Chapter 3: Lexing and Parsing

Aarne Ranta

Slides for the book "Implementing Programming Languages. An Introduction to Compilers and Interpreters", College Publications, 2012.

Lexing and Parsing*

Deeper understanding of the previous chapter

Regular expressions and finite automata

- the compilation procedure
- why automata may explode in size
- why parentheses cannot be matched by finite automata

Context-free grammars and parsing algorithms.

- LL and LR parsing
- why context-free grammars cannot alone specify languages
- why conflicts arise

The standard tools

The code generated by BNFC is processed by other tools:

- **Lex** (Alex for Haskell, JLex for Java, Flex for C)
- **Yacc** (Happy for Haskell, Cup for Java, Bison for C)

Lex and YACC are the original tools from the early 1970's. They are based on the theory of formal languages:

- **Lex** code is **regular expressions**, converted to **finite automata**.
- **Yacc** code is **context-free grammars**, converted to **LALR(1) parsers**.

The theory of formal languages

A formal language is, mathematically, just any set of **sequences of symbols**.

Symbols are just elements from any finite set, such as the 128 7-bit ASCII characters.

Programming languages are examples of formal languages.

In the theory, usually simpler languages are studied.

But the complexity of real languages is mostly due to repetitions of simple well-known patterns.

Regular languages

A **regular language** is, like any formal language, a set of **strings**, i.e. sequences of **symbols**, from a finite set of symbols called the **alphabet**.

All regular languages can be defined by **regular expressions** in the following set:

expression	**language**
'a'	$\{\mathtt{a}\}$
AB	$\{ab \mid a \in [\![A]\!], b \in [\![B]\!]\}$
A | B	$[\![A]\!] \cup [\![B]\!]$
A*	$\{a_1 a_2 \ldots a_n \mid a_i \in [\![A]\!], n \geq 0\}$
`eps`	$\{\epsilon\}$ (empty string)

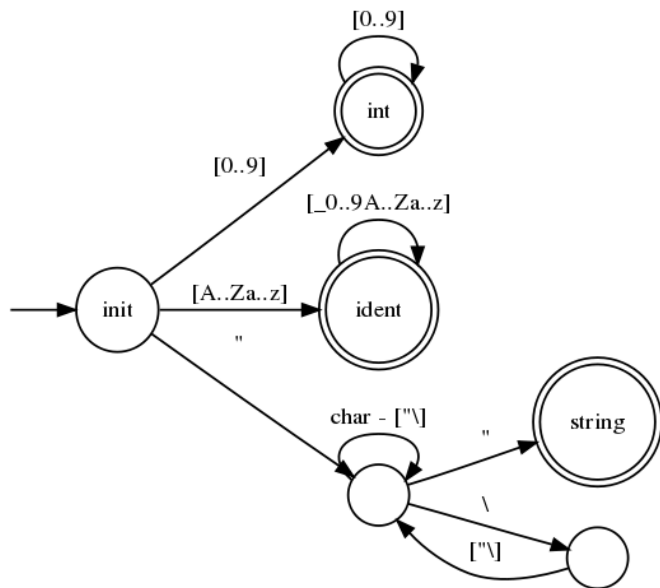
$[\![A]\!]$ is the set corresponding to the expression A.

Finite automata

The most efficient way to analyse regular languages is by **finite automata**, which can be compiled from regular expressions.

Finite automata are graphs with symbols on edges.

Example: a string that is either an integer literal or an identifier or a string literal.



The corresponding regular expression

```
  digit digit*
| letter ('_' | letter | digit)*
| '"' (char - ('\' | '"') | '\' ('\' | '"'))* '"'
```

Recognition

Start from the **initial state**, that is, the node marked "init".

It go to the next **state** (i.e. node) depending on the first character.

- If it is a digit `0...9`, the state is the one marked "int".
  - With more digits, the recognition loops back to this state.

- If it is a letter, go to ident
- If it is a double quote, go to next state

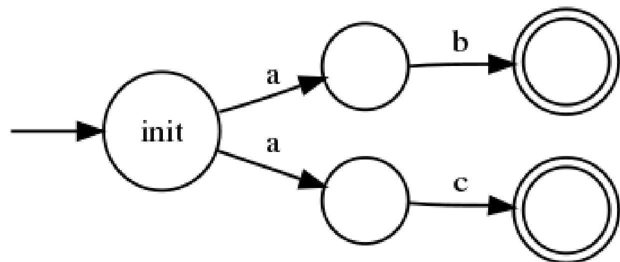
Deterministic and nondeterministic automata

Deterministic: any input symbol has at most one **transition**, that is, at most one way to go to a next state.

