G51MAL: Lecture 17

LR(0) Parsing and Parser Generators

Henrik Nilsson

University of Nottingham, UK

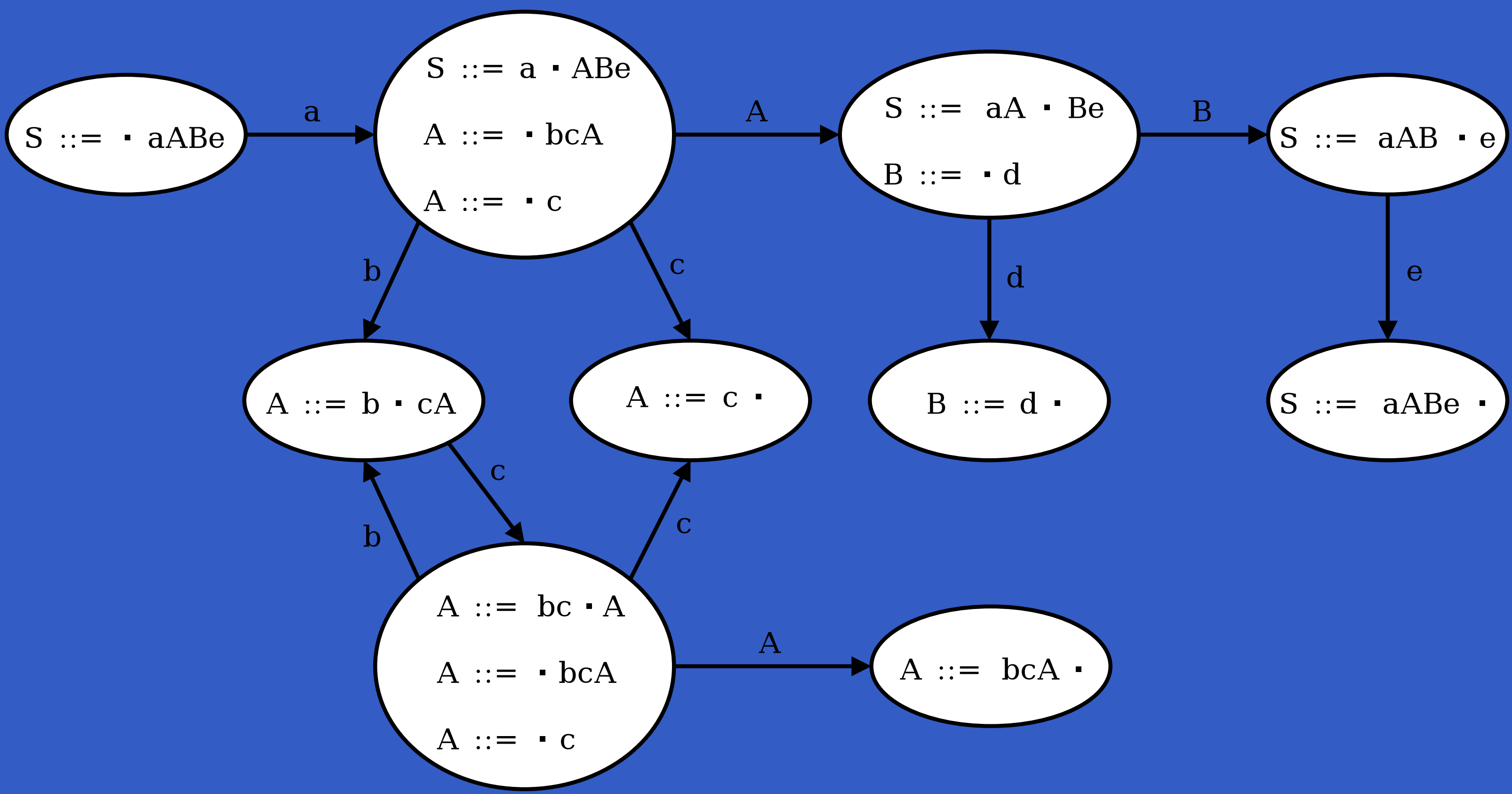
This Lecture

- Briefly explaining the basics of LR(0) parsing to show practical application of Deterministic PDA.
- Quick outline of the Happy parser generator.

LR(0) Parsing (1)

A DFA recognising ***viable prefixes*** for the CFG

$$S ::= aABe \qquad A ::= bcA \mid c \qquad B ::= d$$



LR(0) Parsing (2)

How to construct such a DFA is beyond the scope of this course. See e.g. Aho, Sethi, Ullman (1986) for details. However, some observations:

LR(0) Parsing (2)

How to construct such a DFA is beyond the scope of this course. See e.g. Aho, Sethi, Ullman (1986) for details. However, some observations:

- Consider a right-sentential form, e.g. $abcAde$. Note that prefixes ϵ, a, ab, abc, $abcA$ are recognised by the DFA (all states are considered final).

LR(0) Parsing (2)

How to construct such a DFA is beyond the scope of this course. See e.g. Aho, Sethi, Ullman (1986) for details. However, some observations:

- Consider a right-sentential form, e.g. $abcAde$. Note that prefixes ϵ, a, ab, abc, $abcA$ are recognised by the DFA (all states are considered final).
- Note that strings like acb, which are not a prefix of any right-sentential form, are not accepted.

LR(0) Parsing (3)

Given a DFA recognising viable prefixes, a LR(0) parser can be constructed as follows:

LR(0) Parsing (3)

Given a DFA recognising viable prefixes, a LR(0) parser can be constructed as follows:

- In a state ***without complete items***: ***Shift***

LR(0) Parsing (3)

Given a DFA recognising viable prefixes, a LR(0) parser can be constructed as follows:

- In a state ***without complete items***: ***Shift***
  - Read next terminal symbol and push it onto internal ***parse stack***.

LR(0) Parsing (3)

Given a DFA recognising viable prefixes, a LR(0) parser can be constructed as follows:

- In a state ***without complete items***: ***Shift***
  - Read next terminal symbol and push it onto internal ***parse stack***.
  - Move to new state by following edge labelled by the read terminal.

LR(0) Parsing (4)

- In a state with a ***single complete item***: **Reduce**

LR(0) Parsing (4)

- In a state with a ***single complete item***: *Reduce*
  - The top of the parse stack contains the ***handle*** of the current right-sentential form (since we have recognised a viable prefix for which a single ***complete*** item is valid).

LR(0) Parsing (4)

- In a state with a ***single complete item***: ***Reduce***
  - The top of the parse stack contains the ***handle*** of the current right-sentential form (since we have recognised a viable prefix for which a single ***complete*** item is valid).
  - The handle is just the RHS of the valid item.

LR(0) Parsing (4)

- In a state with a ***single complete item***: ***Reduce***
  - The top of the parse stack contains the ***handle*** of the current right-sentential form (since we have recognised a viable prefix for which a single ***complete*** item is valid).
  - The handle is just the RHS of the valid item.
  - Reduce to the previous right-sentential form by replacing the handle on the parse stack with the LHS of the valid item.

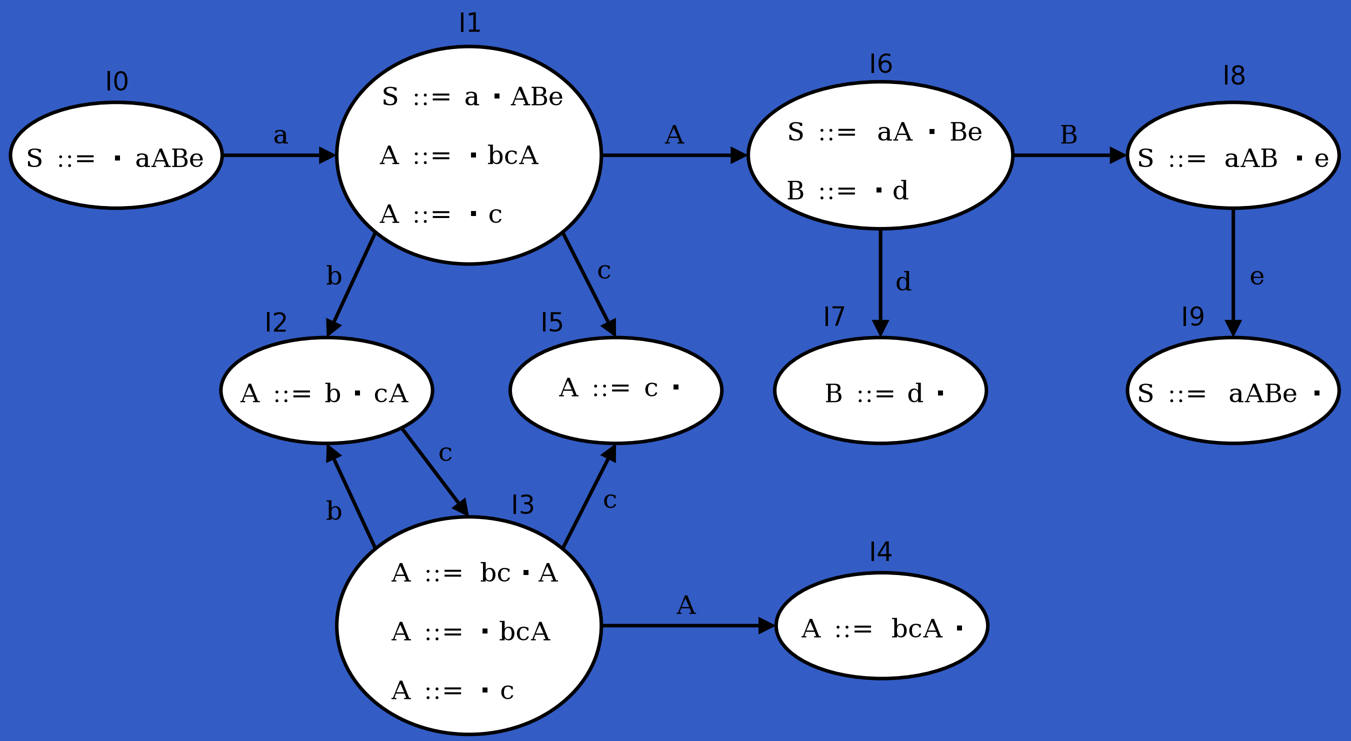
LR(0) Parsing (4)

- In a state with a ***single complete item***: ***Reduce***
  - The top of the parse stack contains the ***handle*** of the current right-sentential form (since we have recognised a viable prefix for which a single ***complete*** item is valid).
  - The handle is just the RHS of the valid item.
  - Reduce to the previous right-sentential form by replacing the handle on the parse stack with the LHS of the valid item.
  - Move to the state indicated by the new viable prefix on the parse stack.

LR(0) Parsing (5)

- If a state contains both complete and incomplete items, or if a state contains more than one complete item, then the grammar was not LR(0).

