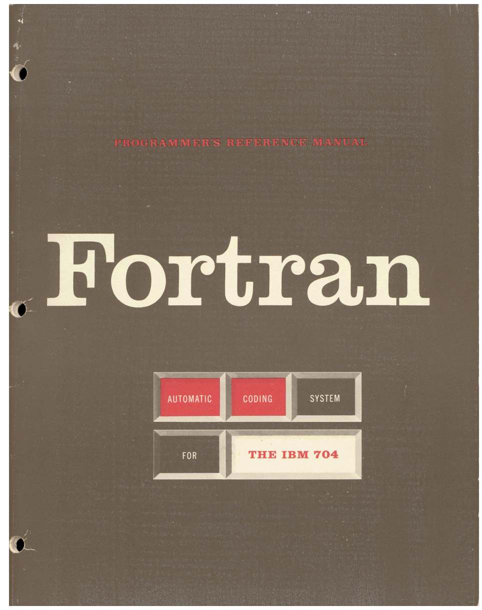
Compilers

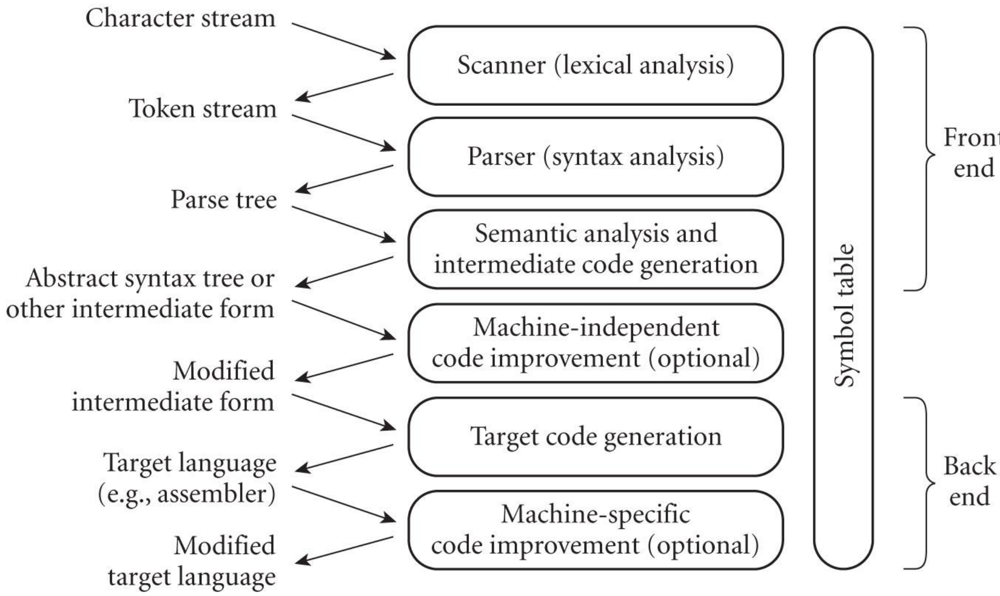


CSE 4305 / CSE 5317
M02 Syntactic Analysis
2023 Fall

M02

Syntactic Analysis

Phases of Compilation



Program Analysis

- A compiler's *Front End* is concerned with the *Analysis* of the source program.
  - Determining the *Meaning* of the source program.

- Generally thought of as breaking apart into three *phases*:
  - *Lexical Analysis*: Convert a text source program into *tokens*.
  - *Syntactic Analysis*: Convert a token stream into a *parse tree*.
  - *Semantic Analysis*: Enforce semantic rules and convert a parse tree into an *intermediate form*.
    - Two activities, but often happen in an interleaved fashion.

Syntactic Analysis

- Take a stream of *tokens* and convert it into a *parse tree*.

- Capture the *hierarchical* structure of the program.
  - Declarations, definitions, blocks, statements, expressions, …
- Provide representation for subsequent *semantic* analysis.
- Detect *syntactic* errors.
  - Mostly, improper structure, including misspelled keywords, bad statement and expression construction.
  - But *not*, e.g., undeclared identifiers, mismatched function calls.
  - (Why not?)