Example: the one above.

Nondeterministic: some symbols may have many transitions.

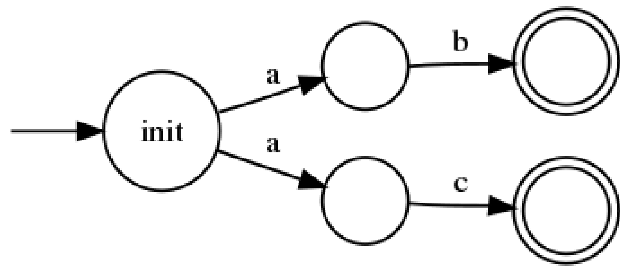
Example: `a b | a c`



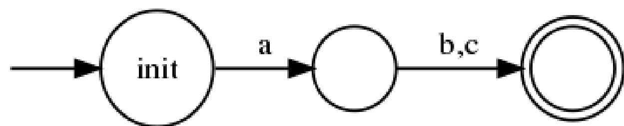
Correspondence

`a (b | c)`

Nondeterministic:



Deterministic:



Why nondeterministic at all

Deterministic ones can be tedious to produce.

Example: recognizing English words

```
a able about account acid across act addition adjustment ...
```

It would be a real pain to write a bracketed expression in the style of `a (c | b)`, and much nicer to just put |'s between the words and let the compiler do the rest!

Fortunately, nondeterministic automata can always be converted to deterministic ones.

The compilation of regular expressions

The standard compilation of regular expressions:

1. **NFA generation**: convert the expression into a **non-deterministic finite automaton, NFA**.
2. **Determination**: convert the NFA into a **deterministic finite automaton, DFA**.
3. **Minimization**: minimize the size of the deterministic automaton.

As usual in compilers, each of these phases is simple in itself, but doing them all at once would be complicated.

Step 1. NFA generation

Input: a regular expression written by just the five basic operators.

Output: an NFA which has exactly one initial state and exactly one final state.

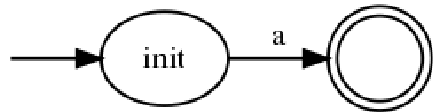
The "exactly one" condition makes it easy to combine the automata.

The NFA's use **epsilon transitions**, which consume no input, marked with the symbol ϵ.

NFA generation is an example of **syntax-directed translation**, and could be recommended as an extra assignment!

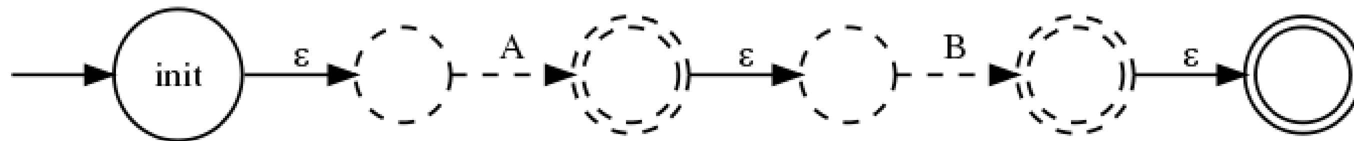
NFA from a single symbol

Symbol. The expression `a` is compiled to



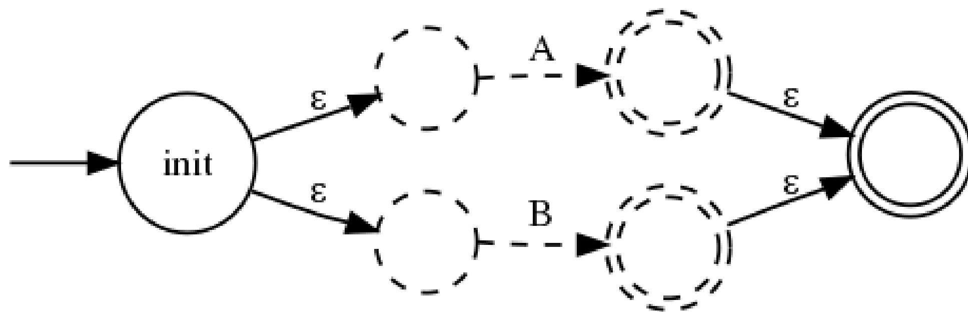
NFA from sequence

Sequence. The expression A B is compiled by combining the automata for A and B (drawn with dashed figures with one initial and one final state) as follows:



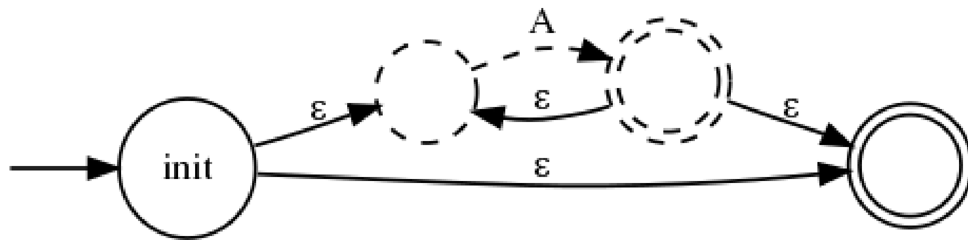
NFA from union

Union. The expression $A \mid B$ is compiled as follows:



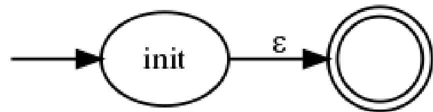
NFA from closure

Closure. The expression $A*$ is compiled as follows:



NFA from empty string

Empty. The expression `eps` is compiled to



The mathematical definition of automata

Definition. A **finite automaton** is a 5-tuple $\langle \Sigma, S, F, i, t \rangle$ where

- Σ is a finite set of symbols (the **alphabet**)
- S is a finite set of **states**
- $F \subset S$ (the **final states**)
- $i \in S$ (the **initial state**)
- $t : S \times \Sigma \to \mathcal{P}(S)$ (the **transition function**)

An automaton is **deterministic**, if $t(s, a)$ is a singleton for all $s \in S, a \in \Sigma$. Otherwise, it is **nondeterministic**, and then moreover the transition function is generalized to $t : S \times \Sigma \cup \{\epsilon\} \to \mathcal{P}(S)$ (with **epsilon transitions**).

Step 2. Determination

Input: NFA

Output: DFA for the same regular language

Procedure: **subset construction**

- idea: for every state *s* and symbol *a* in the automaton, form a new state $\sigma(s, a)$ that "gathers" all those states to which there is a transition from *s* by *a*.

The subset construction

$\sigma(s, a)$ is the set of those states s_i to which one can arrive from *s* by consuming just the symbol *a*. This includes of course the states to which the path contains epsilon transitions.

The transitions from $\sigma(s, a) = \{s_1, \ldots, s_n\}$ for a symbol *b* are all the transitions with *b* from any s_i. (Must of course be iterated to build $\sigma(\sigma(s, a), b)$.)

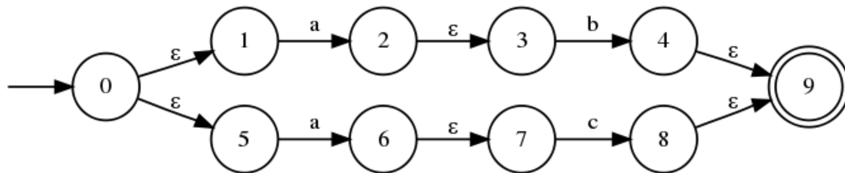
The state $\sigma(s, a) = \{s_1, \ldots, s_n\}$ is final if any of s_i is final.

Example of compilation

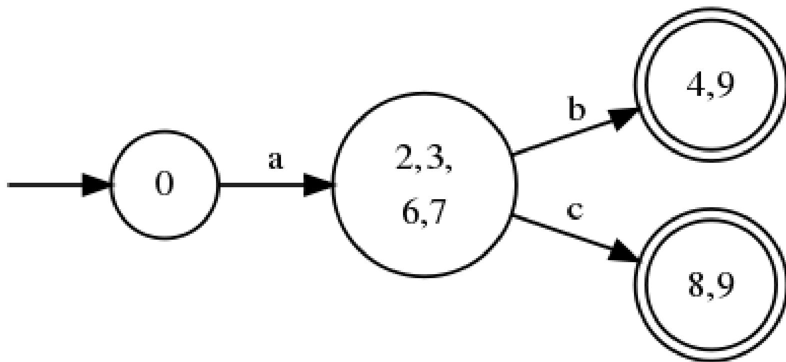
Start with the expression

```
a b | a c
```

Step 1. Generate NFA by syntax-directed translation



Step 2. Generate DFA by subset construction



Step 3. Minimization

The DFA above still has **superfluous states**.

