LR (Bottom-up) Parsing:

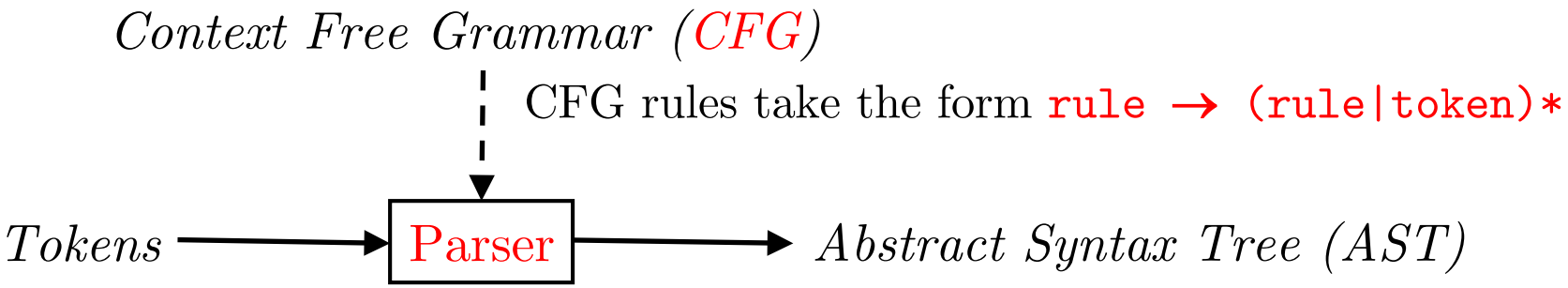
Tokens to AST Tree

Naranker Dulay
n.dulay@imperial.ac.uk

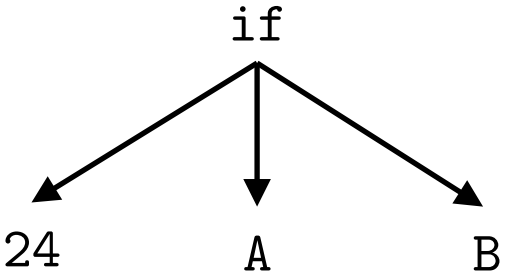
https://www.doc.ic.ac.uk/~nd/compilers

Parsing - main idea

Parsing (syntax analysis) transforms a stream of tokens into an abstract syntax tree based on the (typically context free) grammar of the language




```
IF
LPAREN
INT(24)
RPAREN
IDENT('A')
ELSE
IDENT('B')
```

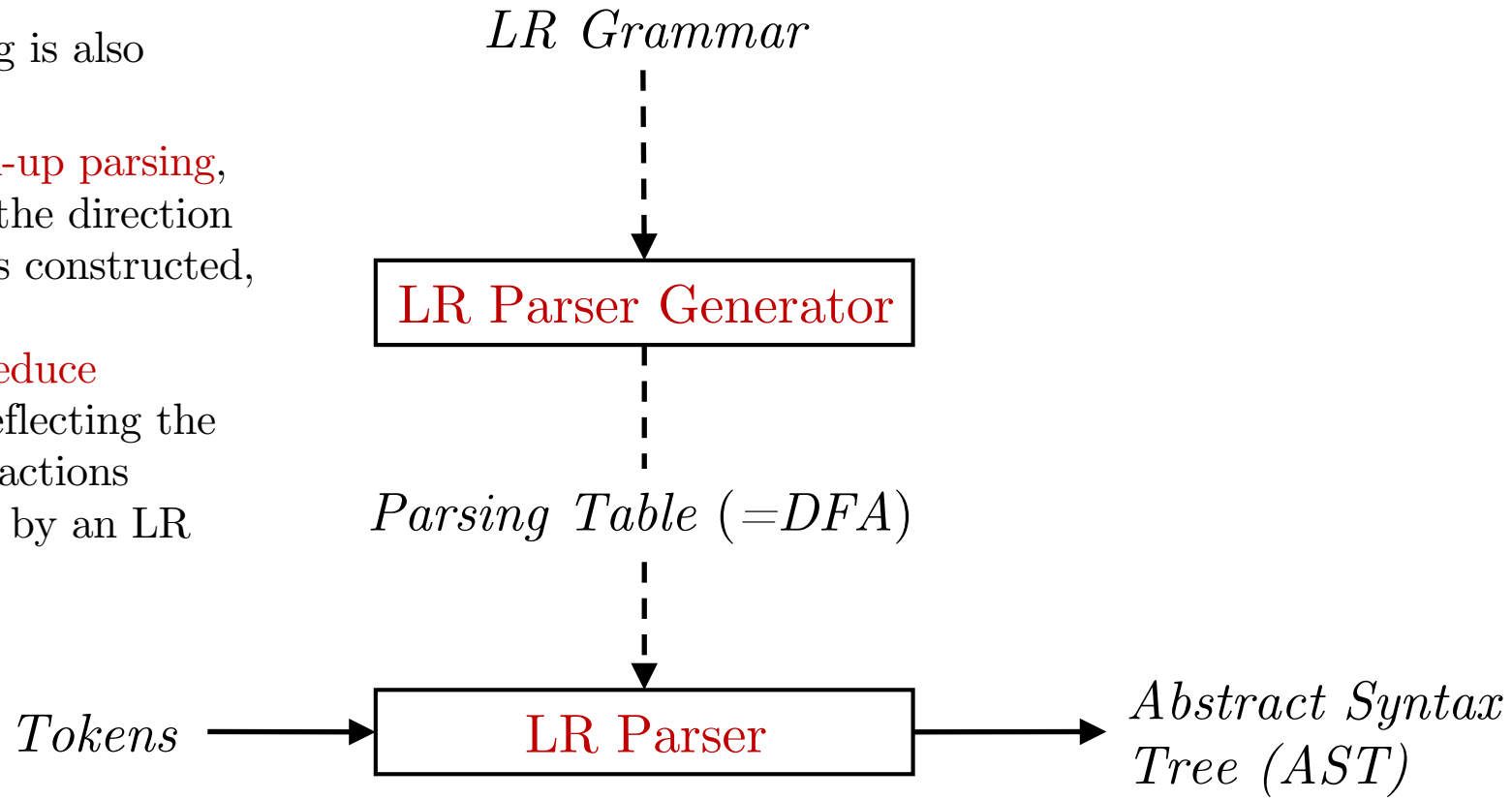


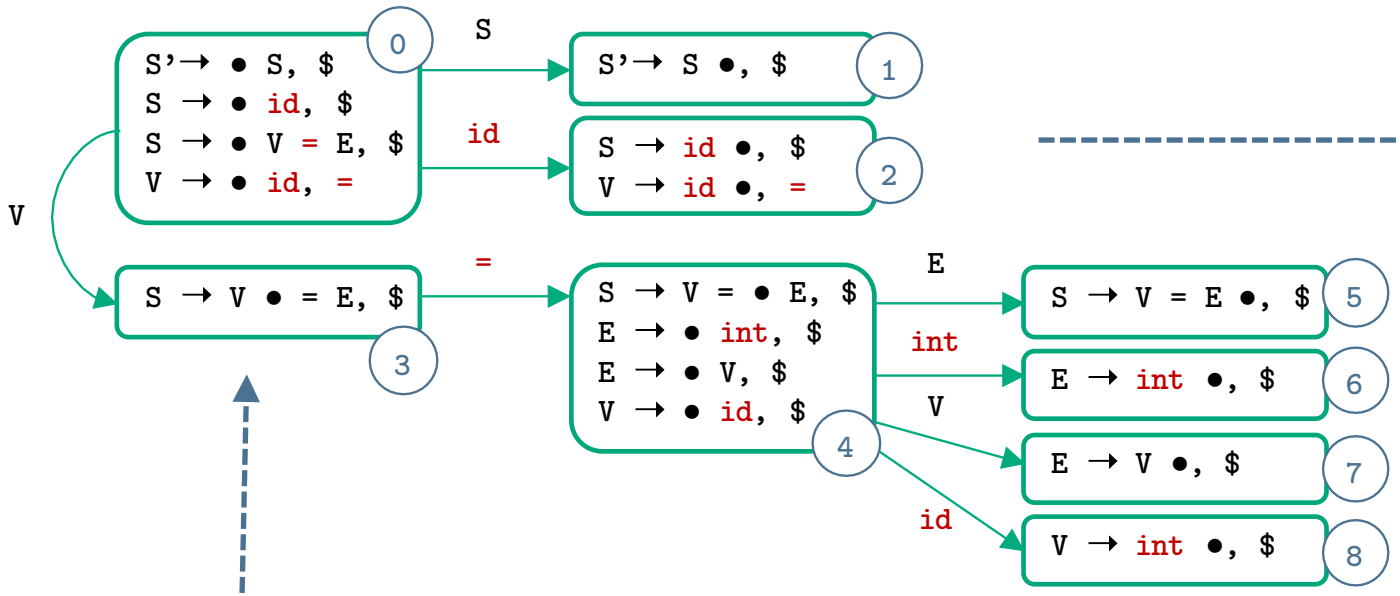
LR Parsing - main idea

LR parsing is also known as:

(i) bottom-up parsing, reflecting the direction the AST is constructed,

(ii) shift-reduce parsing, reflecting the two main actions performed by an LR parser.