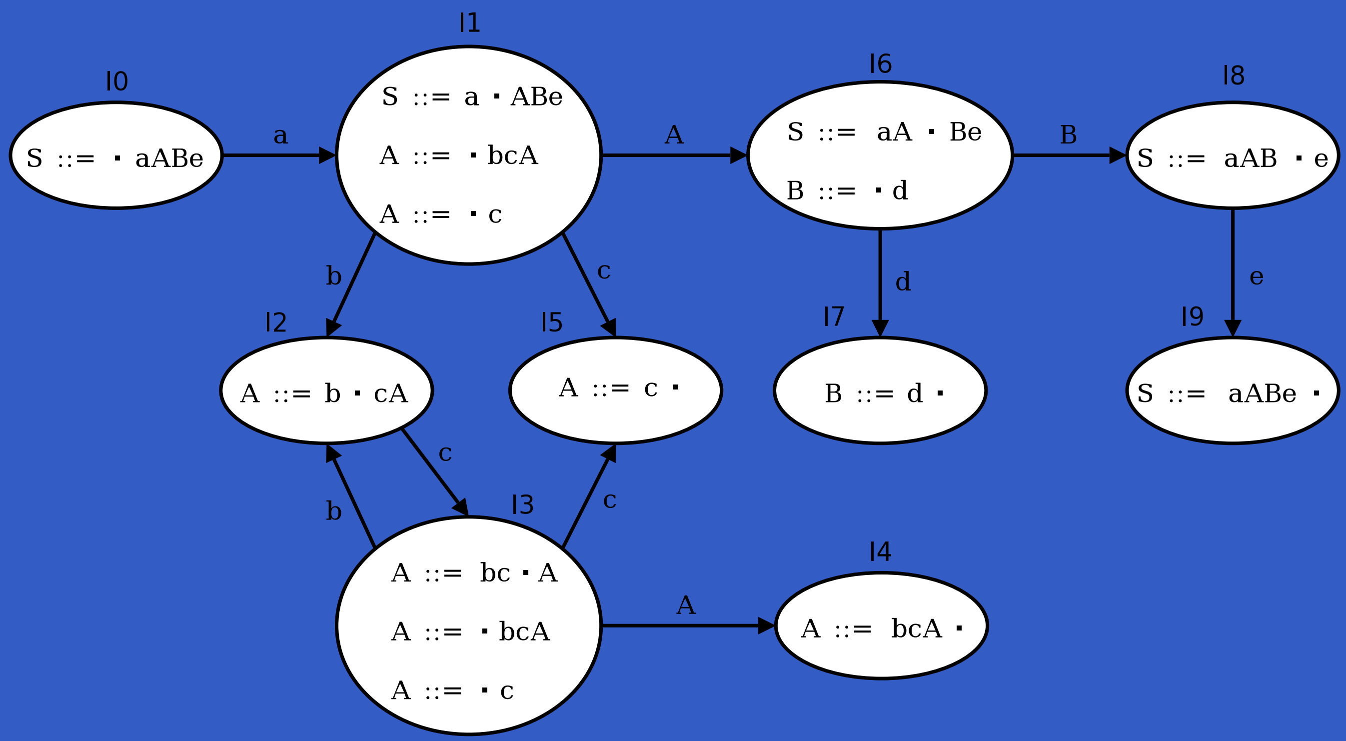
LR(0) Parsing (6)



Note: γw is the current right-sentential form.

State	Stack (γ)	Input (w)	Move
I0	ϵ	$abccde$	

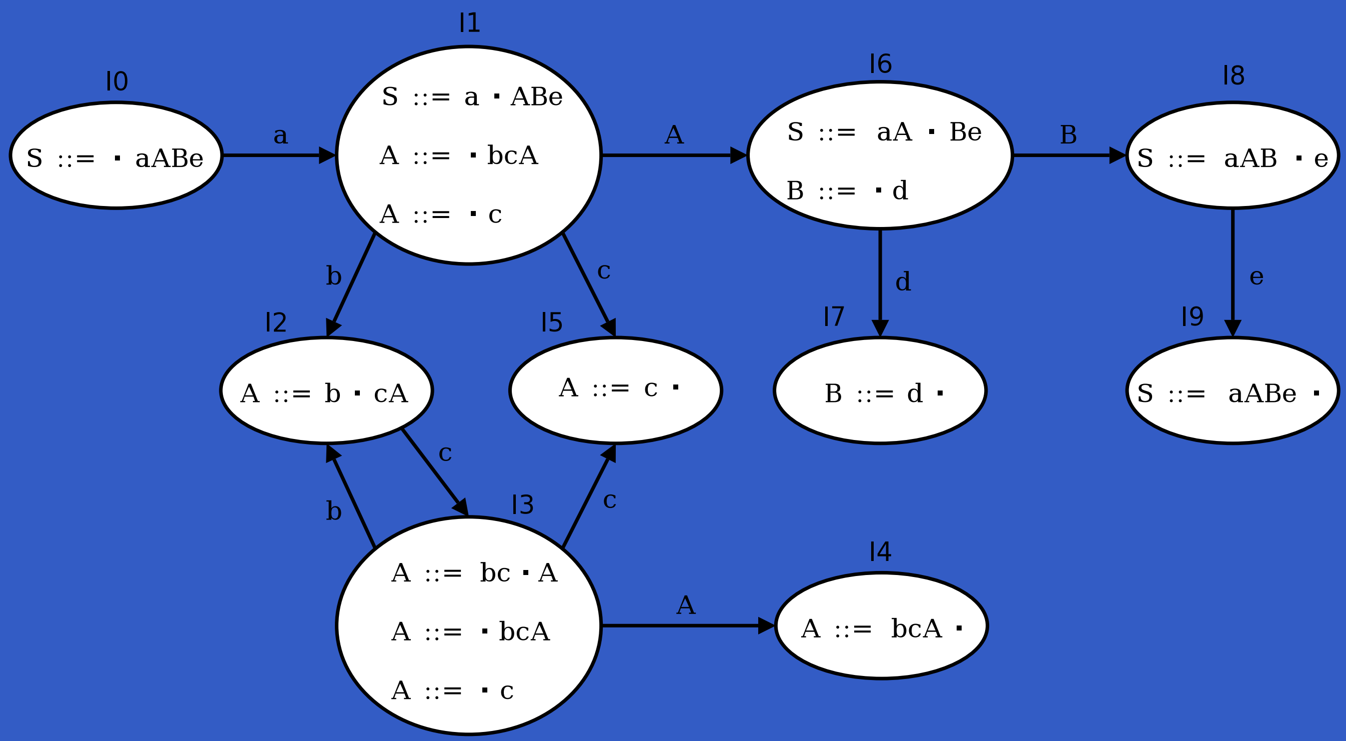
LR(0) Parsing (6)



Note: γw is the current right-sentential form.

State	Stack (γ)	Input (w)	Move
I0	ϵ	$abccde$	Shift

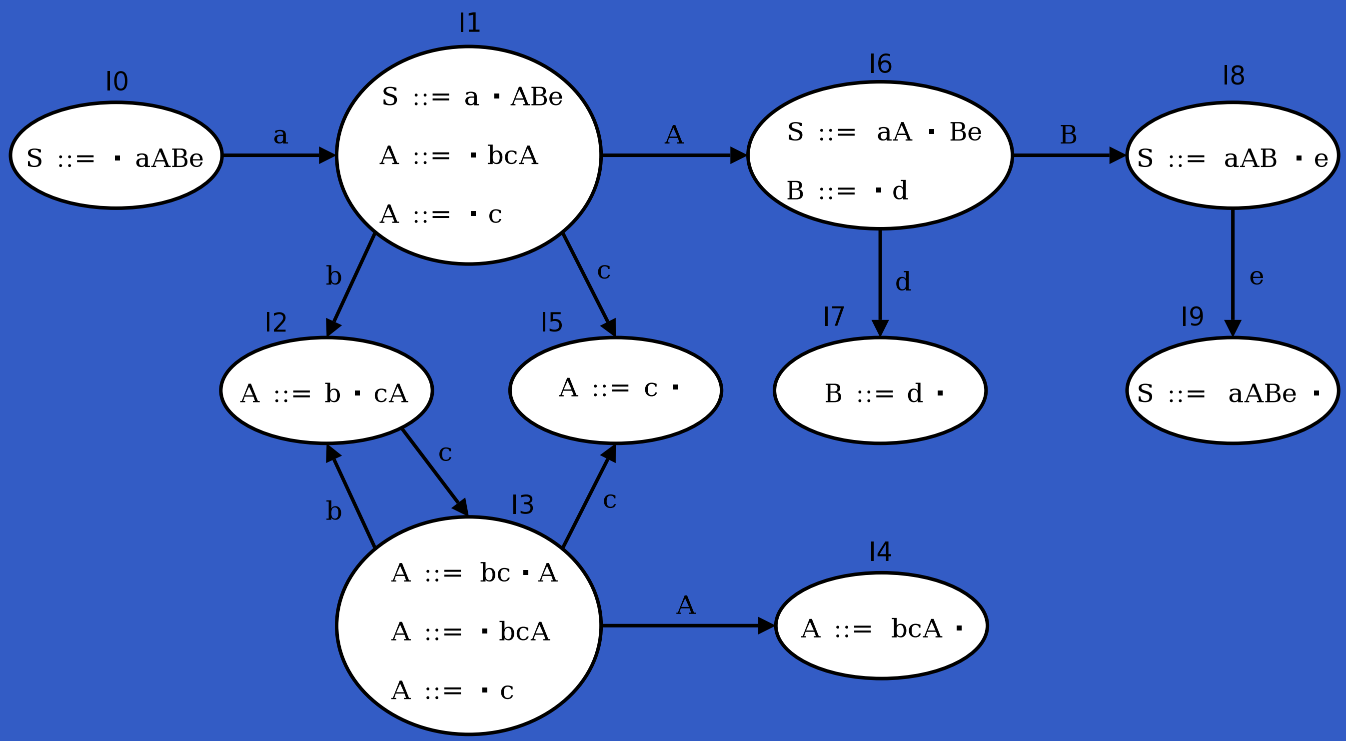
LR(0) Parsing (6)



Note: γw is the current right-sentential form.

State	Stack (γ)	Input (w)	Move
I0	ϵ	$abccde$	Shift
I1	a	$bccde$	

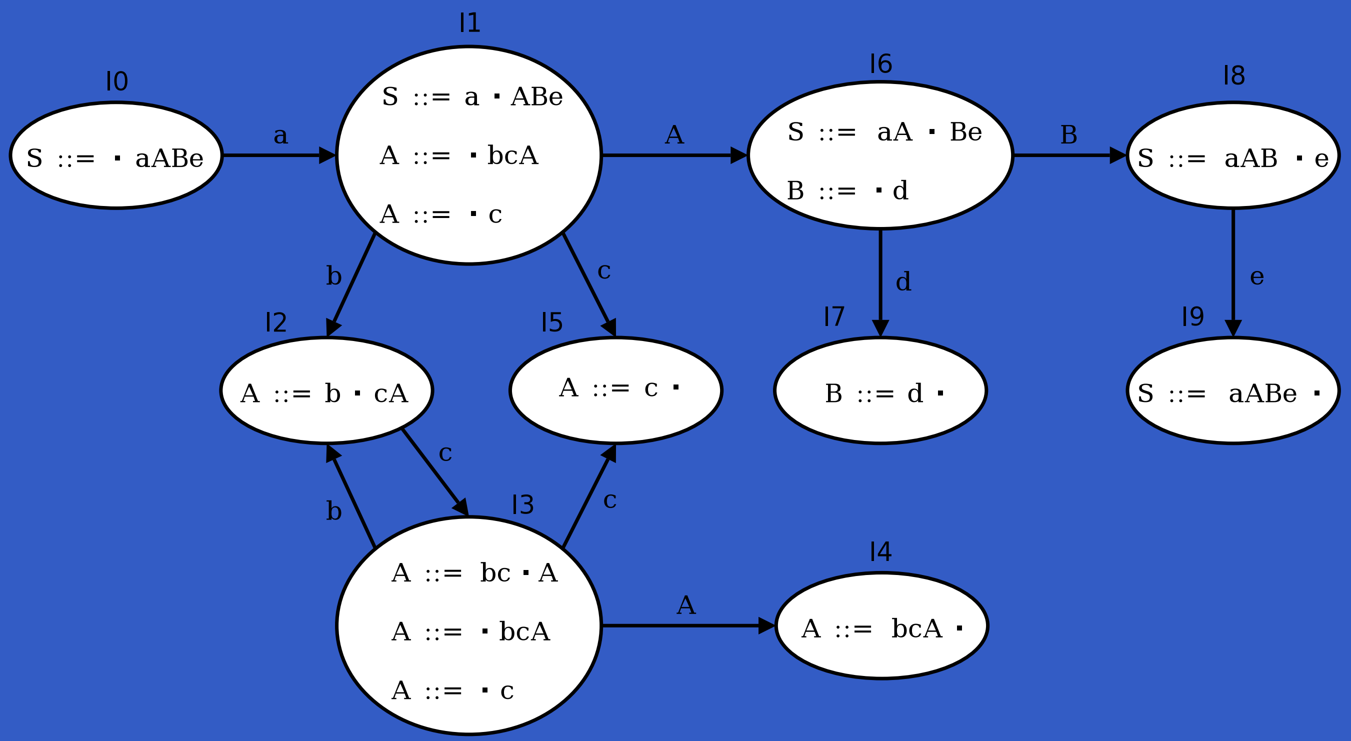
LR(0) Parsing (6)