They are states without any **distinguishing strings**.

A distinguishing string for states s and u is a sequence x of symbols that ends up in an accepting state when starting from s and in a non-accepting state when starting from u.

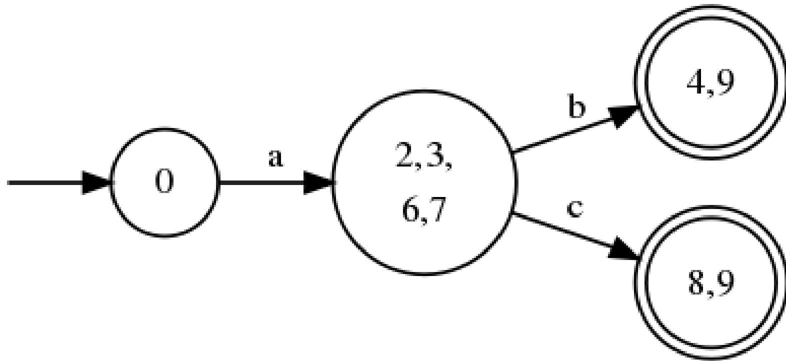
In the previous DFA, the states 0 and {2,3,6,7} are distinguished by the string `ab`. When starting from 0, it leads to the final state {4,9}. When starting from {2,3,6,7}, there are no transitions marked for `a`, which means that any string starting with `a` ends up in a **dead state** which is non-accepting.

But the states {4,9} and {8,9} are not distinguished by any string.

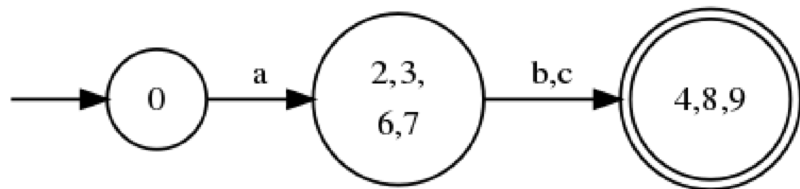
Minimization means merging superfluous states.

Example of minimization

Input: DFA



Output: minimal DFA



Details of the algorithm omitted.

Correspondence theorem

The following three are equivalent:

- regular languages
- regular expressions
- finite automata (by determination, both NFA and DFA)

Closure properties

Regular languages are closed under many operations.

Example: **complement**. If L is a regular language, then also $-L$, is one: the set of all those strings of the alphabet that do not belong to L.

Proof: Assume we have a DFA corresponding to A. Then the automaton for $-A$ is obtained by inverting the status of each accepting state to non-accepting and vice-versa! This requires a version of the DFA where all symbols have transitions from every state; this can always be guaranteed by adding a dedicated dead state as a goal for those symbols that are impossible to continue with.

Exponential size

The size of a DFA can be exponential in the size of the NFA (and therefore of the expression).

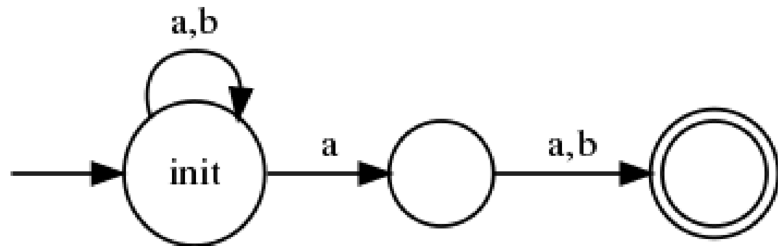
The subset construction shows a potential for this, because there could be a different state in the DFA for *every* subset of the NFA, and the number of subsets of an *n*-element set is 2^n.

Example of size explosion

Strings of *a*'s and *b*'s, where the *n*th element *from the end* is an *a*.