DFA to Parsing Table

Grammar to DFA

r1: S → *id*
r2: S → V '=' E
r3: V → *id*
r4: E → V
r5: E → *int*

State z	ACTION				GOTO		
	id	*int*	=	$	E	V	S
0	S2					G3	G1
1				A			
2			R3	R1			
3			S4				
4	S8	S6			G5	G7	
5				R2			
6				R5			
7				R4			
8				R3			

Parsers for Context-Free Grammars

LL and LR grammars are subsets of CFGs that we can parse in O(n) time instead of $O(n^3)$ required for general CFGs.

LL - Top-Down Parsers

- Build AST from root to leaves.

LR - Bottom-Up Parsers

- Build AST from leaves to root.

LL(k):	Left-to-right scan, Leftmost-derivation, k-token look-ahead
LL(0)	No such thing.
LL(1)	Most popular LL. Can implement as an automaton or as a recursive descent parser
k = 2	Rarely Sometimes useful
k>=2	Rarely needed

LR(k):	Left-to-right scan, Rightmost-derivation (in reverse), k-token look-ahead
LR(0)	Weak. Useful for learning about LR.
SLR(1)	Stronger, superseded by LALR(1)
LALR(1)	Fast and popular. Close in power to LR(1)
LR(1)	Powerful -- needs more memory
k => 2	Possible, but rarely needed/used

LR(n) is a superset of an LL(n) and hence more powerful.

Chomsky Hierarchy

In the following

- **R** is a non-terminal (the name of a rule/production, non-leaf node in AST)
- **t** is a sequence of terminals (tokens from lexical analyser, leaf nodes in AST)
- **α**, **β**, **φ** are sequences of terminals and non-terminals

---

type 3 – **regular grammars** (Deterministic Finite Automata)

$$R \rightarrow t$$

type 2 – **context free grammars** (Pushdown Automata)

$$R \rightarrow \alpha$$

type 1 – **context sensitive grammars** (Linear Bounded Automata)

$$\alpha\ R\ \beta \rightarrow \alpha\ \varphi\ \beta$$

type 0 – **unrestricted grammars** (Turing Machines)

$$\alpha \rightarrow \beta$$

LR Parsers

Most programming languages can be described by a Context-Free Grammar (CFG) and virtually all CFGs for programming languages can be described by an LR grammar and implemented with an LR parser.

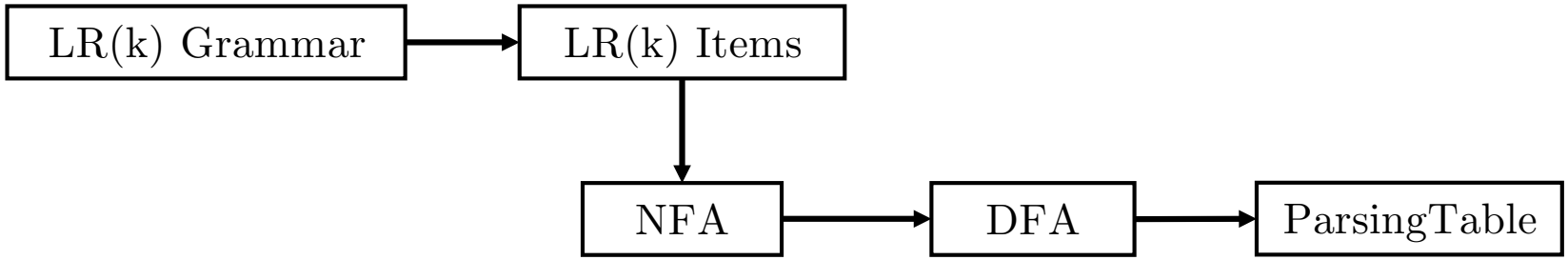
LR Parsers are too complex to build by hand however. We need an LR parser-generator e.g. yacc/bison, SableCC, CUP, Beaver. The principles of LR operation are "relatively easy" to understand however.

LR Parsers can be implemented efficiently in O(n) time.

LR Grammar to Parsing Table

There are several methods for generating LR parsing tables. The methods essentially define parsers/grammars of different expressive power, e.g. LR(0), SLR(1), LALR(1), LR(1), LR(2), LR(3) etc. Less powerful methods may fail to produce a parsing table for a given context-free grammar.

The parsers resulting from these methods are similar - in each case their states consist of LR(k) items of the grammar, but vary in the number of states and/or in the size of their lookahead sets.



LR(0) Items

LR(0) parsers are the simplest LR parsers. They don't use the current token in order to perform a reduce.

An LR(0) item is a "rule" of the grammar with a dot • in some position on the right hand side of the rule.

An item indicates how much of a rule has been seen, e.g. the item `E` → `E + •  int` indicates that we have seen an expression and a plus sign and hope to encounter an integer next. LR(0) items thus indicate intermediate steps in the recognition of the RHS of a rule.

Example: The rule `X` → `ABC` has 4 LR(0) items

`X` → • `ABC`	Initial item
`X` → `A` • `BC`	
`X` → `AB` • `C`	
`X` → `ABC` •	Complete Item

Example

The Grammar

For LR, we always add a start rule with an end-of-input symbol $

E' → E $

rule r1: E → E '+' *int* -- label rules for parsing table

rule r2: E → *int*

has 8 LR(0) items:

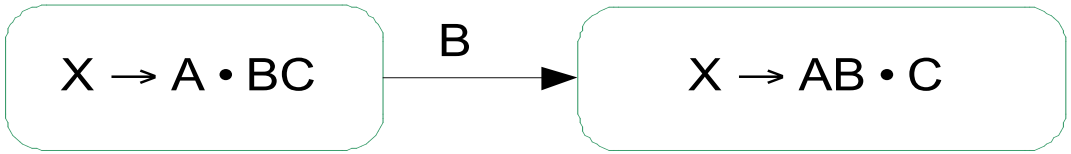
E' → • E	Initial item
E' → E •	Complete Item
E → • E + *int*	Initial item
E → E • + *int*	
E → E + • *int*	
E → E + *int* •	Complete Item
E → • *int*	
E → *int* •	Complete Item

LR(0) to NFA

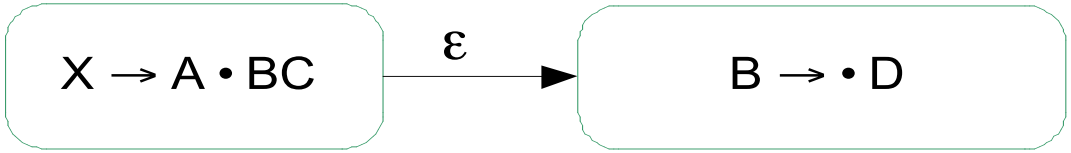
LR(0) items are used as states of a finite automaton that maintain information about the progress of a *shift-reduce parse*.

We build a NFA from the LR(0) items and from this NFA we can build a DFA using the subset construction.

