Design of concrete grammars

Objectives

The concrete grammars 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 provable related to the **documented grammar**;
- can be **mapped to** a suitable **abstract grammar**.

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 any modification**.
  (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**

Grammar design together with language design

Read grammars before writing a new grammar.

Apply **grammar patterns systematically** (cf. GdP-2.5, GdP-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

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 unreasonable complex**:
  e. g. distinction of compile-time expressions from ordinary
  expressions requires duplication of the expression syntax.

Reasons of LALR(1) conflicts

Grammar condition does not hold:

Grammar class	Reason	Frequency
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.

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

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.

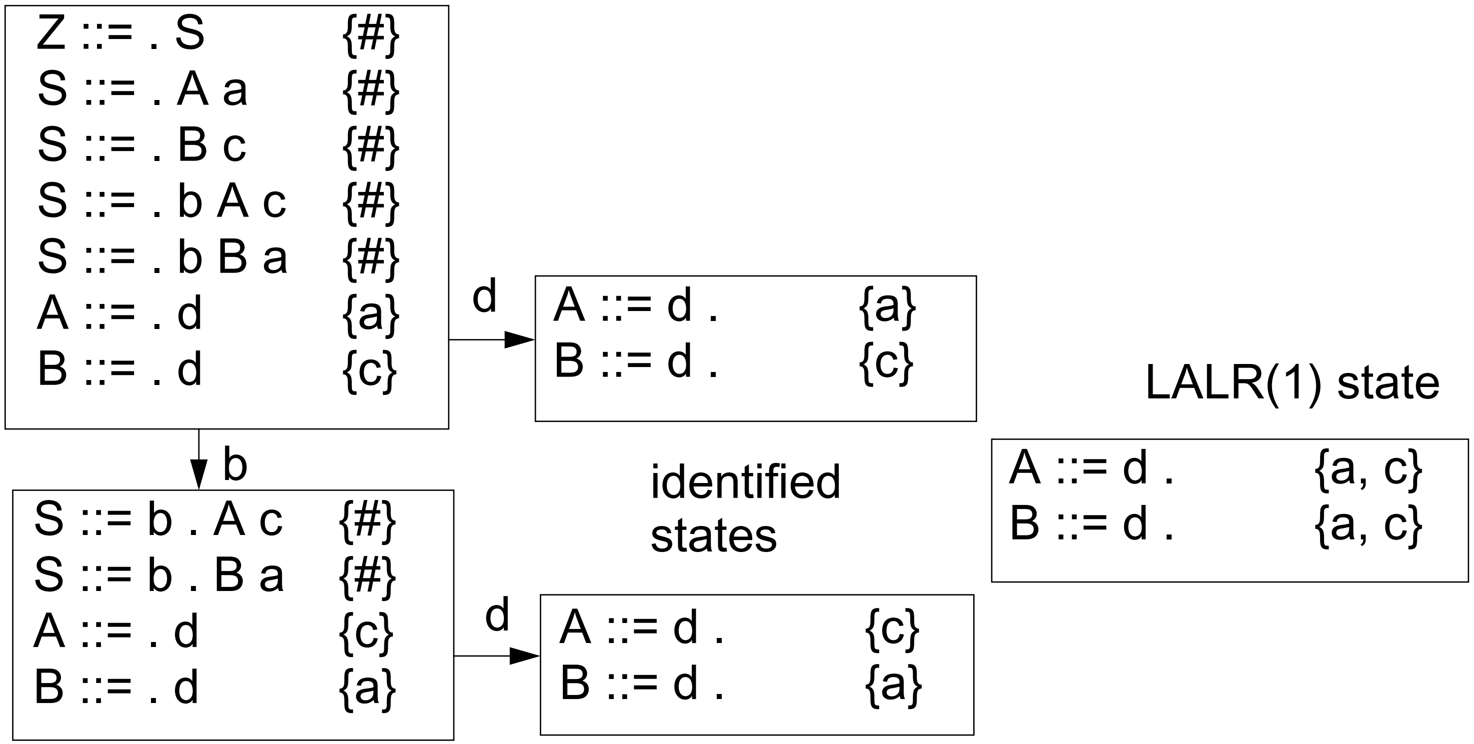
LR(1) but not LALR(1)

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



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