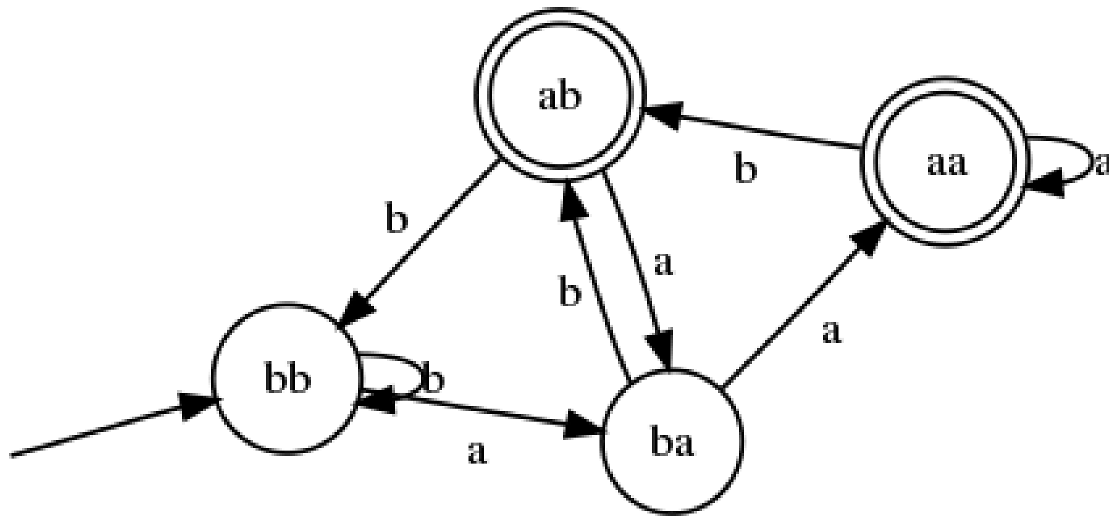
Consider this in the case n=2.

```
(a|b)* a (a|b)
```



Deterministic version

The *states* must "remember" the last two symbols that have been read. Thus the states can be named *aa*, *ab*, *ba*, and *bb*.



Of course, also possible by mechanical subset construction.

Matching parentheses

Use `a` for "(" and `b` for ")", and consider the language

$$\{a^n b^n | n = 0, 1, 2 \ldots\}$$

Easy to define in a BNF grammar:

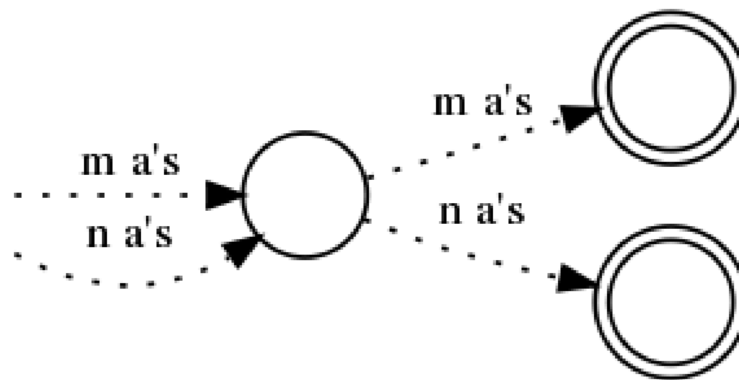
```
S ::= ;
S ::= "a" S "b" ;
```

But is this language regular? I.e. can there be a finite automaton?

Matching parentheses is not a regular language

Let s_n be the state where the automaton has read *n a*'s and starts to read *b*'s.

Thus there must be a different state for every n, because, if we had $s_m = s_n$ for some $m \neq n$, we would recognize $a^n b^m$ and $a^m b^n$.



Nested comments

You might want to treat them in the lexer. Thus

```
a /* b /* c */ d */ e
```

would give

```
a                   e
```

But in standard compilers, it gives

```
a              d */ e
```

Reason: the lexer is implemented by a finite automaton, which cannot match parentheses - in this, case comment delimiters.

Context-free grammars and parsing

A **context-free grammar** is the same as a **BNF grammar**, consisting of rules of the form

$$C ::= t_1 \ldots t_n$$

where each t_i is a terminal or a nonterminal.