NFA Transitions: Given an item `X → A • BC`, add the following transition



In addition, if `B` is a non-terminal then forall `B → • D` add ε transitions of the form:



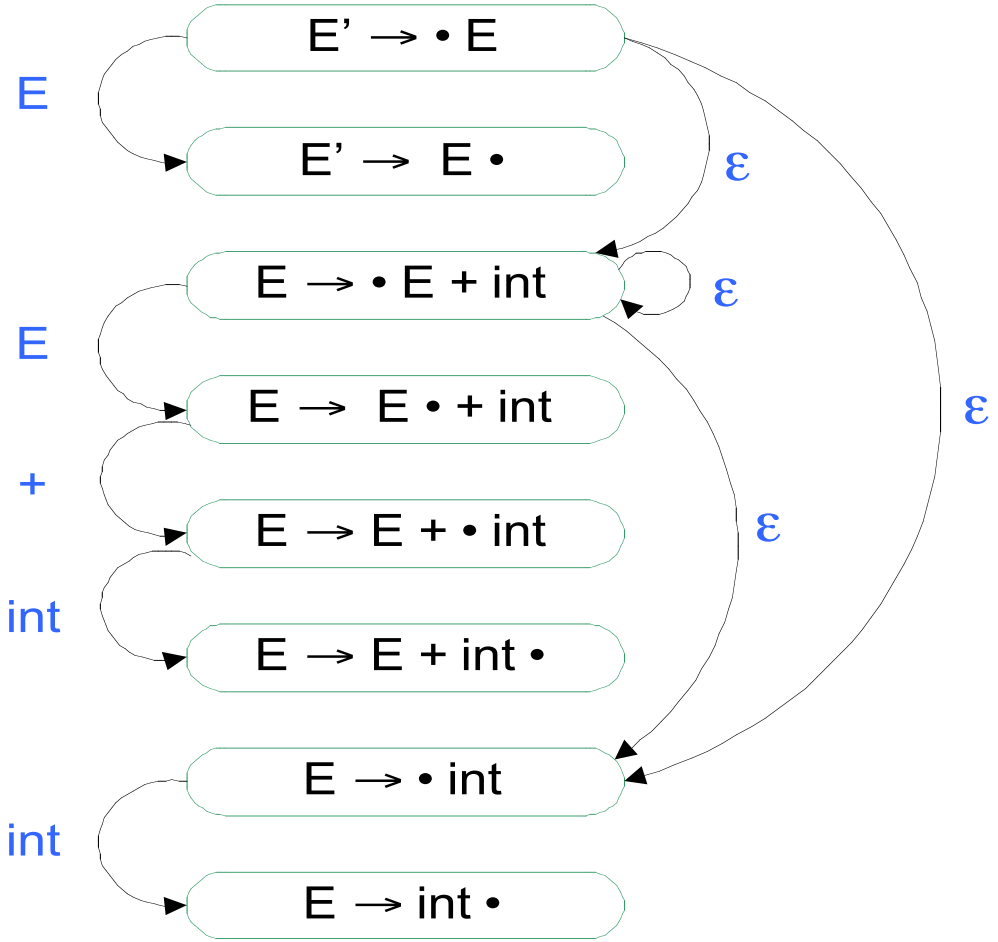
Example

Grammar

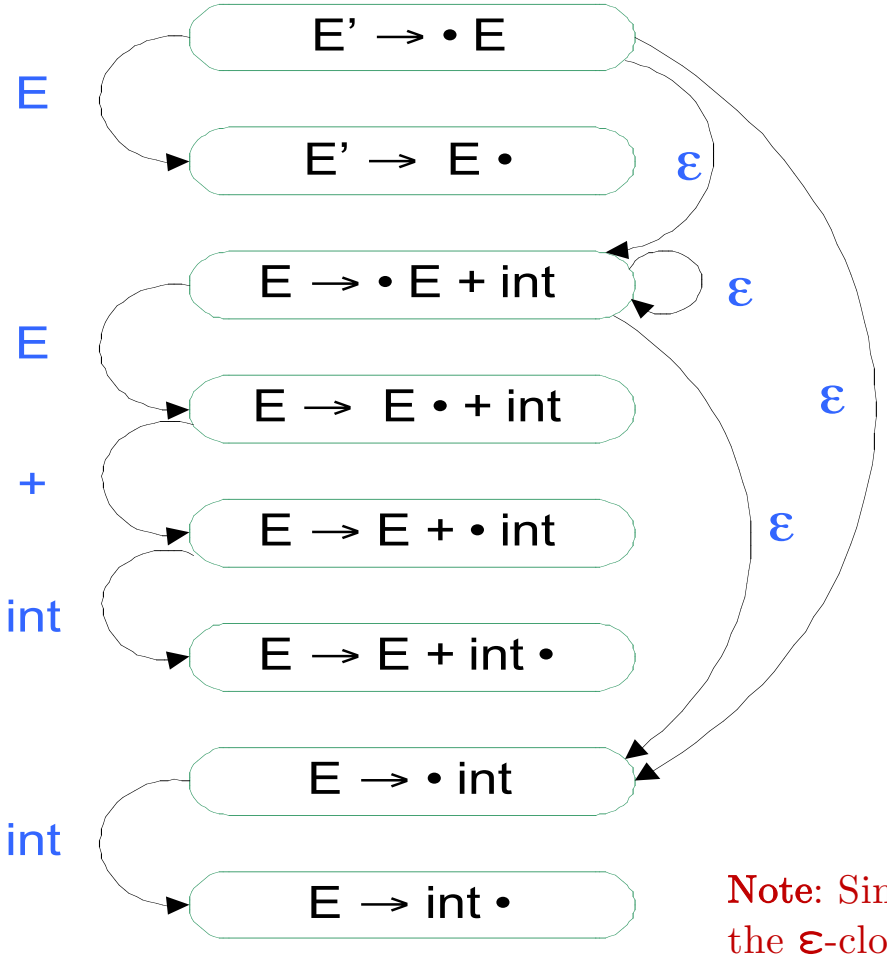
```
     E' → E $
r1:  E → E '+' int
r2:  E → int
```

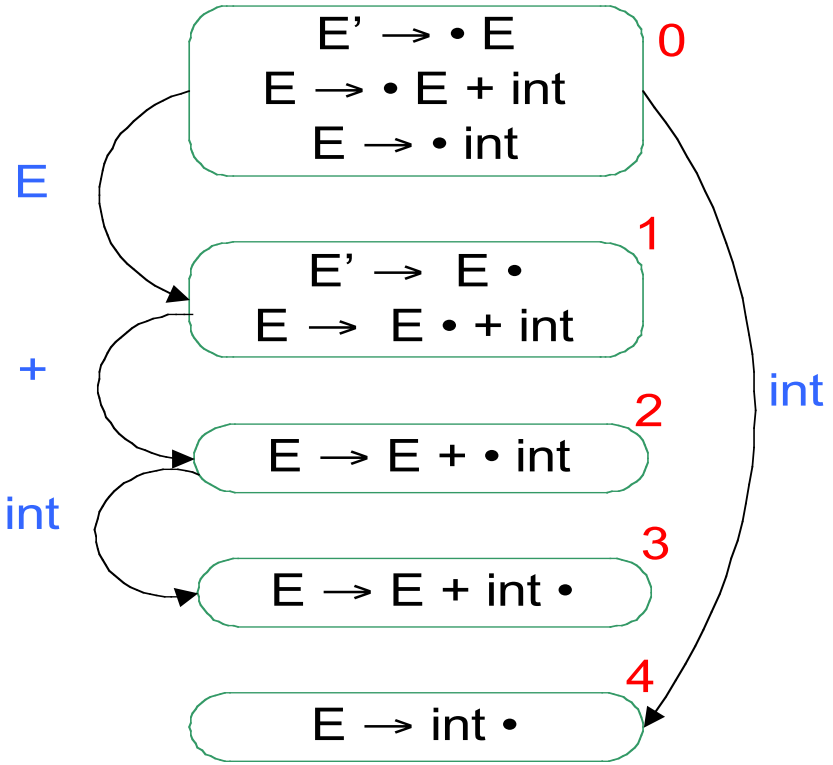
Items

```
E' → • E $
E' → E •
E → • E + int
E → E • + int
E → E + • int
E → E + int •
E → • int
E → int •
```



NFA to DFA





Note: Since its easy to determine the ε-closures from items, we'll construct the DFA directly.