Note: γw is the current right-sentential form.

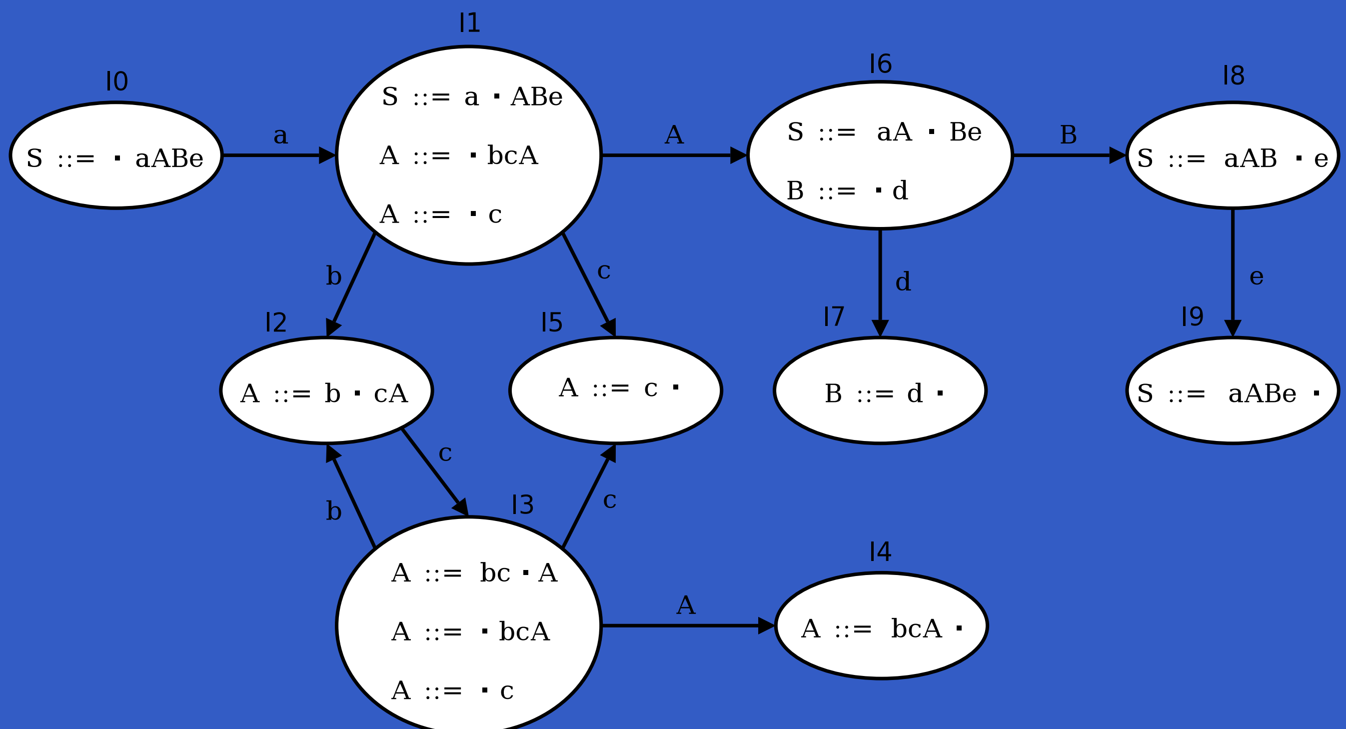
State	Stack (γ)	Input (w)	Move
I0	ϵ	$abccde$	Shift
I1	a	$bccde$	Shift

LR(0) Parsing (7)



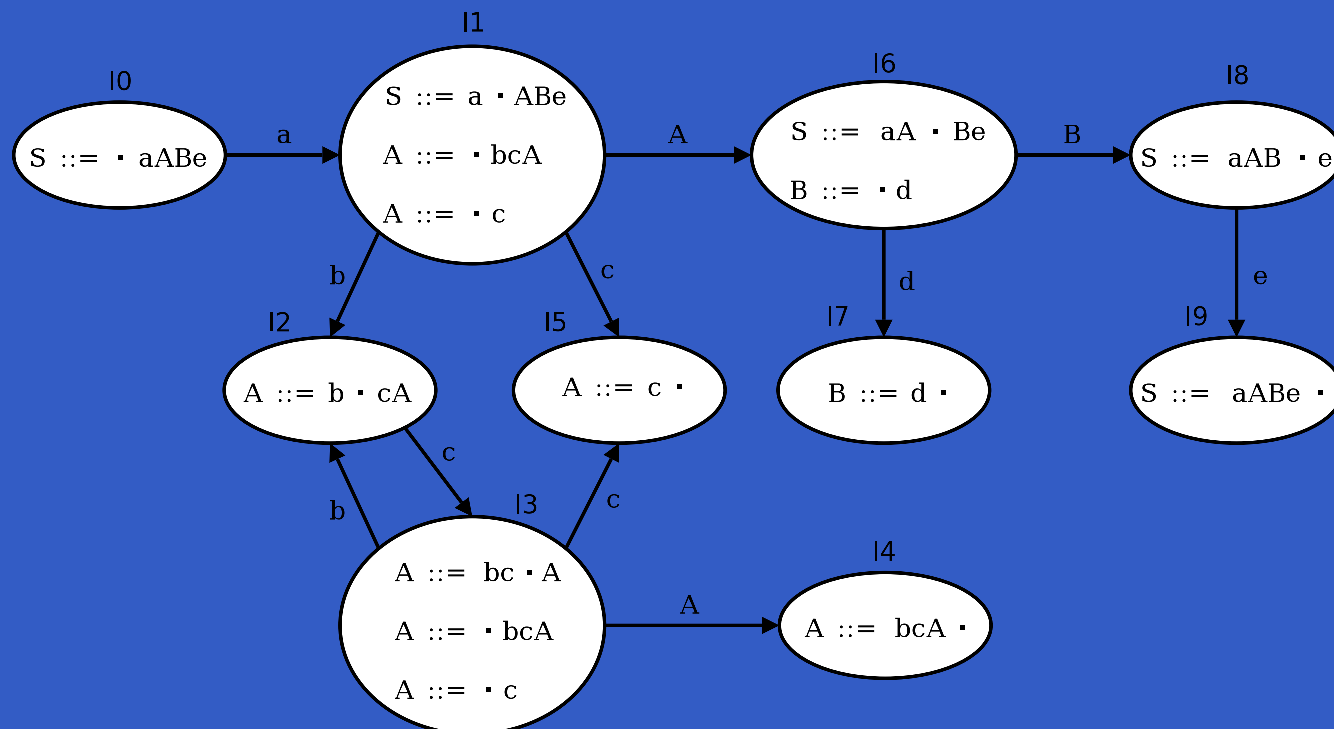
State	Stack (γ)	Input (w)	Move
I2	ab	$ccde$	

LR(0) Parsing (7)



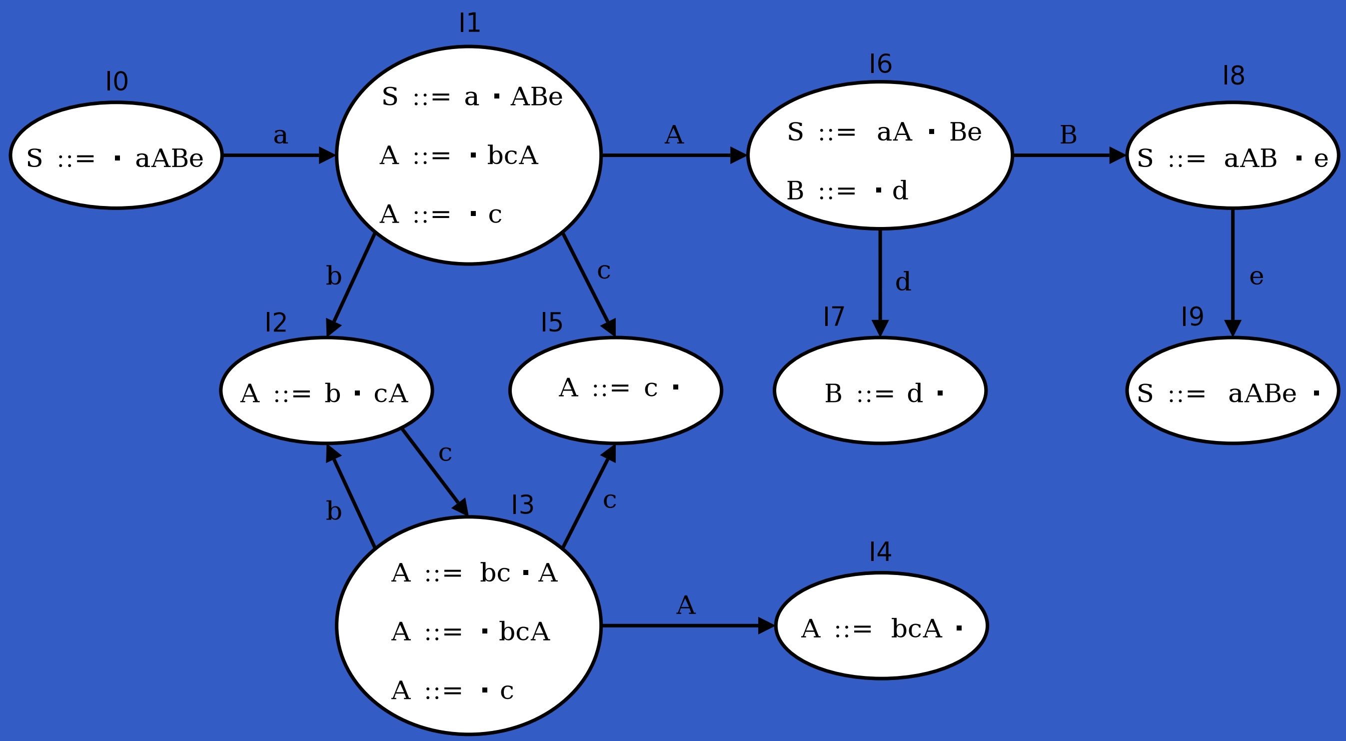
State	Stack (γ)	Input (w)	Move
I2	ab	$ccde$	Shift