Syntactic Analysis

- So how to do this analysis?
- In the *Lexical Analysis* phase, we used a formal specification of the token formats.
  - Regular expressions (with action routines).

- Can we use Regular Expressions as the notation for the formal specification of program structure?
- *No!*

Why Not Regular Expressions?

- We have shown that Regular Expressions can be directly converted to an NFA, which can then be converted to a DFA.
- And that DFA can then be used to accept or reject an input string as belonging or not belonging to the RE's language.

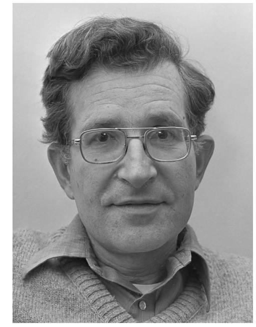
- The important letter here is ***F***, for *finite*.
- That means that the DFA cannot be used to recognize, e.g., *nesting* that is "too deep".
  - There just won't be enough states in the DFA.

Context-Free Grammar (CFG)

- Regular Expressions are limited in expressiveness.
  - Cannot define strings that are required to have …
    - Well-formed and/or nested parentheses, brackets, etc.
      e.g., `([[{(){}[][]}]{[]}])`
    - Matching pairs
      e.g., $\mathtt{a}^n\mathtt{b}^n \longrightarrow$ `ab`, `aabb`, `aaabbb`, …

- Next step up in expressiveness is a *Context-Free Grammar*.
  - Definitions can refer to themselves, i.e., can *recurse*.

Context-Free Grammar



Noam Chomsky

1. A set of *Terminal Symbols*.

    The *tokens* generated by the lexical analyzer.

2. A set of *Non-terminal Symbols*.

    Representing syntactic categories.
    Distinguished from the *Terminal Symbols*.

3. A set of *Production Rules*.

    Relating the syntactic categories to their structure. The LHS of each must be a single *Non-Terminal Symbol*. The RHS of each can be arbitrarily complex.

4. A *Start Symbol*.

    One of the *Non-terminal Symbols*.

https://commons.wikimedia.org/wiki/File:Noam_Chomsky_(1977).jpg

CFG Example

$$\mathit{expr} \longrightarrow \mathit{id} \mid \mathit{number} \mid \texttt{-}\ \mathit{expr} \mid \texttt{(}\ \mathit{expr}\ \texttt{)} \mid \mathit{expr}\ \mathit{op}\ \mathit{expr}$$

$$\mathit{id} \longrightarrow (\ \texttt{\_} \mid \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z}\ )(\ \texttt{\_} \mid \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \mid \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9}\ )^*$$

$$\mathit{op} \longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}$$

- Notice that *expr* refers to itself. This definition is *recursive*.
    - It's *not* left or right recursive exclusively.
- Q: What set of strings does *id* define?

CFG Example

- To *derive* (or *generate*) a string from a CFG, begin with the start symbol and replace non-terminals according to the rules until only terminals remain.
  - A *sentential form* is the start symbol or any form derived from it.
  - A *sentence* is a sentential form which has only terminal symbols.
- Example, generate

  `slope * x + intercept`

  using the *expr* CFG.