Number states from 0

DFA to LR(0) Parsing Table

For each terminal transition X $\xrightarrow{T}$ Y add P [X, T] = sY (shift Y)

For each non-terminal transition X $\xrightarrow{N}$ Y add P [X, N] = gY (goto Y)

For each state X containing the item R' → .. • add P [X, $] = a (accept)

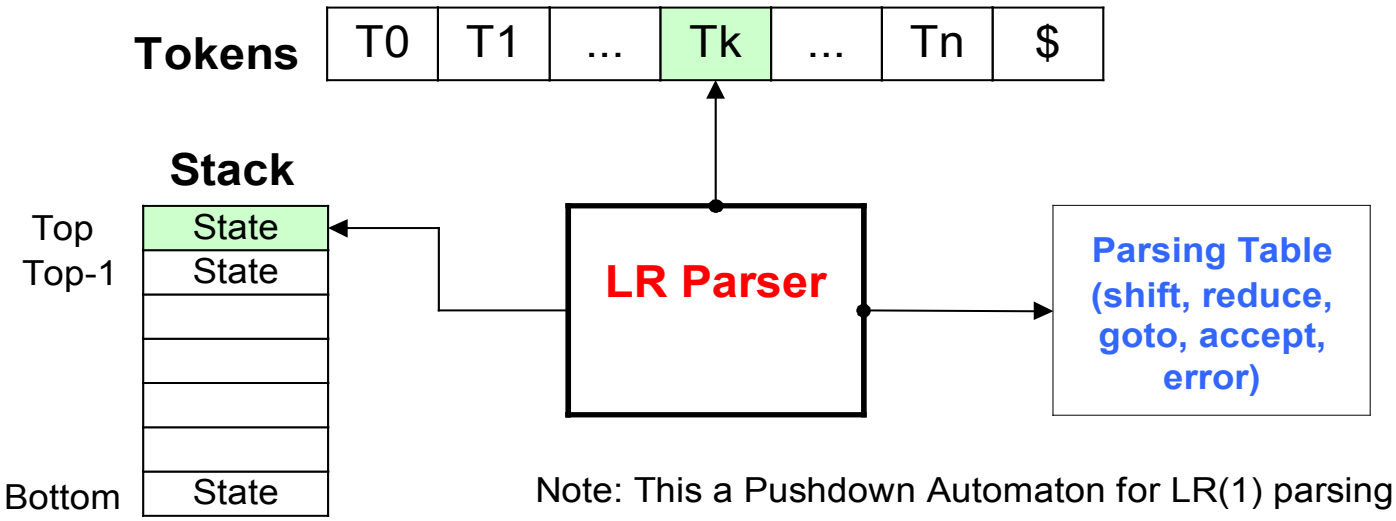
For each state X containing an item R → .. • add P [X, T] = rN (reduce) for every terminal T. Note: N is number for rule R.

Note:

For LR(0) parsers if there is more than **one action** for a table row then the grammar is not LR(0), e.g. we may have a shift and a reduce (**shift-reduce** conflict) or two reduces (**reduce-reduce** conflict). Blank cells indicate an error.

State	ACTION			GOTO
	int	+	$	E
0	s4			g1
1		s2	a	
2	s3			
3	r1	r1	r1	
4	r2	r2	r2	

Model of an LR parser



Operation: We push state 0 onto the stack then repeatedly perform:

```
SWITCH ParsingTable[Stack[Top],CurrentToken]
   shift  Sn:  Push state n onto the Stack, Advance Current Token
   reduce Rn:  Remove L elements from the Stack where L=length of RHS of rule n.
               Push ParsingTable[Stack[Top],LHS of R]; i.e. the "Goto" action
               reduce also generates an AST node for the rule
   accept a:   Accept input stream (i.e. parse was successful)
   error:      Report Error
   goto   Gn:  This case is not selected directly. Looked-up in the reduce case
               above.
```

Example

Grammar E' → E $

For LR, always add a start rule with an end-of-input symbol $

rule r1: E → E '+' *int* -- label rules for parsing table
rule r2: E → *int*

Parsing Table

STATE	ACTION (Terminals)			GOTO (Non-Terminals)
	int	+	$	E
0	s2			g1
1		s3	a	
2		r2	r2	
3	s4			
4		r1	r1	

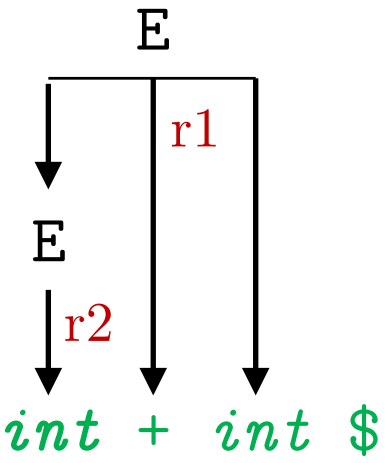
Abbreviations: s (shift), r (reduce), a (accept), g (goto), blank (error)
Note: s and g are followed by a state number, r is followed by a rule number.

Example Contd

E' → E $
r1: E → E '+' *int*
r2: E → *int*

State	ACTION			GOTO
	int	+	$	E
0	s2			g1
1		s3	a	
2		r2	r2	
3	s4			
4		r1	r1	

Stack	Tokens	Action
0	*int* + *int* $	T[0,int] = s2
0 2	+ *int* $	T[2,+] = r2, pop 1 elem, push T[0,E]
0 1	+ *int* $	T[1,+] = s3
0 1 3	*int* $	T[3,int] = s4
0 1 3 4	$	T[4,$] = r1, pop 3 elems, push T[0,E]
0 1	$	T[1,$] = a



SLR(1) Optional Material

SLR(1) parsers are similar to LR(0) parsers except that we only add reduce actions into a SLR(1) parsing table "if appropriate":

SLR(1): Add a reduce action for an item `A` $\rightarrow$ `B` $\bullet$ only if the current token can FOLLOW `A` somewhere in the grammar.

An SLR(1) parsing table has the same states and gotos as an LR(0) parsing table, only the actions differ.

This change increases the expressiveness of SLR(1) parsers over LR(0) parsers considerably and many programming language constructs can be parsed by SLR(1) that LR(0) cannot parse. We won't look at SLR parsers in any further detail but rather consider the more powerful LR(1) and LALR(1) parsers.

FIRST and FOLLOW Sets

The **FIRST set** for a sequence of rules (non-terminals) and tokens (terminals) α, is the set of all tokens that could start a derivation of α, plus ε if α could derive ε.

$$\texttt{FIRST}(\alpha) = \{t : \alpha \Rightarrow^* t\beta\} \cup (\texttt{if } \alpha \Rightarrow^* \varepsilon \texttt{ then } \{\varepsilon\} \texttt{ else } \{\})$$

$\Rightarrow^*$ zero or more derivations

$\Rightarrow^+$ one or more derivations

We say that a non-terminal `A` is **NULLABLE** if $\texttt{A} \Rightarrow^* \varepsilon$

The **FOLLOW set** for a rule (non-terminal) `A`, is the set of all tokens that could immediately follow `A`, plus \$ if `A` can end the input.

$$\texttt{FOLLOW(A)} = \{t : S \Rightarrow^+ \alpha A t \beta\} \cup (\texttt{if } S \Rightarrow^* \alpha A \texttt{ then } \{\$\} \texttt{ else } \{\})$$

where `S` is the start symbol

FIRST Set

The **FIRST set** for a token `T` is {`T`}, for ε is {ε}

The **FIRST set** for a rule `A`, is the set of all tokens that could start a derivation of `A`, plus ε if `A` could derive ε

foreach alternative of A of the form A $\rightarrow \beta_1\ \beta_2\ \ldots\ \beta_n$

```
        Include FIRST(β1) - {ε} in FIRST(A)
        If ε in FIRST(β1)
             Include FIRST (β2) - {ε} in FIRST(A)
             If  ε in FIRST(β2)
                   ...
                   Include FIRST (βn) - {ε} in FIRST(A)
                   If  ε in FIRST(βn)
                         Include ε in FIRST(A)
```

The **FIRST set** for a sequence `ABC..` is formed as for the rule case above.

FIRST Set Example

```
Expr'   → Expr $
Expr    → Term Expr2
Expr2   → '+' Term Expr2 | ε
Term    → Factor Term2
Term2   → '*' Factor Term2 | ε
Factor  → '(' Expr ')' | id
```

```
FIRST(Factor) = { (, id }
FIRST(Term2)  = { *, ε }
FIRST(Term)   = FIRST(Factor) = { (, id }
FIRST(Expr2)  = { +, ε }
FIRST(Expr)   = FIRST(Term) = { (, id }
```

FOLLOW Sets

The **FOLLOW set** for a rule `A`, is the set of all tokens that could immediately follow `A`. Note: ε is never in the FOLLOW set.