LR(0) Parsing (7)



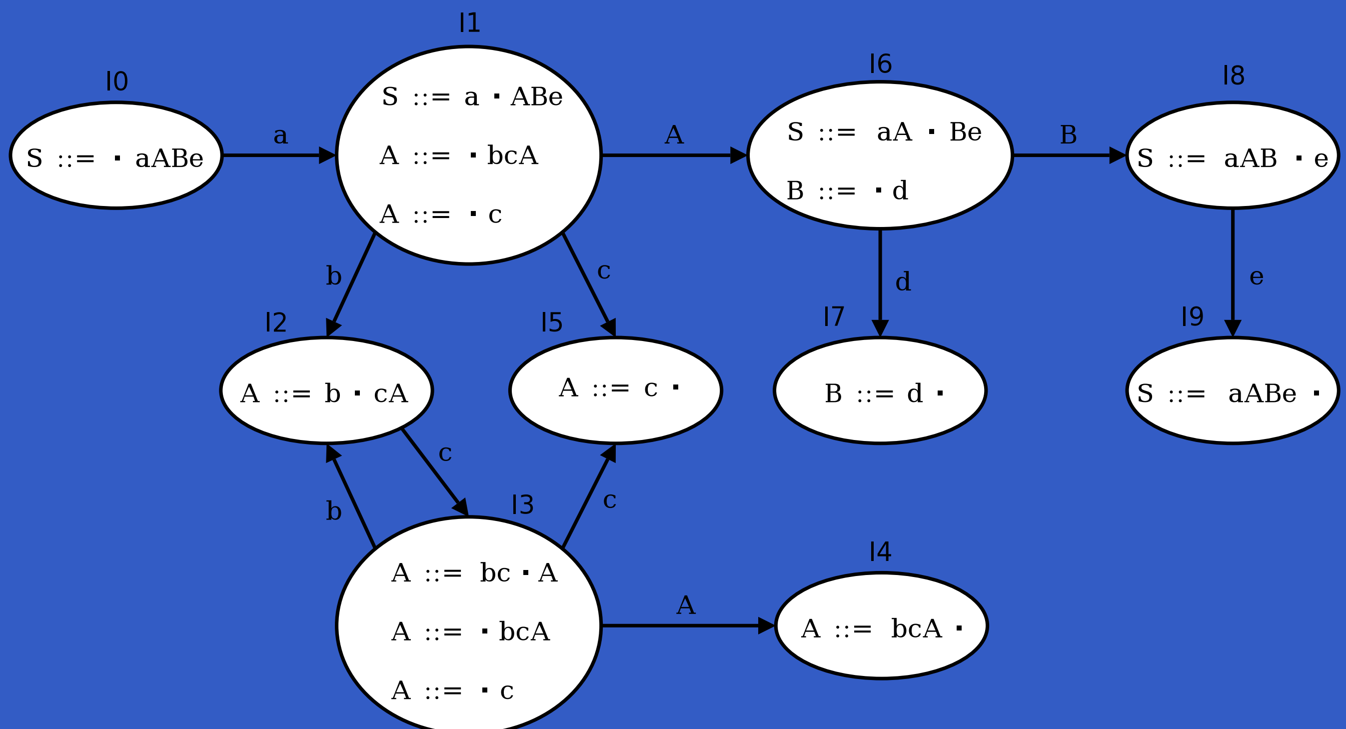
State	Stack (γ)	Input (w)	Move
I2	ab	$ccde$	Shift
I3	abc	cde	

LR(0) Parsing (7)



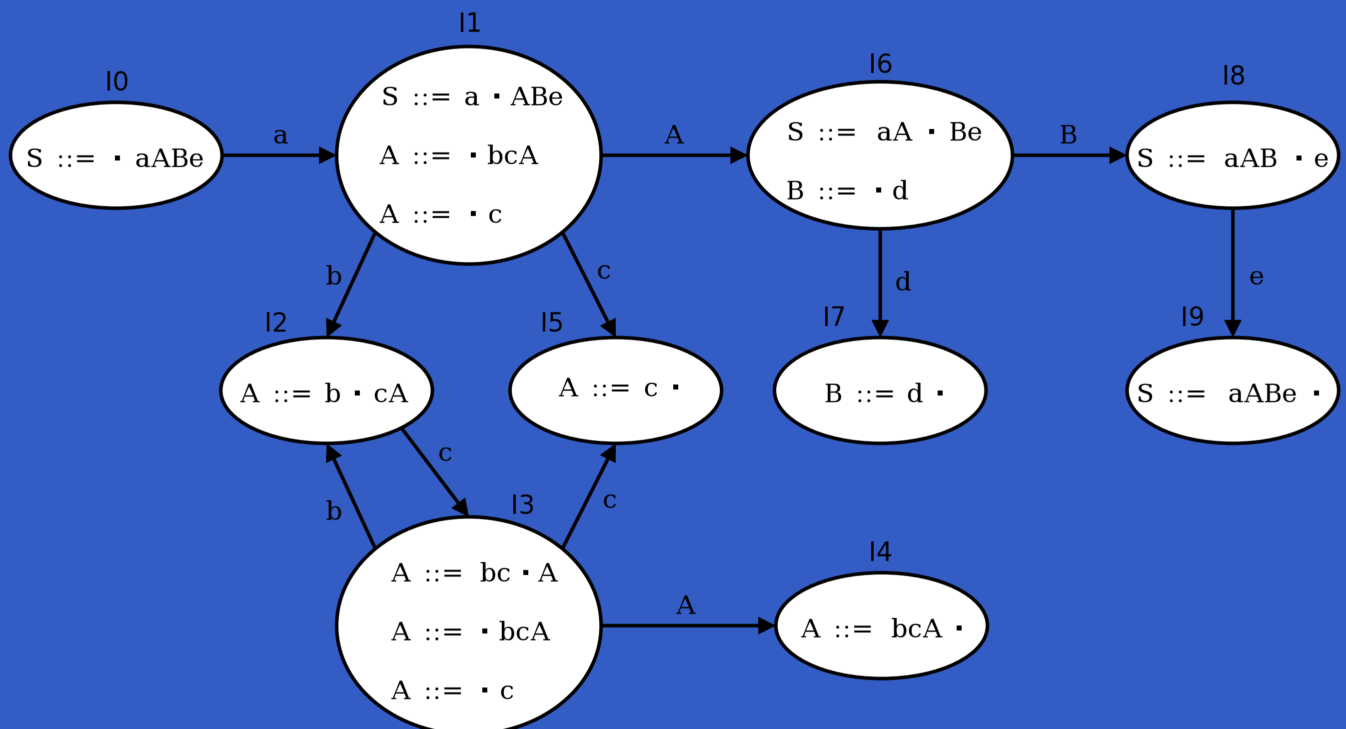
State	Stack (γ)	Input (w)	Move
I2	ab	$ccde$	Shift
I3	abc	cde	Shift

LR(0) Parsing (7)



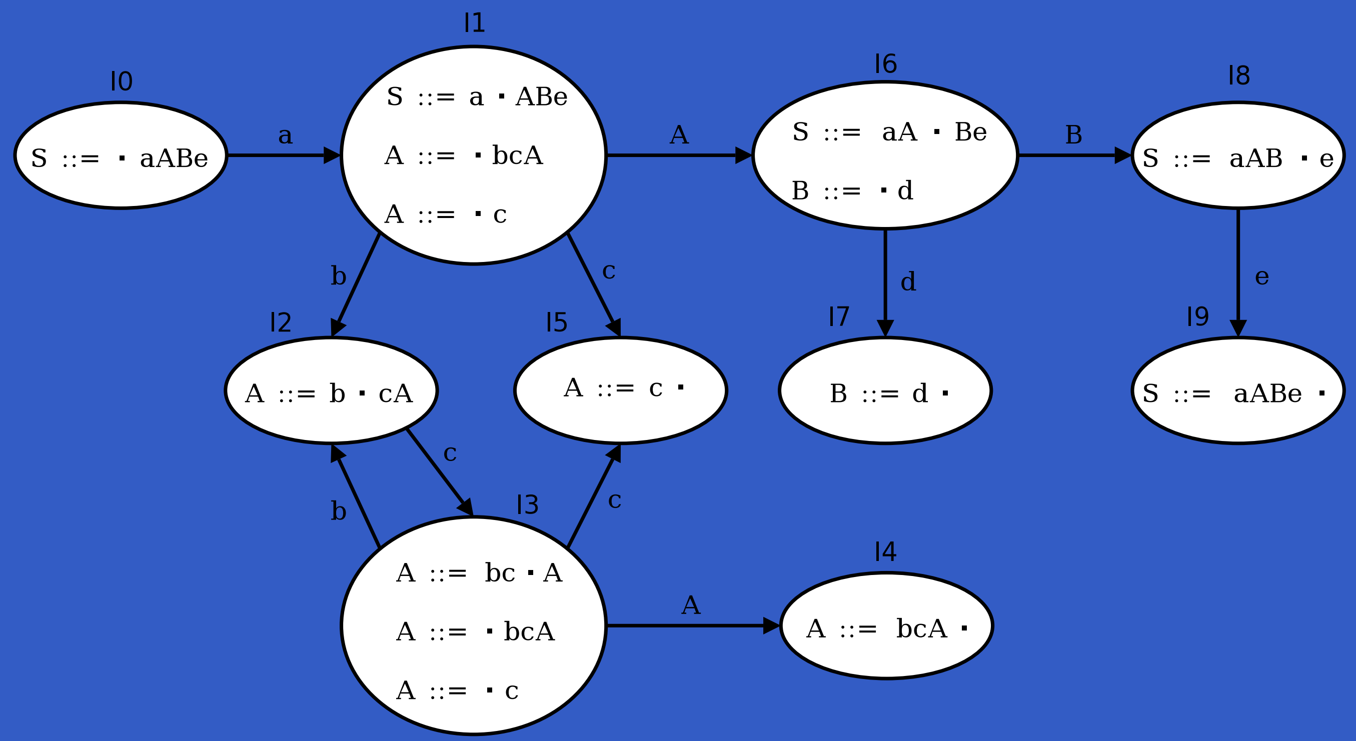
State	Stack (γ)	Input (w)	Move
I2	ab	$ccde$	Shift
I3	abc	cde	Shift
I5	$abcc$	de	

LR(0) Parsing (7)



