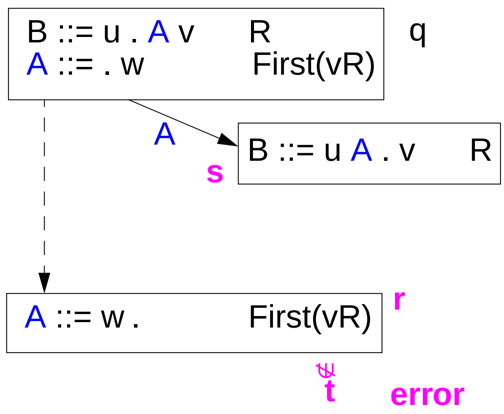
rp: reduce production p

e: error

~: not reachable
don't care

nonterminal table

- has **no reduce entries** and **no error entries** (only **shift** and **don't-care** entries)
  reason:
  a reduction to A reaches a state from where a shift under A exists (by construction)



unreachable entries in terminal table:
if t is erroneus input in state r, then
state s will not be reached with input t

Lecture Programming Languages and Compilers WS 2013/14 / Slide 322

Objectives:

Understand properties of LR tables

In the lecture:

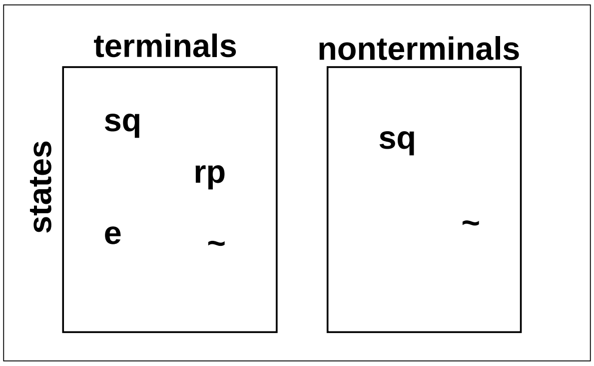
Explanation of

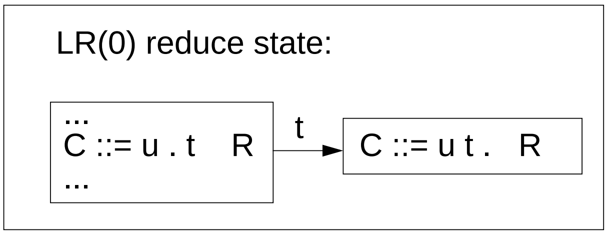
- pair of tables and their entries,
- unreachable entries,

Questions:

- Why are there no error entries in the nonterminal part?
- Why are there unreachable entries?

Implementation of LR automata





Compress tables:

- **merge rows or columns** that differ only in irrelevant entries; method: graph coloring
- extract a **separate error matrix** (bit matrix); increases the chances for merging
- **normalize the values of rows or columns**; yields smaller domain; supports merging
- **eliminate LR(0) reduce states**; new operation in predecessor state: **shift-reduce** eliminates about 30% of the states in practical cases

About 10-20% of the original table sizes can be achieved!

Directly programmed LR-automata are possible - but usually too large.

Lecture Programming Languages and Compilers WS 2013/14 / Slide 323

Objectives:

Implementation of LR tables

In the lecture:

Explanation of

- compression techniques, derived from general table compression,
- Singleton reduction states yield an effective optimization.

Questions:

- Why are there no error entries in the nonterminal part?
- Why are there unreachable entries?
- Why does a parser need a shift-reduce operation if the optimization of LR(0)-reduction states is applied?

Parser generators

PGS	Univ. Karlsruhe; in Eli	**LALR(1), table-driven**
Cola	Univ. Paderborn; in Eli	**LALR(1), optional: table-driven or directly programmed**
Lalr	Univ. / GMD Karlsruhe	**LALR(1), table-driven**
Yacc	Unix tool	**LALR(1), table-driven**
Bison	Gnu	**LALR(1), table-driven**
Llgen	Amsterdam Compiler Kit	**LL(1), recursive descent**
Deer	Univ. Colorado, Bouder	**LL(1), recursive descent**

Form of grammar specification:
EBNF: Cola, PGS, Lalr; **BNF**: Yacc, Bison

Error recovery:

simulated continuation, automatically generated:	Cola, PGS, Lalr
error productions, hand-specified:	Yacc, Bison

Actions:
statements in the implementation language

at the end of productions:	Yacc, Bison
anywhere in productions:	Cola, PGS, Lalr

Conflict resolution:

modification of states (reduce if ...)	Cola, PGS, Lalr
order of productions:	Yacc, Bison
rules for precedence and associativity:	Yacc, Bison

Implementation languages:
C: Cola, Yacc, Bison **C, Pascal, Modula-2, Ada**: PGS, Lalr

Lecture Programming Languages and Compilers WS 2013/14 / Slide 324

Objectives:

Overview over parser generators

In the lecture:

- Explain the significance of properties

Suggested reading:

Kastens / Übersetzerbau, Section 4.5

3.5 Syntax Error Handling
General criteria

- **recognize error as early as possible**
  LL and LR can do that:
  no transitions after error position
- **report the symptom in terms of the source text**
  rather than in terms of the state of the parser
- **continue parsing short after the error position**
  analyze as much as possible
- **avoid avalanche errors**
- **build a tree that has a correct structure**
  later phases must not break
- **do not backtrack, do not undo actions,**
  not possible for semantic actions
- **no runtime penalty for correct programs**

Lecture Programming Languages and Compilers WS 2013/14 / Slide 325

Objectives:

Accept strong requirements

In the lecture:

- The reasons for and the consequences of the requirements are discussed.
- Some of the requirements hold for error handling in general - not only that of the syntactic analysis.

Error position

Error recovery: Means that are taken by the parser after recognition of a syntactic error in order to continue parsing

Correct prefix: The token sequence $w \in T^*$ is a correct prefix in the language L(G), if there is an $u \in T^*$ such that $\mathbf{w\ u} \in \mathbf{L(G)}$; i. e. w can be extended to a sentence in L(G).

Error position: t is the (first) error position in the **input w t x** , where $t \in T$ and $w, x \in T^*$, if **w is a correct prefix** in L(G) and **w t is not a correct prefix**.

Example: `int compute (int i) { a = i * / c; return i;}`

w = `int compute (int i) { a = i *`, t = `/`

LL and LR parsers recognize an error at the error position; they can not accept t in the current state.

Lecture Programming Languages and Compilers WS 2013/14 / Slide 326

Objectives:

Error position from the view of the parser

In the lecture:

Explain the notions with respect to parser actions using the examples.

Questions:

Assume the programmer omitted an opening parenthesis.

- Where is the error position?
- What is the symptom the parser recognizes?

Error recovery

Continuation point:
A token d at or behind the error position t such that **parsing of the input continues at d**.

Error repair
with respect to a consistent derivation
- regardless the intension of the programmer!

Let the input be w t x with the error position at t and let w t x = w y d z, then the recovery (conceptually) **deletes y** and **inserts v**, such that **w v d is a correct prefix** in L(G), with d ∈ T and w, y, v, z ∈ T*.

error position

w t x =
w y d z
w v d z

continuation

Examples:

```
  w      y d  z
a = i * / c;...
a = i *   c;...
```

delete /

```
  w     yd    z
a = i * / c;...
a = i *e/ c;...
```

insert error identifier e

```
   w      y d z
a = i * / c;...
a = i * e  ;...
```

delete / c
and **insert** error id. **e**

Lecture Programming Languages and Compilers WS 2013/14 / Slide 327

Objectives:

Understand error recovery

In the lecture:

Explain the notions with respect to parser actions using the examples.

Questions:

Assume the programmer omitted an opening parenthesis.

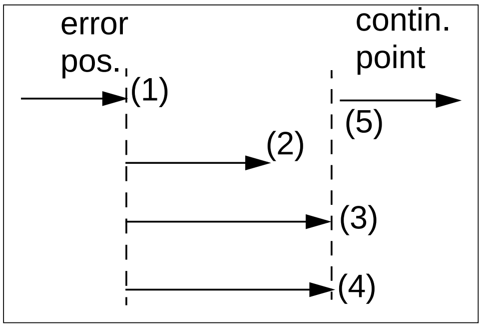
- What could be a suitable repair?

Recovery method: simulated continuation

Problem: Determine a continuation point close to the error position and reach it.
Idea: Use parse stack to determine a set D of tokens as potential continuation points.

Steps of the method:

1. **Save the contents of the parse stack** when an error is recognized.
2. **Compute a set D $\subseteq$ T of tokens that may be used as continuation point** (**anchor set**)
   Let a modified parser run to completion:
   Instead of reading a token from input it is inserted into D; (modification given below)
3. **Find a continuation point d**: Skip input tokens until a token of D is found.
4. **Reach the continuation point d**:
   Restore the saved parser stack as the current stack.
   Perform dedicated transitions until d is acceptable.
   Instead of reading tokens (conceptually) insert tokens.
   Thus a correct prefix is constructed.
5. **Continue normal parsing**.



Augment parser construction for steps 2 and 4:
For each parser state select a transition and its token,
such that the parser empties its stack and terminates as fast as possible.
This selection can be **generated automatically**.
The quality of the recovery can be improved by deletion/insertion of elements in D.

Lecture Programming Languages and Compilers WS 2013/14 / Slide 328

Objectives:

Error recovery can be generated

In the lecture:

- Explain the idea and the steps of the method.
- The method yields a correct parse for any input!
- Other, less powerful methods determine sets D statically at parser construction time, e. g. semicolon and curly bracket for errors in statements.

Questions:

- How does this method fit to the general requirements for error handling?