(i) `Foreach derivation of the form B → C A D`
`Include FIRST(D) - {ε} in FOLLOW(A)`
`if D ⇒* ε include FOLLOW(B) in FOLLOW(A)`

This last `if` can be written
`if {ε} in FIRST(D) include FOLLOW(B) in FOLLOW (A)`

Note: `C` & `D` can be empty. If `D` is empty then we also include `FOLLOW(B)` in `FOLLOW(A)`

(ii) If rule `A` can end the input we include $ in `FOLLOW(A)`
Recall we use `$` as an End-of-Input token.

FOLLOW Set Example

```
Expr'  →  Expr $
Expr   →  Term Expr2
Expr2  →  '+' Term Expr2 | ε
Term   →  Factor Term2
Term2  →  '*' Factor Term2 | ε
Factor →  '(' Expr ')' | id
```

```
FOLLOW(Expr)    = { ), $ }
FOLLOW(Expr2)   = FOLLOW(Expr) + ~~FOLLOW(Expr2)~~  = { ), $ }
FOLLOW(Term)    = FIRST(Expr2) + FOLLOW(Expr) + FOLLOW(Expr2)
                = { +, ), $ }
FOLLOW(Term2)   = FOLLOW(Term) + ~~FOLLOW(Term2)~~ = { +, ), $ }
FOLLOW(Factor)  = FIRST(Term2) + FOLLOW(Term) + FOLLOW(Term2)
                = { *,  +, ), $ }
```

LR(1) and LALR(1) Parsers

LR parsers essentially differ in how they perform reduction:

LR(0)	A reduce item `X` → `A•` always causes a reduction (the current token is not used in LR(0) parsing)
SLR(1)	A reduce item `X` → `A•` causes a reduction only if the current token is in FOLLOW(X)
LR(1)	A reduce item `X` → `A•, t` causes a reduction only if the current token is equal to the look-ahead token `t`
LALR(1)	Like LR(1) but we combine LR(1) states that differ in the look-ahead token of the item only.

LR(1) is the most powerful of these methods and sufficient for most programming languages. Its disadvantage is that it requires large parsing tables. LALR(1) is a modification of LR(1) which retains virtually all the benefits of LR(1), while producing much smaller parsing tables. We'll look at LR(1) first, then LALR(1).

LR(1) Items

An **LR(1) item** is a pair `[ LR(0)item, t ]`

(i) a `LR(0) item` as before, plus

(ii) a look-ahead token `t`

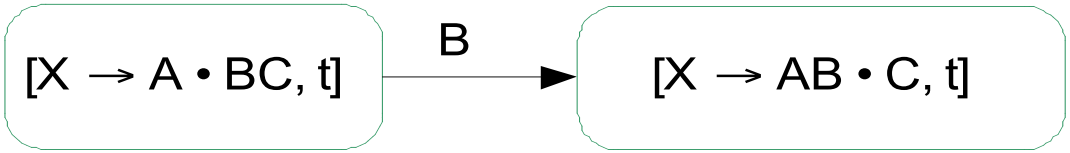
Intuitively the LR(1) item

`[ X → A•BC, t ]`

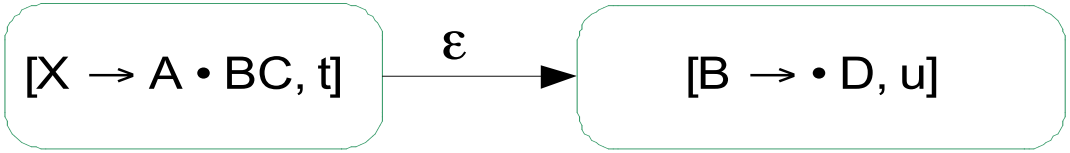
indicates that `A` is on top of the stack and that we only want to recognise `B` if `B` is followed by a string derivable from `Ct` i.e. if the current token is in `FIRST(Ct)`

LR(1) Transitions

Given an LR(1) item: `[X → A•BC, t]` add the transition



Also if `B` is a rule (non-terminal), then for every rule `B → D` add an ε-transition for every token `u` in `FIRST(Ct)` i.e add a transition iff the string following `B` is derivable from `Ct`.



We include the auxiliary item `[X' → •X, $]` to start the construction of the parsing table, which proceeds as before.

Example

Grammar

S → *id* | V '=' E

V → *id*

E → V | *int*

Grammar Expanded

S' → S $

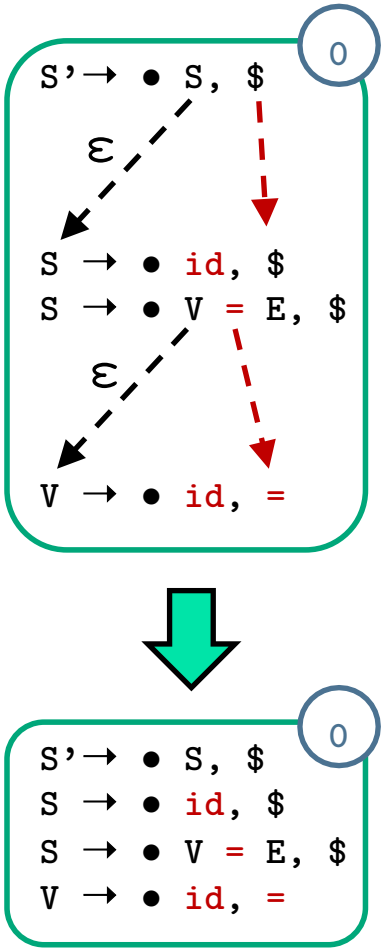
S → *id*

S → V '=' E

V → *id*

E → V

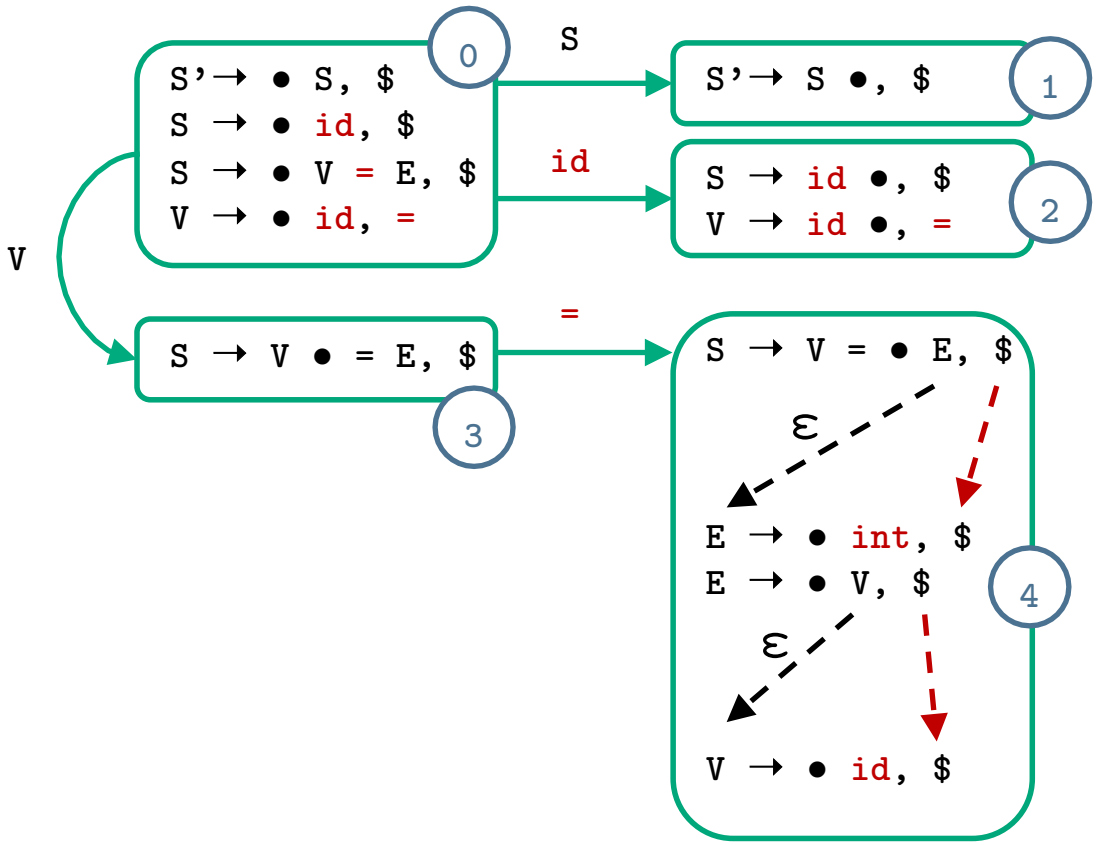
E → *int*



Example contd (1)