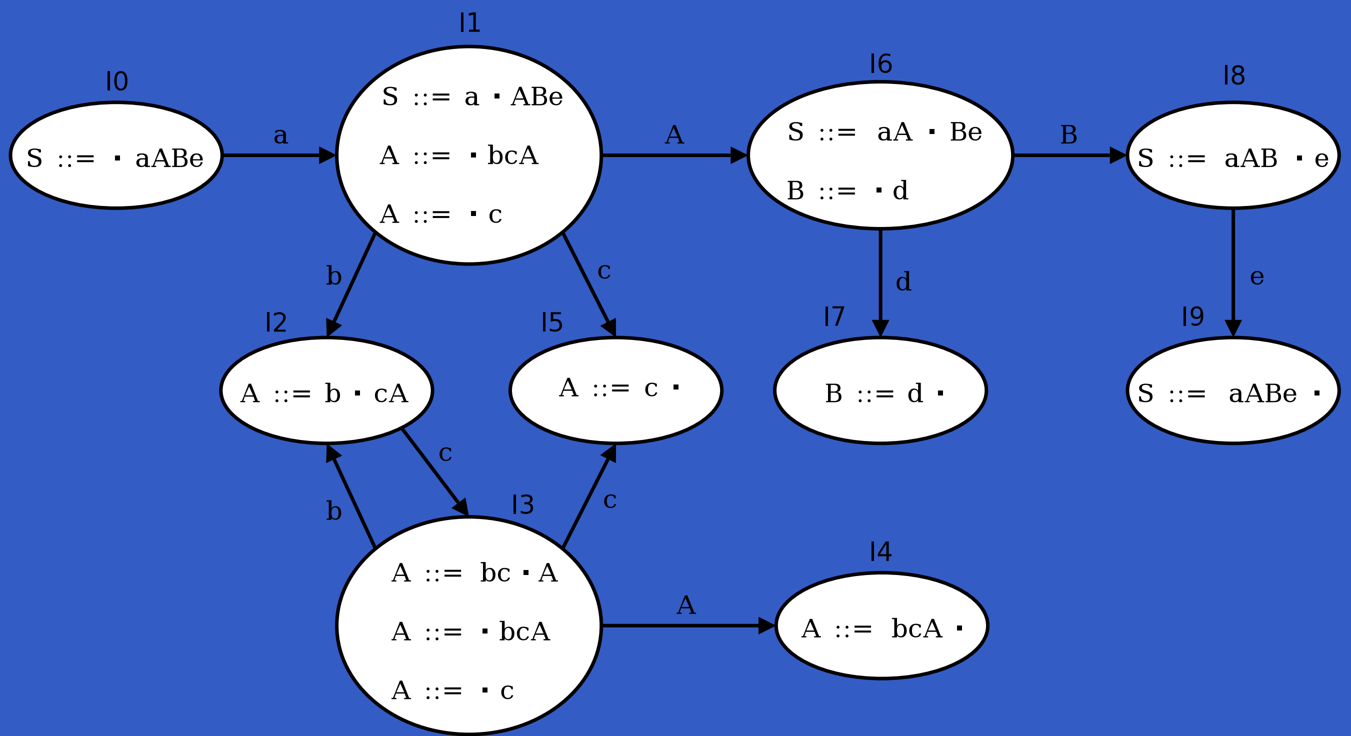
State	Stack (γ)	Input (w)	Move
I2	ab	$ccde$	Shift
I3	abc	cde	Shift
I5	$abcc$	de	Reduce by $A ::= c$

LR(0) Parsing (8)



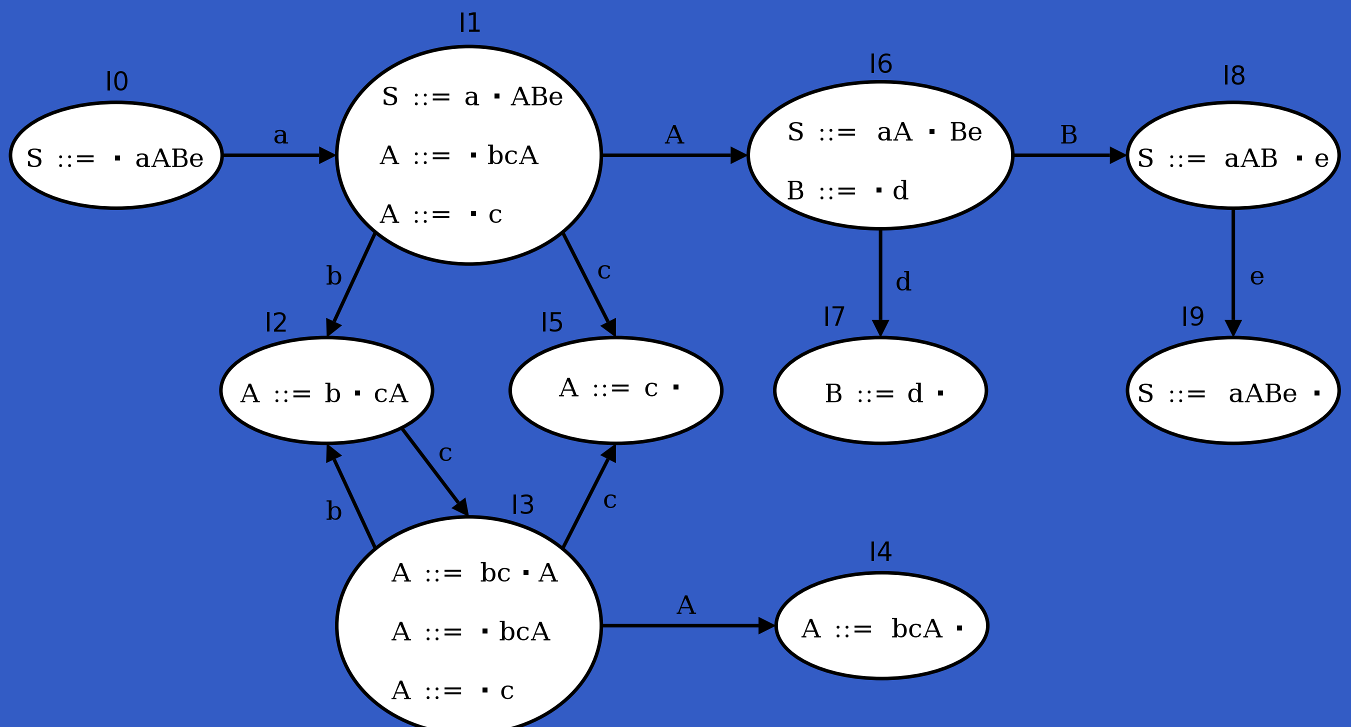
State	Stack (γ)	Input (w)	Move
I4	$abcA$	de	

LR(0) Parsing (8)



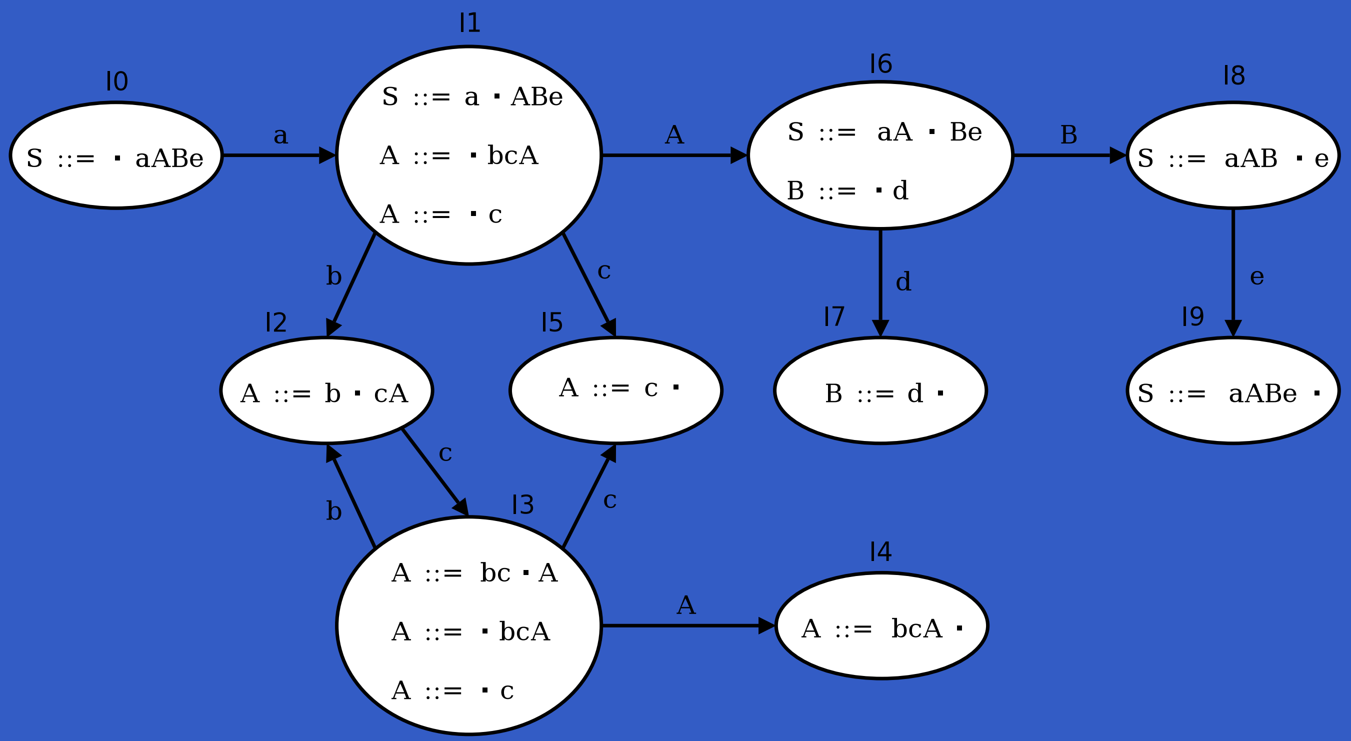
State	Stack (γ)	Input (w)	Move
I4	$abcA$	de	Reduce by $A ::= bcA$

LR(0) Parsing (8)



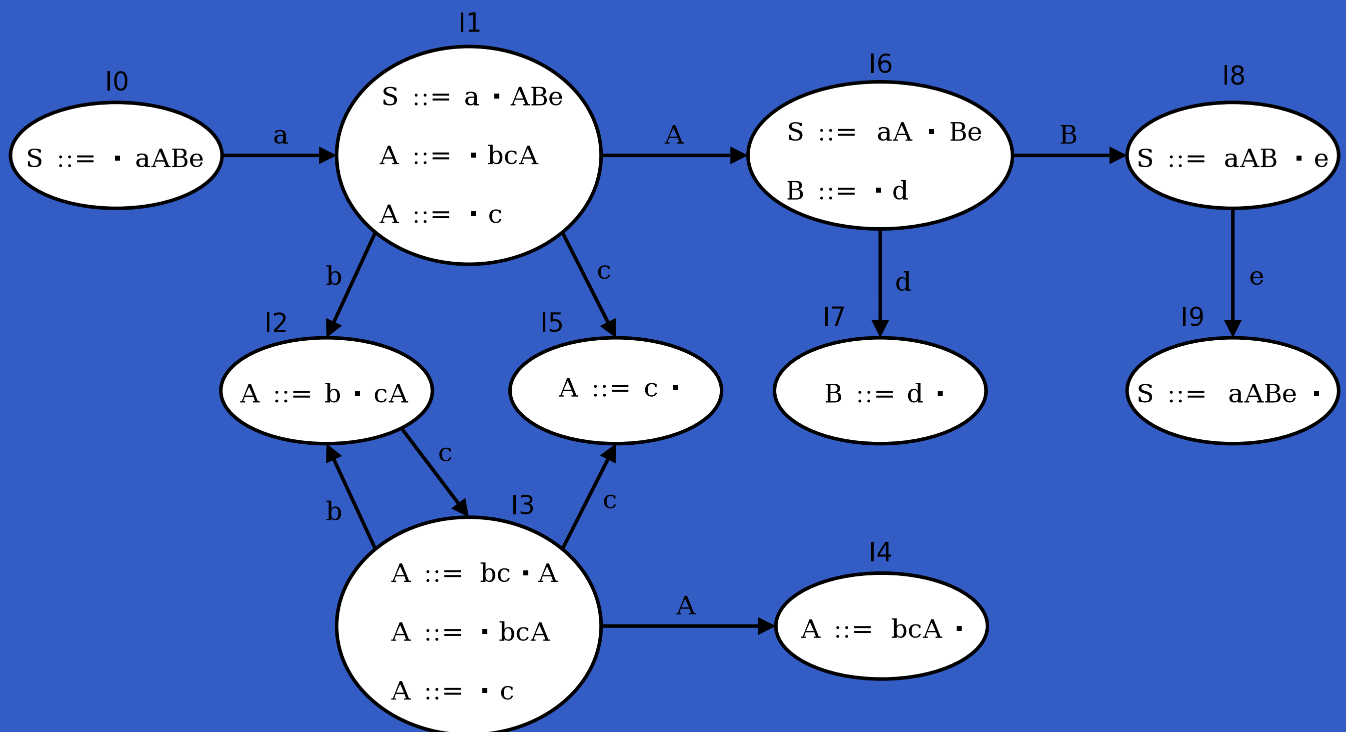
State	Stack (γ)	Input (w)	Move
I4	$abcA$	de	Reduce by $A ::= bcA$
I6	aA	de	