All regular languages can be defined by context-free grammars. The inverse does not hold, as proved by matching parentheses.

Complexity

Price to pay: context-free parsing can be more complex than recognition with automata - *cubic* (O(n^3)), whereas recognition with a finite automaton is *linear* in the length of the string (O(n)).

However, programming languages are usually designed in such a way that their parsing is linear. They use a restricted subset of context-free grammars.

LL(k) parsing

The simplest practical way to parse programming languages.

LL(k) = *left-to-right parsing, leftmost derivations, lookahead k*.

Also called **recursive descent parsing**

Sometimes used for implementing parsers by hand (without parser generators).

Example: the **parser combinators** of Haskell.

Lookahead

Each category has a function that inspects the first token and decides what to do.

One token is the **lookahead** in LL(1). LL(2) parsers inspect two tokens, and so on.

Example

Grammar

```
SIf.    Stm ::= "if" "(" Exp ")" Stm ;
SWhile. Stm ::= "while" "(" Exp ")" Stm ;
SExp.   Stm ::= Exp ;
EInt.   Exp ::= Integer ;
```

LL(1) parsing functions, skeleton

Stm pStm() :
  if (*next* = `"if"`) . . . // *try to build tree with SIf*
  if (*next* = `"while"`) . . . // *try to build tree with SWhile*
  if (*next is integer*) . . . // *try to build tree with SExp*

Exp pExp() :
  if (*next is integer* k) **return** *SExp* k

A complete parsing function branch

```
Stm pStm() :
  if (next = "if")
    ignore("if")
    ignore("(")
    Exp e := pExp()
    ignore(")")
    Stm s := pStm()
    return SIf (e, s)
```

In words: parse expression *e* and statement *s*, build a *SIf* three, ignore the terminals.

Conflicts

Example: `if` statements with and without `else`

```
SIf.     Stm ::= "if" "(" Exp ")" Stm
SIfElse. Stm ::= "if" "(" Exp ")" Stm "else" Stm
```

In an LL(1) parser, which rule to choose when we see the token `if`?

As there are two alternatives, we have a **conflict**.

Rewriting the grammar

One way to solve (some) conflicts: rewrite the grammar using **left factoring**:

```
SIE.   Stm  ::= "if" "(" Exp ")" Stm Rest
RElse. Rest ::= "else" Stm
REmp.  Rest ::=
```

To get the originally wanted abstract syntax, we have to convert the trees:

```
SIE exp stm REmp            ==>  SIf exp stm
SIE exp stm (RElse stm2)    ==>  SIfElse exp stm stm2
```

Warning: it can be tricky to rewrite a grammar so that it enables LL(1) parsing.

Left recursion

Perhaps the most well-known problem of LL(k).

A rule is left-recursive if it has the form

$$C ::= C \ldots$$

Common in programming languages, because operators like `+` are left associative.

```
Exp ::= Exp "+" Integer
Exp ::= Integer
```

The LL(1) parser loops, because, to build an `Exp`, the parser first tries to build an `Exp`, and so on. No input is consumed when trying this.

Rewriting the grammar

To avoid left recursion

```
Exp  ::= Integer Rest
Rest ::= "+" Integer Rest
Rest ::=
```

The new category `Rest` has **right recursion**, which is harmless. A tree conversion is of course needed as well.

Warning: very tricky, in particular with **implicit left recursion**

```
A ::= B ...
B ::= A ...
```

Parser tables

The mechanical way way to see conflicts and to generate a parser.

A row for each category sought, a column for each token encountered.
The cells show what rules apply.

```
SIf.    Stm ::= "if" "(" Exp ")" Stm ;
SWhile. Stm ::= "while" "(" Exp ")" Stm ;
SExp.   Stm ::= Exp ";" ;
EInt.   Exp ::= Integer ;
```

-	**if**	**while**	**integer**	**(**	**)**	**;**	**$ (END)**
Stm	SIf	SWhile	SExp	-	-	-	-
Exp	-	-	EInt	-	-	-	-

Conflicts in an LL(1) table

Conflict: a cell contains more than one rule.

This happens if we add the `SIfElse` rule: the cell (Stm,`if`) then contains both `SIf` and `SIfElse`.

LR(k) parsing

LR(k), *left-to-right parsing, rightmost derivations, lookahead k*.