Grammar

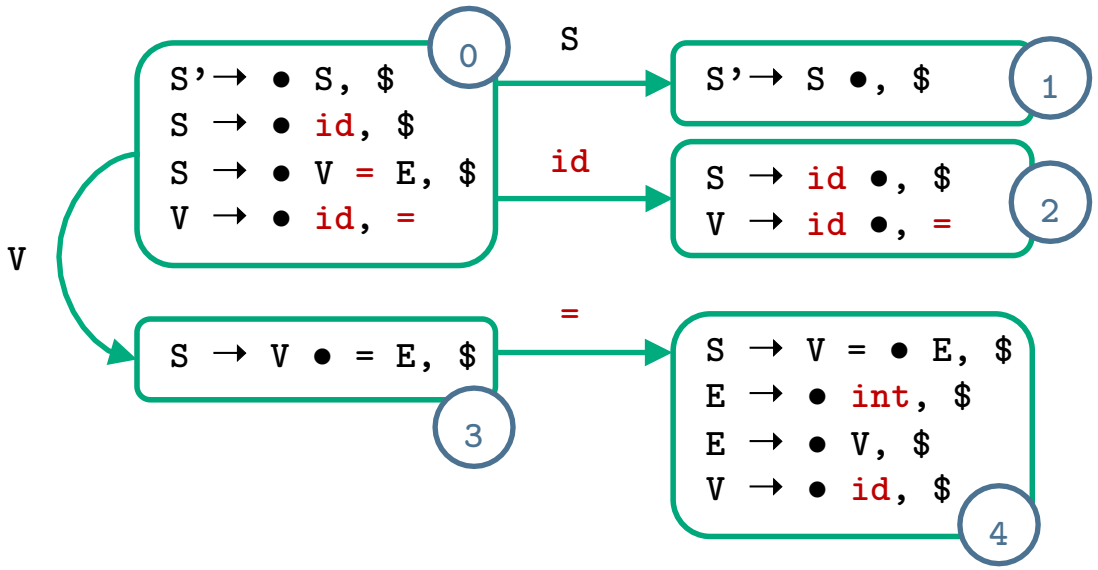
S' → S $
S → *id*
S → V '=' E
V → *id*
E → V
E → *int*



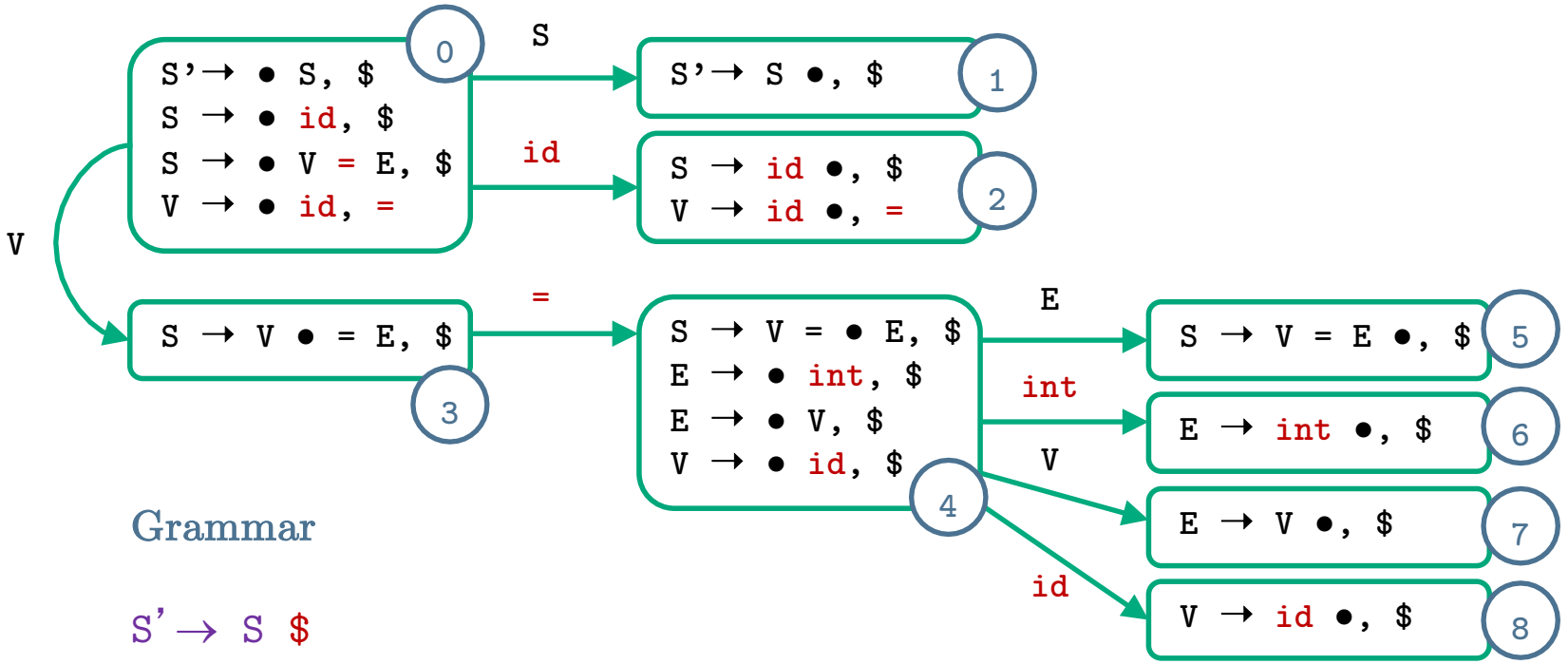
Example contd (2)

Grammar

S' → S $
S → *id*
S → V '=' E
V → *id*
E → V
E → *int*



Example Completed



Grammar

S' → S $

S → *id*

S → V '=' E

V → *id*

E → V

E → *int*

Parsing Table for Example

r1: S → *id*
r2: S → V '=' E
r3: V → *id*
r4: E → V
r5: E → *int*

Remember for LR(1) we only perform a reduction X → A•, t if the current token is t

State	ACTION				GOTO		
	id	*int*	=	$	E	V	S
0	s2					g3	g1
1				a			
2			r3	r1			
3			s4				
4	s8	s6			g5	g7	
5				r2			
6				r5			
7				r4			
8				r3			

LALR(1) Parsers

LALR(1) parsers are like LR(1) parsers but we perform an optimisation where we merge LR(1) states that have the same LR(0) items but differ in their look-ahead token.

An LALR item thus consists of an LR(0) item and a set of look-ahead tokens:

An LALR(1) item is a pair `[ LR(0) item, {t0, .., tn} ]`

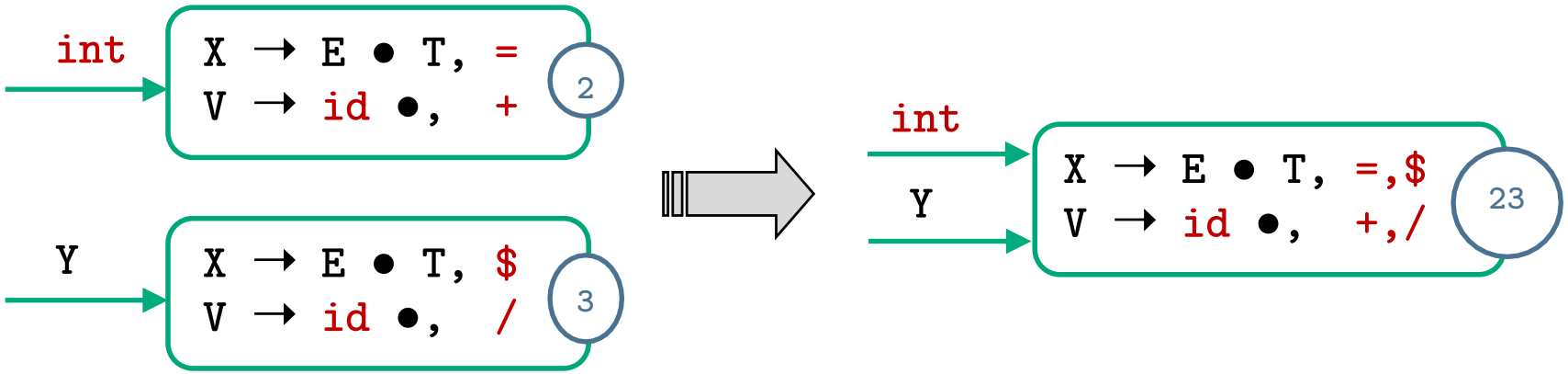
Because of this merging, it is possible for LALR(1) parsing to produce spurious reductions before an error is detected, whereas LR(1) parsing would detect the error immediately, e.g. for the grammar:

`A` → `'('` `A` `')'`
`A` → `'+'`

The input string `+)` will be caught immediately after shifting `+` in LR(1) while LALR(1) will shift `+` and then do the reduction `A` → `+` before catching the error if its look-ahead set for `A` → `+`• is `{ +, ) }`

LALR(1) Example

In the following example LR(1) states 2 and 3 have the same LR(0) items. We can therefore combine them



LR(1) States LALR(1) State

Note: It is possible to construct the DFA of LALR(1) items directly from the DFA of LR(0) items by using a technique known as propagating lookaheads. For details see Aho p240.