LR(0) Parsing (8)



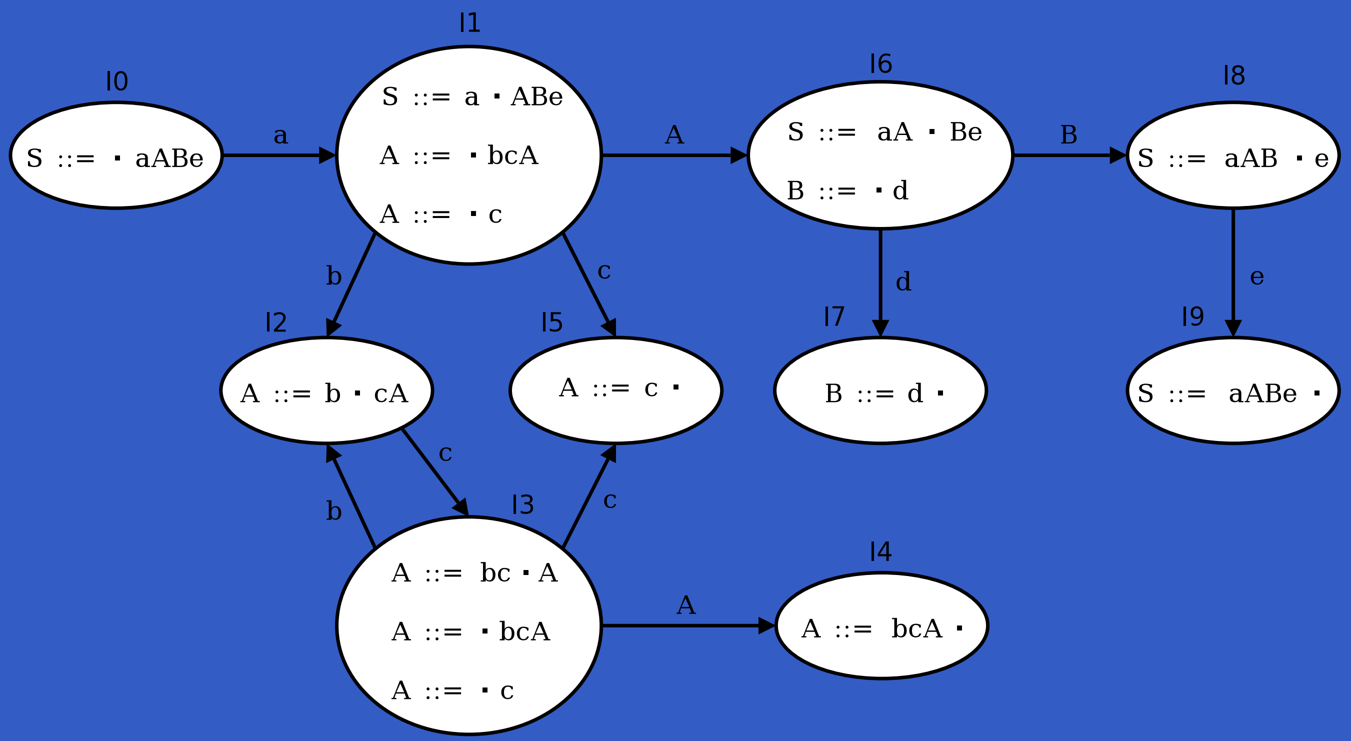
State	Stack (γ)	Input (w)	Move
I4	$abcA$	de	Reduce by $A ::= bcA$
I6	aA	de	Shift

LR(0) Parsing (8)



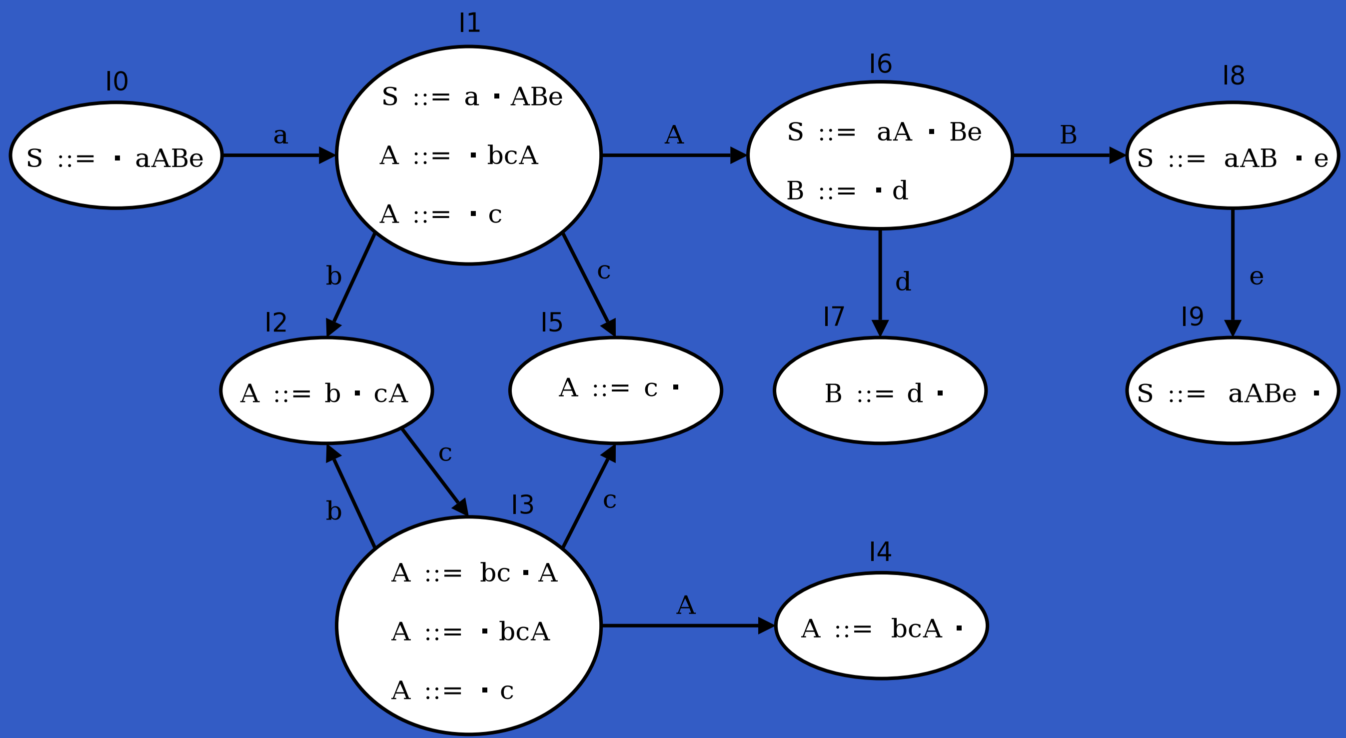
State	Stack (γ)	Input (w)	Move
I4	$abcA$	de	Reduce by $A ::= bcA$
I6	aA	de	Shift
I7	aAd	e	

LR(0) Parsing (8)



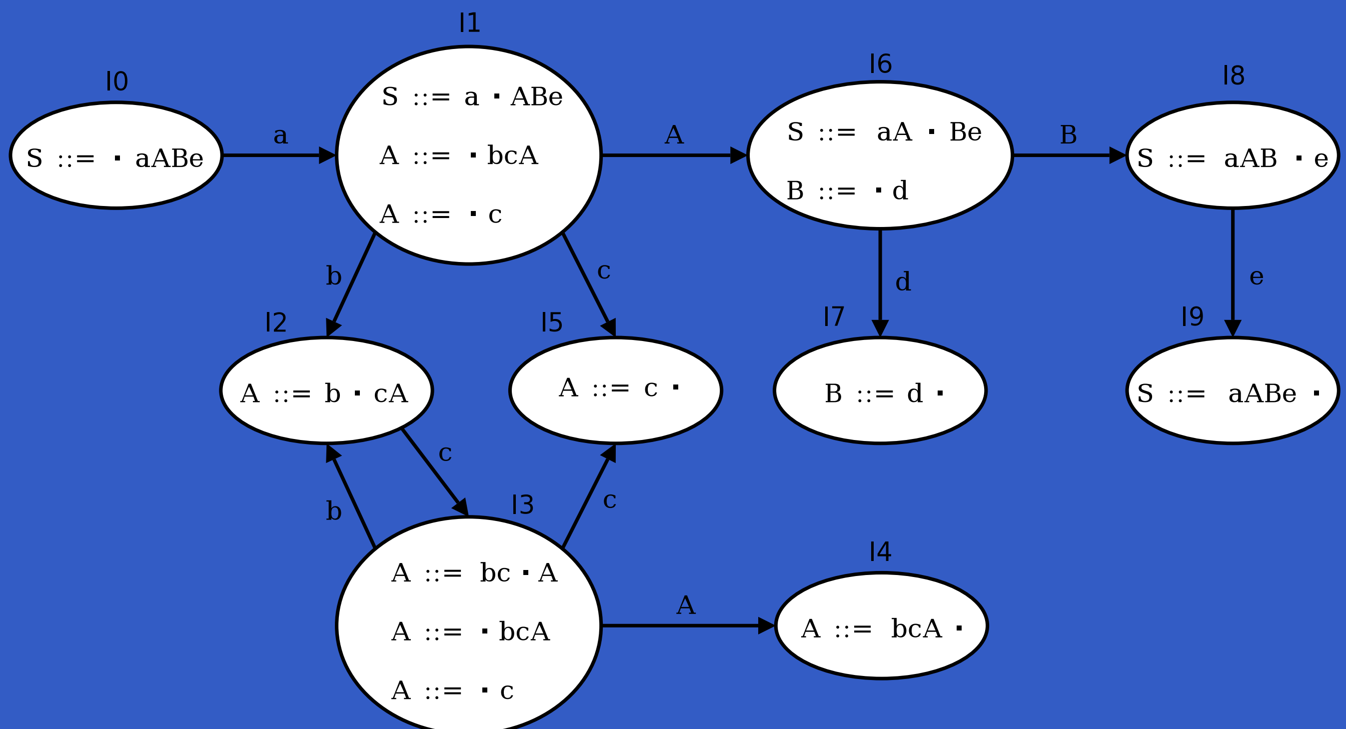
State	Stack (γ)	Input (w)	Move
I4	$abcA$	de	Reduce by $A ::= bcA$
I6	aA	de	Shift
I7	aAd	e	Reduce by $B ::= d$