Used in YACC-like parsers (and thus in BNFC).

No problems with left factoring or left recursion!

Both algorithms read their input left to right. But LL builds the trees from left to right, LR from right to left.

But LL uses **leftmost derivation**, LR uses **rightmost derivation**.

Leftmost and rightmost derivations

Leftmost (as in LL)

```
Stm --> while ( Exp ) Stm
    --> while (   1 ) Stm
    --> while (   1 ) if ( Exp ) Stm
    --> while (   1 ) if (   0 ) Stm
    --> while (   1 ) if (   0 ) Exp ;
    --> while (   1 ) if (   0 )   6 ;
```

Rightmost (as in LR)

```
Stm --> while ( Exp ) Stm
    --> while ( Exp ) if ( Exp ) Stm
    --> while ( Exp ) if ( Exp ) Exp ;
    --> while ( Exp ) if ( Exp )   6 ;
    --> while ( Exp ) if (   0 )   6 ;
    --> while (   1 ) if (   0 )   6 ;
```

How LR(1) works

Read input, builds a **stack** of results, combined results when a grammar rule can be applied to the top of the stack.

Decide an **action** when seeing the next token (lookahead 1):

- **Shift**: read one more token (i.e. move it from input to stack).
- **Reduce**: pop elements from the stack and replace by a value.
- **Goto**: jump to another state and act accordingly.
- **Accept**: return the single value on the stack when no input is left.
- **Reject**: report that there is input left but no action to take, or that the input is finished but the stack is not one with a single value of expected type.

Shift and reduce are the most common actions.

LR(1) example

Grammar

```
1. Exp  ::= Exp "+" Exp1
2. Exp  ::= Exp1
3. Exp1 ::= Exp1 "*" Integer
4. Exp1 ::= Integer
```

Parsing `1 + 2 * 3`

stack	input	action
	1 + 2 * 3	shift
1	+ 2 * 3	reduce 4
Exp1	+ 2 * 3	reduce 2
Exp	+ 2 * 3	shift
Exp +	2 * 3	shift
Exp + 2	* 3	reduce 4
Exp + Exp1	* 3	shift
Exp + Exp1 *	3	shift
Exp + Exp1 * 3	3	reduce 3
Exp + Exp1		reduce 1
Exp		accept

How to decide on the action

(Looking at the previous slide)

Initially, the stack is empty, so the parser must *shift* and put the token `1` to the stack.

The grammar has a matching rule, rule 4, and so a *reduce* is performed.

Then another reduce is performed by rule *2*. Why? Because the next token (the lookahead) is `+`, and there is a rule that matches the sequence `Exp +`.

If the next token were `*`, the second reduce would not be performed. This is shown later, when the stack is `Exp + Exp1`.

LR(1) parser table

Rows: **parser states**

Columns: for terminals and nonterminals

Cells: parser actions

Parser state = grammar rule together with the position (a dot .) that has been reached.

```
Stm ::= "if" "(" . Exp ")" Stm
```

Example: an LR(1) table produced by BNFC and Happy from the previous grammar. There are two added rules:

- rule (0) that produces integer literal terminals (`L_int`) from the nonterminal `Integer`
- a start rule, which adds the token `$` to mark the end of the input

For *shift*, the next state is given. For *reduce*, the rule number is given.

		`+`	`*`	`$`	`L_int`
0	(start)	-	-	-	s3
3	`Integer -> L_int .`	r0	r0	r0	-
4	`Exp1 -> Integer .`	r4	r4	r4	-
5	`Exp1 -> Exp1 . "*" Integer`	-	s8	-	-
6	`%start_pExp -> Exp . $` `Exp -> Exp . "+" Exp1`	s9	-	a	-
7	`Exp -> Exp1 .` `Exp1 -> Exp1 . "*" Integer`	r2	s8	r2	-
8	`Exp1 -> Exp1 "*" . Integer`	-	-	-	s3
9	`Exp -> Exp "+" . Exp1`	-	-	-	s3
10	`Exp -> Exp "+" Exp1 .` `Exp1 -> Exp1 . "*" Integer`	r1	s8	r1	-
11	`Exp1 -> Exp1 "*" Integer .`	r3	r3	r3	-

Table size and expressivity

For LR(1): the number of rule positions multiplied by the number of tokens and categories

For LR(2): the square of the number of tokens and categories

LALR(1) = look-ahead LR(1): merging states that are similar to the left of the dot (e.g. states 6, 7, and 10 in the above table). Standard tools (and BNFC) use this.