Ambiguity and Conflicts

Ambiguous grammars (i.e. where we can derive more than 1 parse tree for some input) cannot be LR(k) for any k. An LR(k) parser will reach a state where given the entire stack contents and the next k input tokens it cannot decide whether to shift or reduce.

Ambiguity can be removed by rewriting the grammar, or supported by augmenting the grammar with additional rules, e.g. by defining the associativity and precedence of operators.

`A` → *id* `A` → *id* '`[`' `Expr` '`]`'	Do we shift or do we reduce? => **Shift-Reduce conflict**
`A` → ε	ε-rules lead to a Shift-Reduce conflict anywhere `A` occurs on the RHS.
`Expr` → *id* `Var` → *id*	Which reduce should we perform? => **Reduce-Reduce conflict**

Shift-Reduce Conflicts

Consider the following grammar:

S → *if* E *then* S *else* S
S → *if* E *then* S
S → *id*

What is the interpretation for the input: *if* a *then* *if* b *then* c *else* d

In our parsing table we will have items:

[*if* E *then* S •, *else*]
[*if* E *then* S • *else* S, *any*]

If we shift our interpretation will be:

if a *then* *if* b *then* c *else* d

On the other hand if we reduce our interpretation becomes:

if a *then* *if* b *then* c *else* d

Many LR parser generators give priority to shifting. This is analogous to matching the longest sequence in Lexical Analysis. If this is not appropriate or we want to remove the shift-reduce conflict, then we need to rewrite the grammar.

Shift-Reduce Conflicts Contd.

We can remove a shift-reduce conflict by rewriting the grammar e.g.:

Consider the grammar:

```
S → if E then S else S
S → if E then S
S → other
E → 0 | 1
```

We can rewrite this grammar as:

```
S           → MatchedS
MatchedS    → if E then MatchedS else MatchedS
MatchedS    → other

S           → UnMatchedS
UnMatchedS  → if E then S
UnMatchedS  → if E then MatchedS else UnMatchedS
```

This works by permitting only a matched statement to come before an *else* in an *if* statement, forcing all else-parts to become matched as soon as possible.

Reduce-Reduce Conflicts

A **reduce-reduce conflict** occurs if there is a choice of reducing by 2 (or more) rules.

A common disambiguating rule used by LR parser generators for reduce-reduce conflicts is to give priority to the earliest rule listed in the grammar source file, alternatively we can rewrite the grammar.

Precedence. The following grammar is **ambiguous** because it doesn't define the associatively or precedence of + and *

```
Expr → Expr '+' Expr | Expr '*' Expr | '(' Expr ')' | int
```

We can rewrite this unambiguously as:

```
Expr   → Expr '+' Term   | Term
Term   → Term '*' Factor | Factor
Factor → '(' Expr ')' | int
```

Error Recovery (Optional Material)

On encountering an error entry, we can either stop and report a single error to the programmer, or we can attempt to recover from the error and so allow parsing to continue and further errors to be reported.

Many error recovery schemes exist, some attempt recovery by scanning forward in the token stream, some modify the parse stack, others do both.

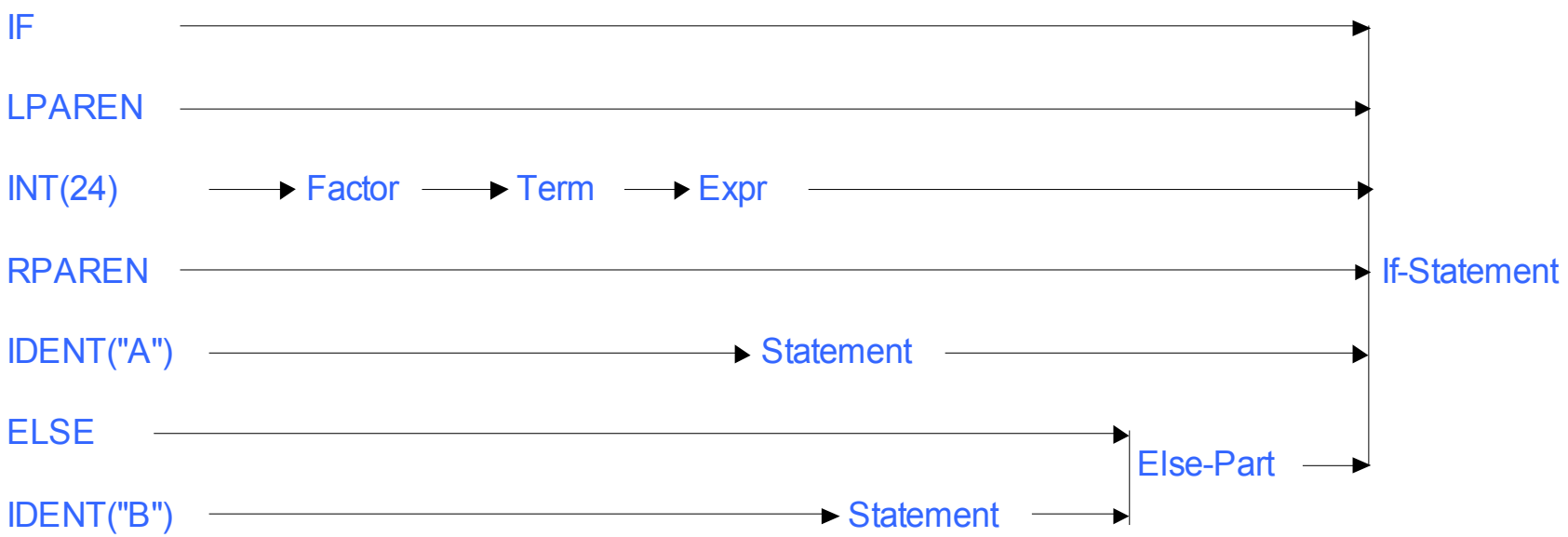
Example. YACC (Yet-Another Compiler-Compiler, Bison in GNU) uses extra Error-Recovery Rules that tell the parser how to resynchronise on error, e.g.

```
Expr → '(' error ')'
Expressions → error ';' Expression
```

The above rules specify that if a syntax error is encountered in an expression that the parser should skip to the next `)` or the next `;` In YACC, `error` is a special terminal symbol but has shift actions entered into the parsing table as if it were a normal token. See Appel p79 or Aho p264 for more details.

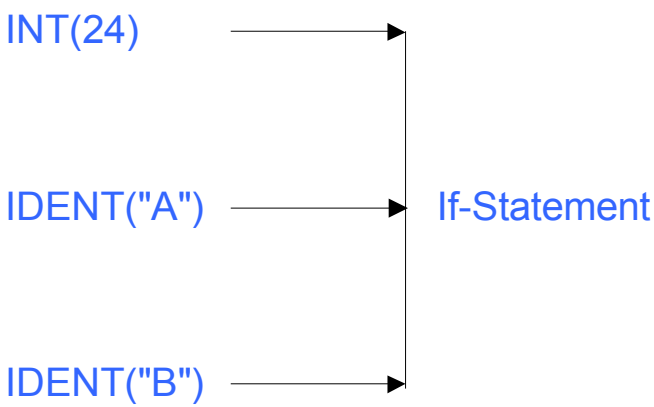
Parse Trees

In a **parse tree**, we build leaf nodes for each input token (when we **shift**) and non-leaf nodes for each rule (when we **reduce**).