LR(0) Parsing (9)



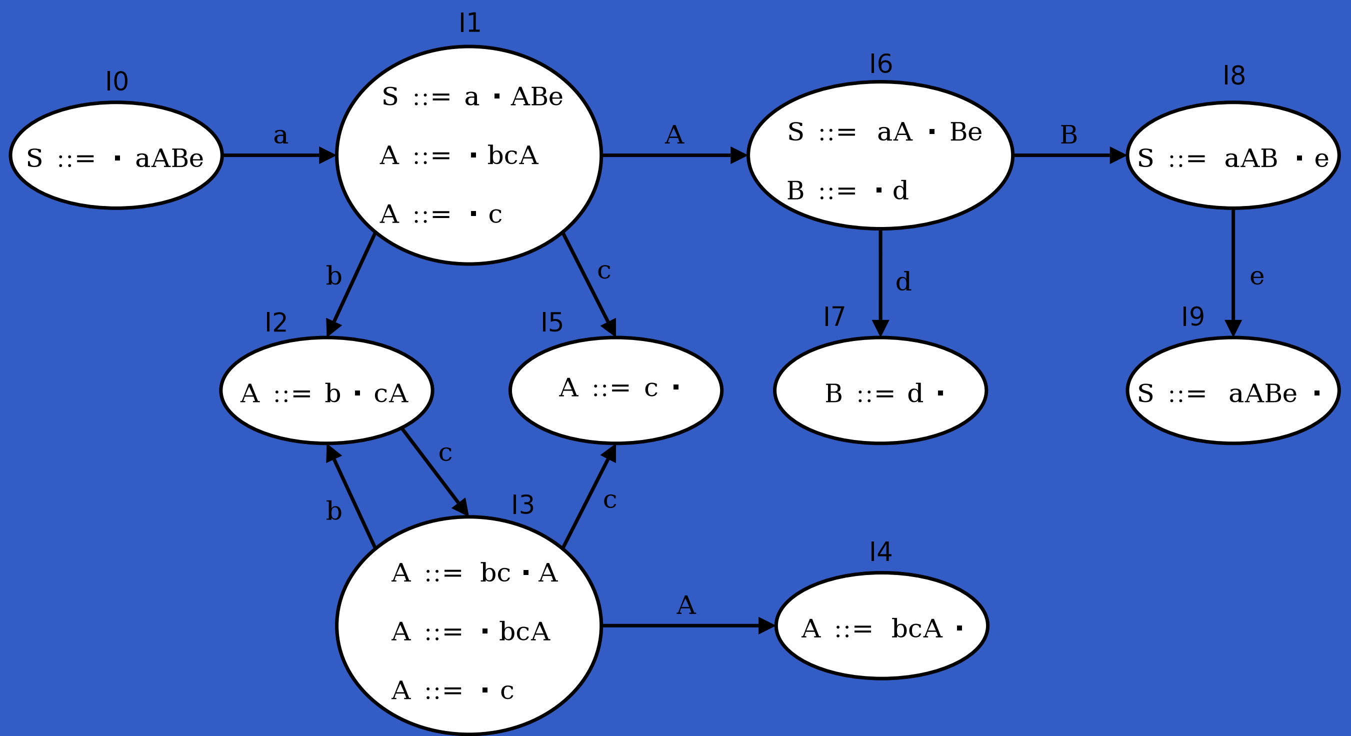
State	Stack (γ)	Input (w)	Move
I8	aAB	e	

LR(0) Parsing (9)



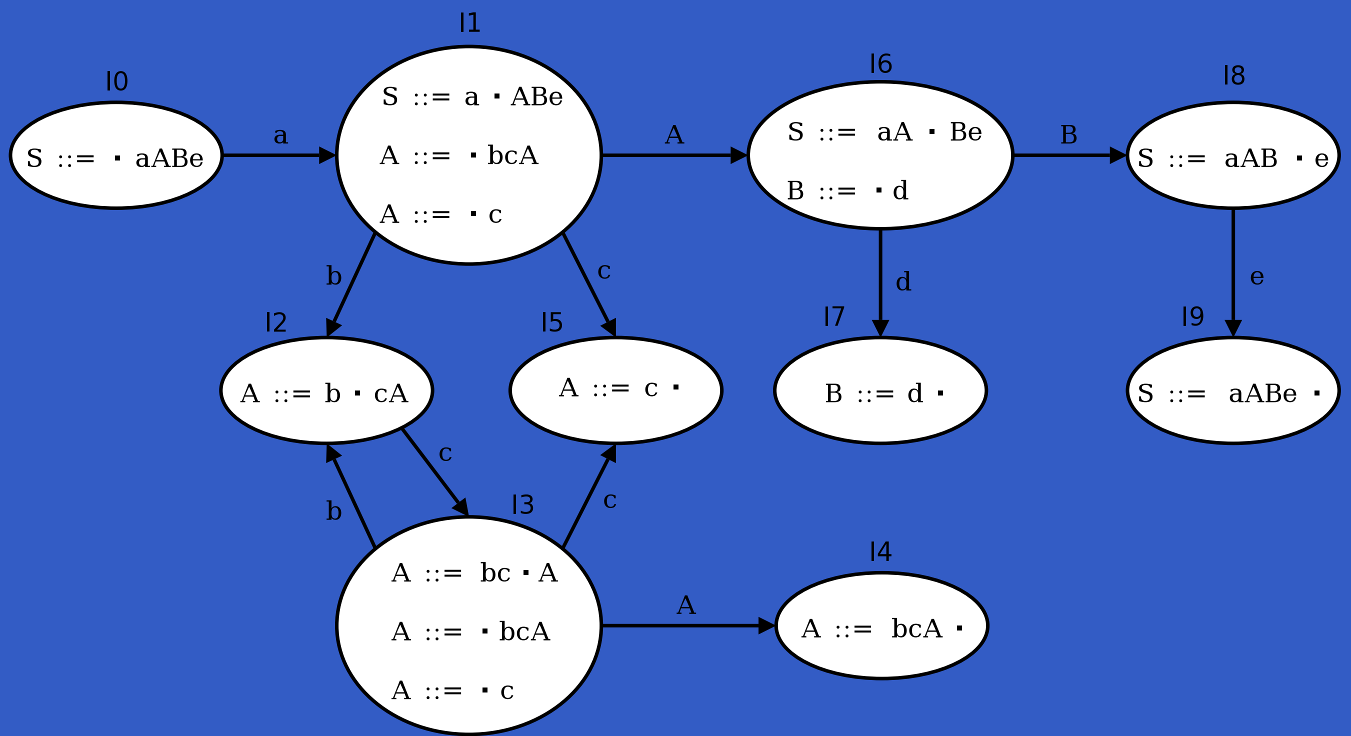
State	Stack (γ)	Input (w)	Move
I8	aAB	e	Shift

LR(0) Parsing (9)



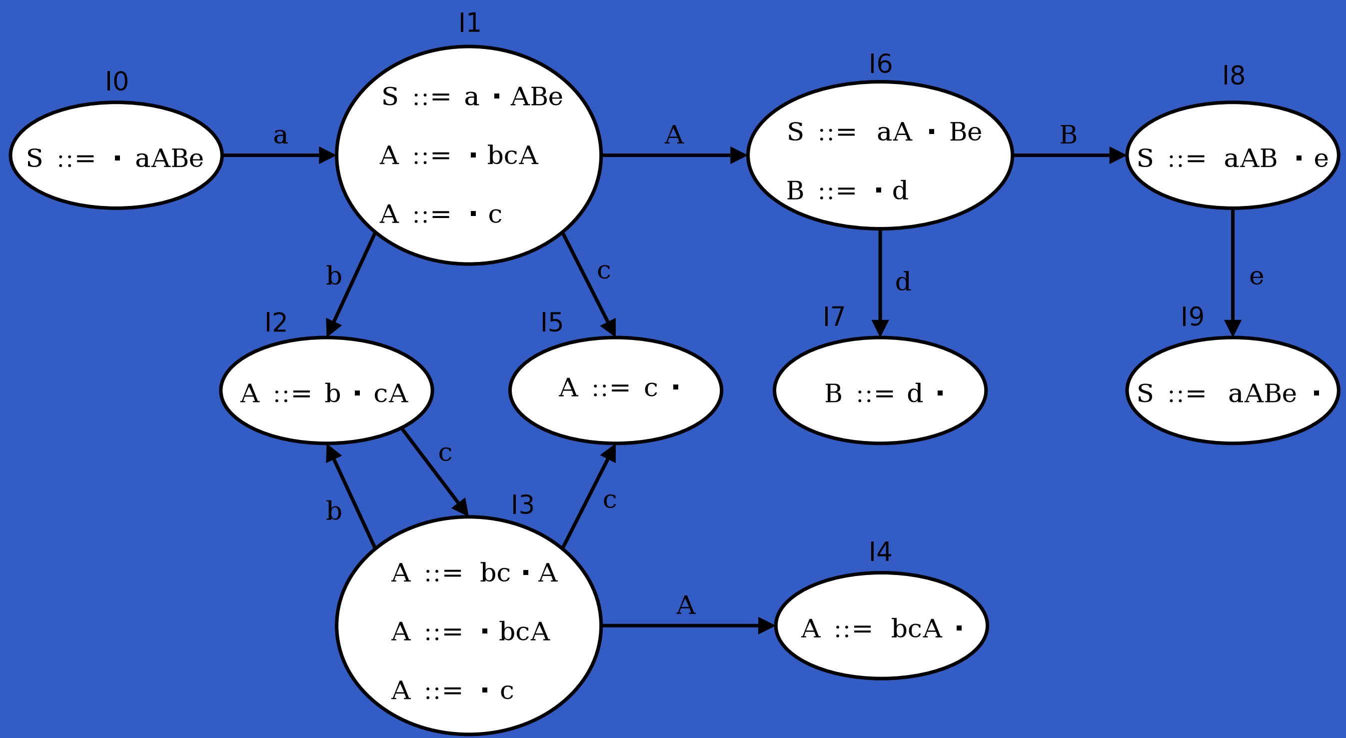
State	Stack (γ)	Input (w)	Move
I8	aAB	e	Shift
I9	$aABe$	ϵ	

LR(0) Parsing (9)



State	Stack (γ)	Input (w)	Move
I8	aAB	e	Shift
I9	$aABe$	ϵ	Reduce by $S ::= aABe$

LR(0) Parsing (9)



State	Stack (γ)	Input (w)	Move
I8	aAB	e	Shift
I9	$aABe$	ϵ	Reduce by $S ::= aABe$
	S	ϵ	Done

LR(0) Parsing (10)

Complete sequence (γw is right-sentential form):

State	Stack (γ)	Input (w)	Move
I0	ϵ	$abccde$	Shift
I1	a	$bccde$	Shift
I2	ab	$ccde$	Shift
I3	abc	cde	Shift
I5	$abcc$	de	Reduce by $A ::= c$
I4	$abcA$	de	Reduce by $A ::= bcA$
I6	aA	de	Shift
I7	aAd	e	Reduce by $B ::= d$
I8	aAB	e	Shift
I9	$aABe$	ϵ	Reduce by $S ::= aABe$
	S	ϵ	Done

Cf: $S \underset{rm}{\Rightarrow} aABe \underset{rm}{\Rightarrow} aAde \underset{rm}{\Rightarrow} abcAde \underset{rm}{\Rightarrow} abccde$

Parser Generators (1)

- Constructing parsers by hand can be very tedious and time consuming.