<u>expr</u>

expr op <u>expr</u>

expr <u>op</u> id

<u>expr</u> + *id*

expr op <u>expr</u> + *id*

expr <u>op</u> id + *id*

<u>expr</u> * *id* + *id*

<u>id</u> * *<u>id</u>* + *<u>id</u>*

`slope * x + intercept`

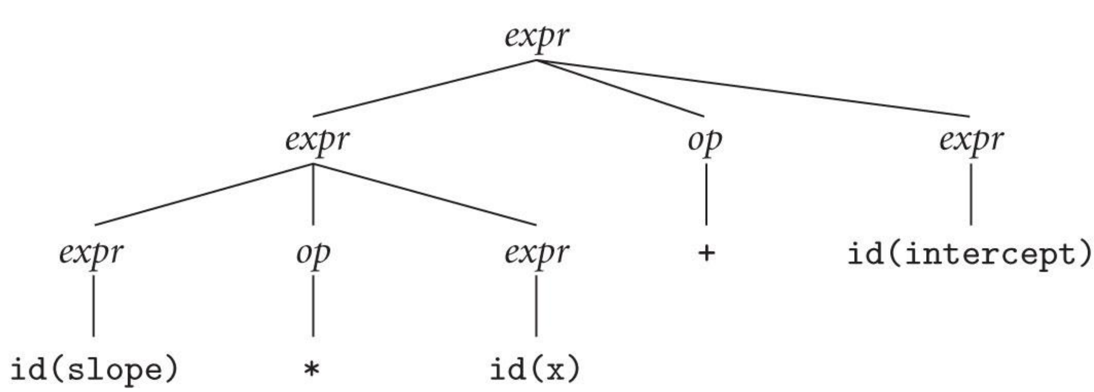
CFG Example

- That derivation was *Right-Most*.
  - We replaced the *right-most* non-terminal each time we took a step in the derivation.
  - *Except* for the final replacement of the *id* non-terminals with their actual words. (This was to make the derivation a little shorter and easier to fit on the slide.)
- A *Left-Most* derivation replaces the *left-most* non-terminal each time a step is taken in the derivation. (Duh.)

CFG Example

- The steps we took in the derivation correspond to the construction of a *parse tree*.
- The *root* of the parse tree is the *start symbol*.
- Every time we use a production rule, it's the same as adding a new (set of) node(s) to the tree.
- All internal nodes of the parse tree are *non-terminals*.
- The leaves of the final parse tree are *terminals*.
    - These terminals are the *tokens* of the original string.

CFG Example Parse Tree (from the Right)



CFG Example

- There's another way to do the generation of

  `slope * x + intercept`

  using the *expr* CFG.
- This derivation goes from the *left* instead of the *right*.

expr

expr op expr

id op expr

*id * expr*

*id * expr op expr*

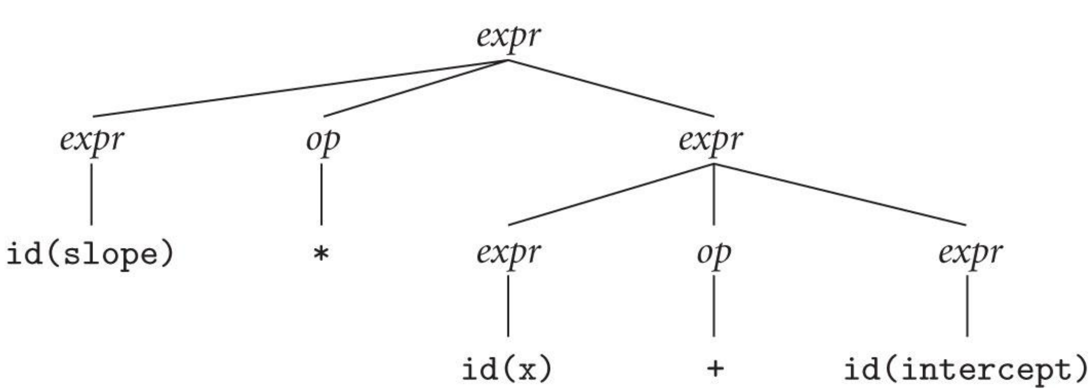
*id * id op expr*

*id * id + expr*

*id * id + id*

`slope * x + intercept`

CFG Example Parse Tree (from the Left)



CFG Example

- The CFG allows two parse trees for the same string because its definition is *ambiguous*.
  - Also, it allows incorrect parsing of operator precedence.
- So is the answer to always derive from the right?
  - After all, that got the correct derivation.
- ***No!***
  - Consider the right-most derivation of `intercept + slope * x`