Abstract Syntax Trees (ASTs)

Although parse trees fully represent the grammatical structure of the input, they hold unnecessary information and are inconvenient to work with. For example we don't normally care about keywords and bracketing symbols in the later stages (semantic analysis, code generation) of the compiler. An **Abstract Syntax Tree** (AST) is a refined version of the parse tree that retains only the information that is useful for later stages of the compiler, or for error reporting or at runtime. For example the AST for an **if** statement might be:



Abstract Syntax Trees Contd.

ASTs can be constructed either with a separate pass, or by attaching AST construction code directly to grammar rules, e.g.

```
Statement → print '(' Expr:E ')'                          | return new PrintNode (E)
Statement → ident:D '=' Expr:E                            | return new AssignNode(D,E)
Statement → if '(' Expr:E ')' Statement:S                 | return new IfNode(E, S, null)
Statement → if '(' Expr:E ')' Statement:S1 else Statement:S2
                                                          | return new IfNode (E, S1, S2)
Statement → Statement:S1 ';' Statement:S2                 | return new StatListNode (S1, S2)
Expr → Expr:E  Op:O  Term:T                               | return new ExprNode (E, O, T)
Expr → Term:T                                             | return T
Term → '(' Expr:E ')'                                     | return E
Term → ident:D                                            | return new IdNode (D.stringvalue)
Term → int:N                                              | return new IntNode (N.intvalue)
Op   → '+'                                                | return new ArithOpNode ('+')
Op   → '-'                                                | return new ArithOpNode ('-')
```

Summary

Bottom-up parsing is a very powerful technique for recognising the syntax of programming languages. Although more complicated to understand than top-down parsers, it is popular since LR grammars are usually simpler and more natural than LL grammars, and LR parser generators can produce efficient parsers with good error reporting.

For more details on bottom up parsing and further exercises see

[Cooper – Chapter 3]
[Appel – Chapter 3]
[Aho – Chapter 4]

Flex-Bison (C++) Example

Grammar

```
program → statement_sequence

statement_sequence → statement | statement_sequence ';' statement

statement → if_statement | while_statement | assign_statement

assign_statement → identifier ':=' expression

if_statement → 'if' expression 'then' statement_sequence 'end'

while_statement → 'while' expression 'do' statement_sequence 'end'

expression → expression '+' expression
           | expression '-' expression
           | expression '*' expression
           | expression '/' expression
           | '(' expression ')'
           | identifier
           | number
```

scanner.l

```
%{
  #include <string>
  #include "astnode.h"
  #include "parser.hpp"
%}

digit          [0-9]
number         {digit}+
letter         [a-zA-Z]
identifier     {letter}+
whitespace     [ \t\n]+
comment        //[^\n]*\n
```

- For this example we'll use C++ code with our bison grammar. astnode.h will hold C++ class definitions for our AST nodes.
- parser.hpp is the include file generated by bison-c++.

scanner.l

```
%%

if              return IF;
then            return THEN;
else            return ELSE;
while           return WHILE;
do              return DO;
":="            return BECOMES;
"="             return EQUALS;
"<"             return LESSTHAN;
"+"             return PLUS;
"-"             return MINUS;
"("             return LPAREN;
")"             return RPAREN;
";"             return SEMICOLON;
```

scanner.l

```
{number}        yylval.string = new std::string(yytext,yyleng);
                return INTEGER;
{identifier}    yylval.string = new std::string(yytext,yyleng);
                return IDENTIFIER;
{whitespace}    { }
.               return ERROR;
```

- To keep our **flex** code very simple, token attributes (e.g. for numbers) will be generated in **bison**. We'll just pass a C++ string.

parser.y

```
%{

#include "astnode.h"
#include <cstdio>
#include <cstdlib>

extern int yylex();  /* Lexical analyser generated by flex */

void yyerror(const char *message) {
  std::printf("Error: %s\n", message);
  std::exit(1);
}

StatSeq *ast;     /* Pointer to root of Abstract Syntax Tree */

%}
```

parser.y

```
%union {
  int            token;
  std::string   *string;
  Identifier    *id;
  Expression    *expression;
  Statement     *statement;
  StatSeq       *statseq;
  ASTnode       *node;
}
```

```
%token <token>     IF THEN ELSE WHILE DO READ WRITE
%token <token>     BECOMES EQUALS LESSTHAN PLUS MINUS
%token <token>     LPAREN RPAREN SEMICOLON
%token <string>    INTEGER ERROR
%token <id>        IDENTIFIER
```

parser.y

```
%type <id>          identifier
%type <statseq>     program statement_sequence
%type <statement>   statement begin_statement assign_statement
%type <statement>   if_statement while_statement
%type <expression>  expression  number
%type <token>       comparator

/* Associativity of operators */
%left PLUS MINUS

/* Start symbol. Will default to first non-terminal symbol */
%start program
```

parser.y

```
program:
  statement_sequence
  { ast = $1; }
;

statement_sequence:
  statement
  { $$ = new StatSequence();
    $$->statements.push_back($1);
  }
| statement_sequence SEMICOLON statement
  { $1->statements.push_back($3);
  }
;

statement:
  if_statement          { $$ = $1; }
| while_statement       { $$ = $1; }
| assign_statement      { $$ = $1; }
;
```

parser.y

```
if_statement:
  IF expression THEN statement_sequence END
  { $$ = new IfStatement(*$2, *$4); }
;

while_statement:
  WHILE expression DO statement_sequence END
  { $$ = new WhileStatement(*$2, *$4); }
;

assign_statement:
  identifier BECOMES expression
  { $$ = new Assignment(*$1, *$3); }
;
```

parser.y

```
expression:
  identifier
  { $$ = $1; }
| number
  { $$ = $1; }
| expression PLUS expression
  { $$ = new Operator(*$1, $2, *$3); }
| expression MINUS expression
  { $$ = new Operator(*$1, $2, *$3); }
| expression comparator expression
  { $$ = new Operator(*$1, $2, *$3); }
| LPAREN expression RPAREN
  { $$ = $2; }
;
```

parser.y

```
identifier:
  IDENTIFIER
  { $$ = new Identifier(*$1);
    delete $1;
  }
;

number:
  INTEGER
  { $$ = new Number(atol($1->c_str()));
    delete $1;
  }
;

comparator:
  EQUALS
| LESSTHAN
;
```

astnode.h

```
#include <vector>
#include <iostream>

class ASTnode {
public:
  virtual ~ASTnode() {}
  virtual void semantic_check () {}
};

class Expression : public ASTnode {};
typedef std::vector<Expression*> ExpressionList;

class Statement : public ASTnode {};
typedef std::vector<Statement*> StatementList;

class StatSeq: public Statement {
public:
  StatementList statements;
  StatSeq () {}
};
```

astnode.h

```
class IfStatement : public Statement {
public:
  Expression& expr;
  StatSeq& thenS;

  IfStatement(Expression& expr, StatSeq& thenS)
        : expr(expr), thenS(thenS) {}
};


class WhileStatement : public Statement {
public:
  Expression& expr;
  StatSeq& doS;

  WhileStatement(Expression& expr, StatSeq& doS)
        : expr(expr), doS(doS) {}
};
```

astnode.h

```
class Assignment : public Statement {
public:
  Identifier& var;
  Expression& expr;
  Assignment(Identifier& var, Expression& expr)
        : var(var), expr(expr) {}
};

class Number : public Expression {
public:
  int value;
  Number(int value) : value(value) {}
};

class Identifier : public Expression {
public:
  std::string id;
  Identifier(const std::string& id) : id(id) {}
};
```

main.cpp

```
#include <iostream>
#include "astnode.h"

using namespace std;

extern int yyparse();

extern StatSeq* ast;

int main(int argc, char **argv) {
  yyparse();
  return 0;
}
```

makefile

```
all: parser
OBJS = scanner.o parser.o main.o
CPPFLAGS =

parser.cpp: parser.y
	bison -d -v -o $@ $^

parser.hpp: parser.cpp

scanner.cpp: scanner.l parser.hpp
	flex -o $@ $^

%.o: %.cpp
	g++ -c $(CPPFLAGS) -o $@ $<

parser: $(OBJS)
	g++ -o $@ $(OBJS)
```