Expressivity:

- LR(0) < LALR(1) < LR(1) < LR(2) ...
- LL(k) < LR(k)

That a *grammar* is in LALR(1), or any other of the classes, means that its parsing table has no conflicts. Thus none of these classes can contain ambiguous grammars.

Finding and resolving conflicts

Conflict: several actions in a cell.

Two kinds of conflicts in LR and LALR:

- **shift-reduce conflict**: between shift and reduce actions.
- **reduce-reduce conflict** between two (or more) reduce actions.

The latter are more harmful, but also easier to eliminate.

The former may be tolerated, e.g. in Java and C.

Example: plain ambiguities

Assume that a grammar tries to distinguish between variables and constants:

```
EVar.   Exp ::= Ident ;
ECons.  Exp ::= Ident ;
```

Any `Ident` parsed as an `Exp` can be reduced with both of the rules.

Solution: remove one of the rules and leave it to the type checker to distinguish constants from variables.

Implicit ambiguities

A fragment of C++, where a declaration (in a function definition) can be just a type (`DTyp`), and a type can be just an identifier (`TId`). At the same time, a statement can be a declaration (`SDecl`), but also an expression (`SExp`), and an expression can be an identifier (`EId`).

```
SExp.   Stm  ::= Exp ;
SDecl.  Stm  ::= Decl ;
DTyp.   Decl ::= Typ ;
EId.    Exp  ::= Ident ;
TId.    Typ  ::= Ident ;
```

Detect the reduce-reduce conflict by tracing down a chain of rules:

```
Stm -> Exp -> Ident
Stm -> Decl -> Typ -> Ident
```

Solution: `DTyp` should only be valid in function parameter lists, and not as statements! This is actually the case in C++.

Dangling else

A classical shift-reduce conflicts

```
SIf.     Stm ::= "if" "(" Exp ")" Stm
SIfElse. Stm ::= "if" "(" Exp ")" Stm "else" Stm
```

The problem arises when `if` statements are nested. Consider the following input and position (.):

```
if (x > 0) if (y < 8) return y ; . else return x ;
```

There are two possible actions, which lead to two analyses of the statement. The analyses are made explicit by braces.

```
shift:   if (x > 0) { if (y < 8) return y ;  else return x ;}
reduce:  if (x > 0) { if (y < 8) return y ;} else return x ;
```

"Solution": always choose shift rather than reduce. This is well established, and a "feature" of languages like C and Java.

Strictly speaking, the BNF grammar is no longer faithfully implemented by the parser.

Debugging the grammar: info files

Happy generates an **info file** from the BNFC-generated parser file.

```
happy -i ParCPP.y
```

The resulting file `ParConf.info` is a very readable.

A quick way to check which rules are overshadowed in conflicts:

```
grep "(reduce" ParConf.info
```

Conflicts tend to cluster on a few rules. Extract these with

```
grep "(reduce" ParConf.info | sort | uniq
```

The conflicts are (usually) the same in all LALR(1) tools. Thus you can use Happy's info files even if you work with another tool.

Debugging the parse actions

Generate a **debugging parser** in Happy:

```
happy -da ParCPP.y
```

When you compile the BNFC test program with the resulting `ParCPP.hs`, it shows the sequence of actions when the parser is executed.

With Bison, you can use `gdb` (GNU Debugger), which traces the execution back to lines in the Bison source file.

The limits of context-free grammars

Some very simple formal languages are *not* context-free.

Example: the **copy language**.

$$\{ww|w \in (a|b)^*\}$$

e.g. aa, $abab$, but not aba. Observe that this is *not* the same as the context-free language

```
S ::= W W
W ::= "a" W | "b" W
W ::=
```

In this grammar, there is no guarantee that the two `W`'s are the same.

A consequence of the copy language

A compiler task: check that every variable is declared before it is used.

Language-theoretically, this is an instance of the copy language:

```
Program ::= ... Var ... Var ...
```

Consequently, checking that variables are declared before use is a thing that cannot be done in the parser but must be left to a later phase (type checking).

Notice that the copy language can still be parsed with a linear algorithm.