Parser Generators (1)

- Constructing parsers by hand can be very tedious and time consuming.
- This is true in particular for LR(k) and LALR parsers: constructing the corresponding DFAs is extremely laborious.

Parser Generators (1)

- Constructing parsers by hand can be very tedious and time consuming.
- This is true in particular for LR(k) and LALR parsers: constructing the corresponding DFAs is extremely laborious.
- E.g., our simple grammar

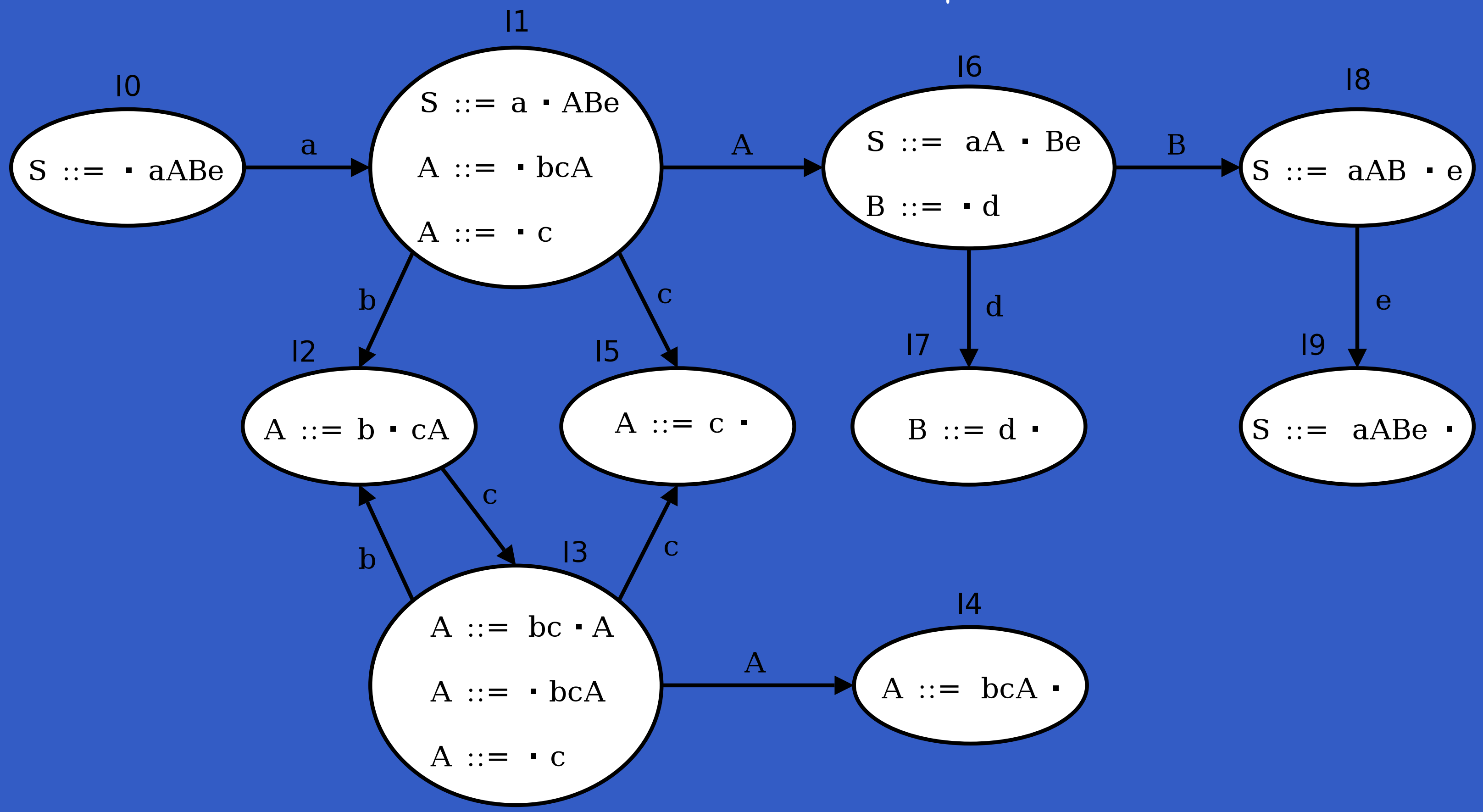
$$\begin{array}{rcl} S & ::= & aABe \\ A & ::= & bcA \mid c \\ B & ::= & d \end{array}$$

gives rise to a 10 state LR(0) DFA!

Parser Generators (2)

An LR(0) DFA recognizing viable prefixes for

$$S ::= aABe \qquad A ::= bcA \mid c \qquad B ::= d$$



Parser Generators (3)

- Parser construction is a mechanical process. Why not write a program to do the hard work for us?

Parser Generators (3)

- Parser construction is a mechanical process. Why not write a program to do the hard work for us?
- A ***Parser Generator*** (or “compiler compiler”) takes a grammar as input and outputs a parser (a program) for that grammar.

Parser Generators (3)

- Parser construction is a mechanical process. Why not write a program to do the hard work for us?
- A ***Parser Generator*** (or “compiler compiler”) takes a grammar as input and outputs a parser (a program) for that grammar.
- The input grammar is augmented with ***“semantic actions”***: code fragments that get invoked when a derivation step is performed.

Parser Generators (3)

- Parser construction is a mechanical process. Why not write a program to do the hard work for us?
- A ***Parser Generator*** (or “compiler compiler”) takes a grammar as input and outputs a parser (a program) for that grammar.
- The input grammar is augmented with ***“semantic actions”***: code fragments that get invoked when a derivation step is performed.
- The semantic actions typically construct an AST or interpret the program being parsed.

Parser Generators (4)

Some examples of parser generators:

Parser Generators (4)

Some examples of parser generators:

- Yacc (“Yet Another Compiler Compiler”): A classic UNIX LALR parser generator for C.
  `http://dinosaur.compilertools.net/`

Parser Generators (4)

Some examples of parser generators:

- Yacc ("Yet Another Compiler Compiler"): A classic UNIX LALR parser generator for C.
  `http://dinosaur.compilertools.net/`
- Bison: GNU project parser generator, a free Yacc replacement, for C and C++.

Parser Generators (4)

Some examples of parser generators:

- Yacc ("Yet Another Compiler Compiler"): A classic UNIX LALR parser generator for C.
  `http://dinosaur.compilertools.net/`
- Bison: GNU project parser generator, a free Yacc replacement, for C and C++.
- Happy: a parser generator for Haskell, similar to Yacc and Bison.
  `http://www.haskell.org/happy/`

Parser Generators (4)

Some examples of parser generators:

- Yacc (“Yet Another Compiler Compiler”): A classic UNIX LALR parser generator for C.
  `http://dinosaur.compilertools.net/`
- Bison: GNU project parser generator, a free Yacc replacement, for C and C++.
- Happy: a parser generator for Haskell, similar to Yacc and Bison.
  `http://www.haskell.org/happy/`
- Cup: LALR parser generator for Java.

Parser Generators (5)

- ANTLR: LL(k) (recursive descent) parser generator and other translator-oriented tools for Java, C#, C++. `http://www.antlr.org/`

Parser Generators (5)

- ANTLR: LL(k) (recursive descent) parser generator and other translator-oriented tools for Java, C#, C++. `http://www.antlr.org/`
- Many more compiler tools for Java here: `http://catalog.compilertools.net/java.html`

Parser Generators (5)

- ANTLR: LL(k) (recursive descent) parser generator and other translator-oriented tools for Java, C#, C++. `http://www.antlr.org/`
- Many more compiler tools for Java here: `http://catalog.compilertools.net/java.html`
- And a general catalogue of compiler tools: `http://catalog.compilertools.net/`

Happy Parser for TXL (1)

We are going to develop a TXL (the Trivial eXpression Language) using Happy. The TXL CFG:

txl-program	::=	*exp*
exp	::=	*add-exp*
add-exp	::=	*mul-exp*
	|	*add-exp* **+** *mul-exp*
	|	*add-exp* **-** *mul-exp*

Happy Parser for TXL (2)

The TXL CFG continued:

mul-exp	::=	*prim-exp*
	|	*mul-exp* **`*`** *prim-exp*
	|	*mul-exp* **`/`** *prim-exp*
prim-exp	::=	**`INTEGER`**
	|	**`IDENTIFIER`**
	|	**`(`** *exp* **`)`**
	|	**`let`** **`IDENTIFIER`** **`=`** *exp* **`in`** *exp*

Happy Parser for TXL (3)

Haskell datatype for tokens:

```
data Token = T_Int Int
           | T_Id Id
           | T_Plus
           | T_Minus
           | T_Times
           | T_Divide
           | T_LeftPar
           | T_RightPar
           | T_Equal
           | T_Let
           | T_In
```

Happy Parser for TXL (4)

Haskell datatypes for AST:

```
data BinOp = Plus | Minus | Times | Divide

data Exp = LitInt   Int
         | Var      Id
         | BinOpApp BinOp Exp Exp
         | Let      Id Exp Exp
```

Happy Parser for TXL (5)

A simple Happy input file looks like follows:

```
{ Module Header }
%name ParserFunctionName
%tokentype { TokenTypeName }

%token
Specification of Terminal Symbols
%%
Grammar productions with semantic actions

{ Further Haskell Code }
```

Happy Parser for TXL (6)

The terminal symbol specification specifies terminals to be used in productions and relates them to Haskell constructors for the tokens:

```
%token
    int       { T_Int $$ }
    ident     { T_Id $$ }
    '+'       { T_Plus }
    '-'       { T_Minus }
    ...
    '='       { T_Equal }
    let       { T_Let }
    in        { T_In }
```

Happy Parser for TXL (5)

The grammar productions are written in BNF, with an additional semantic action defining the return value for each production:

```
add_exp
   : mul_exp                     {$1}
   | add_exp '+' mul_exp {BinOpApp Plus $1 $3}
   | add_exp '-' mul_exp {BinOpApp Minus $1 $3}
mul_exp
   : prim_exp                     {$1}
   | mul_exp '*' prim_exp {BinOpApp Times $1 $3}
   | mul_exp '/' prim_exp {BinOpApp Divide $1 $3}
```

Precedence and Associativity

Happy (like e.g. Yacc and Bison) allows operator precedence and associativity to be explicitly specified to disambiguate a grammar:

```
%left '+' '-'
%left '*' '/'
exp : exp '+' exp { BinOpApp Plus $1 $3 }
    | exp '-' exp { BinOpApp Minus $1 $3 }
    | exp '*' exp { BinOpApp Times $1 $3 }
    | exp '/' exp { BinOpApp Divide $1 $3 }
    ...
```