expr

expr op expr

expr op id

expr * *id*

expr op expr * *id*

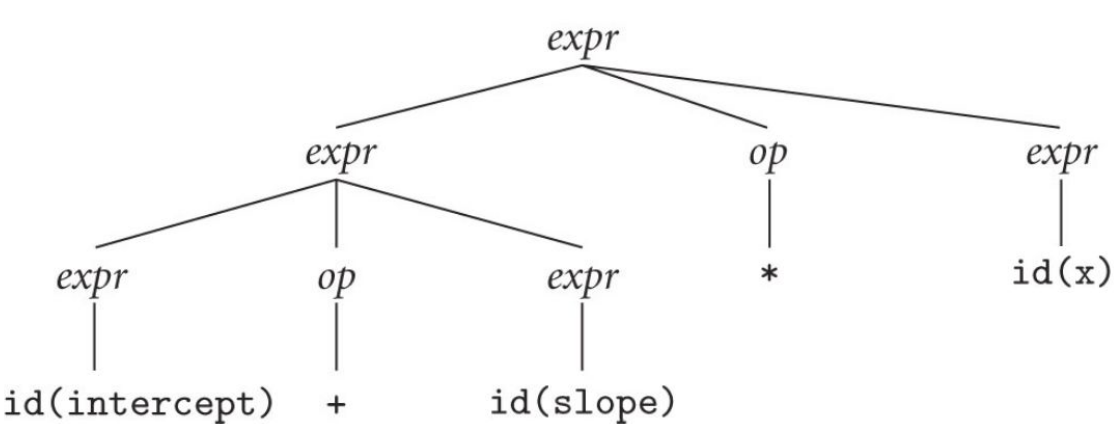
expr op id * *id*

expr + *id* * *id*

id + *id* * *id*

`intercept + slope * x`

CFG Example 2 Parse Tree (from the Right)



Improved CFG Example

- We need a CFG that is not ambiguous *and* honors our concept of operator precedence.
- A better (though more complex) CFG would be

  expr ⟶ *term* | *expr add_op term*

  term ⟶ *factor* | *term mul_op factor*

  factor ⟶ *id* | *number* | - *factor* | (*expr*)

  add_op ⟶ + | -

  mul_op ⟶ * | /

expr

expr add_op term

expr add_op factor

expr add_op id

expr + *id*

term + *id*

term mul_op factor + *id*

term mul_op id + *id*

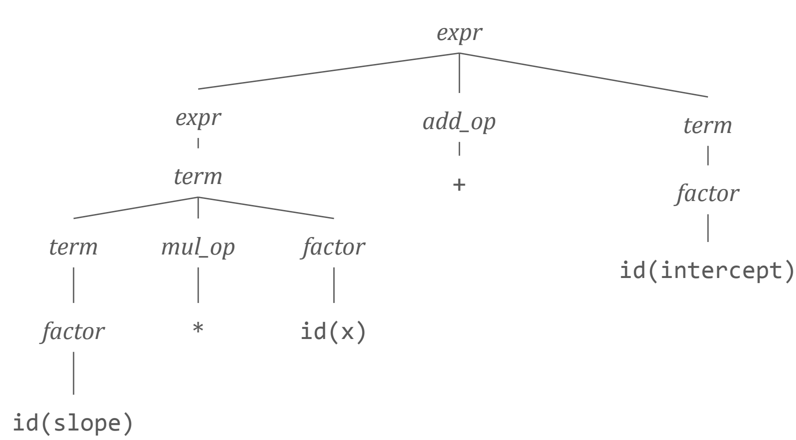
term * *id* + *id*

factor * *id* + *id*

id * *id* + *id*

`slope * x + intercept`

Improved CFG Parse Tree (from the Right)



Improved CFG Example

- What about doing the derivation from the left?
- We still get the correct derivation!
  - The grammar is unambiguous and it honors our intuitive understanding of operator precedence.
  - Multiply and divide are "tighter" than addition and subtraction.
- But what about *associativity*?
  - How are "like" operations grouped?

expr

expr add_op term

term add_op term

term mul_op factor add_op term

factor mul_op factor add_op term

id mul_op factor add_op term

*id * factor add_op term*

*id * id add_op term*

*id * id + term*

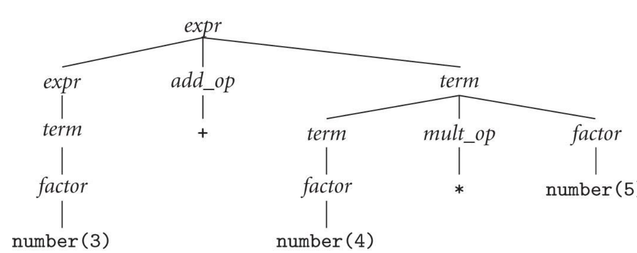
*id * id + factor*

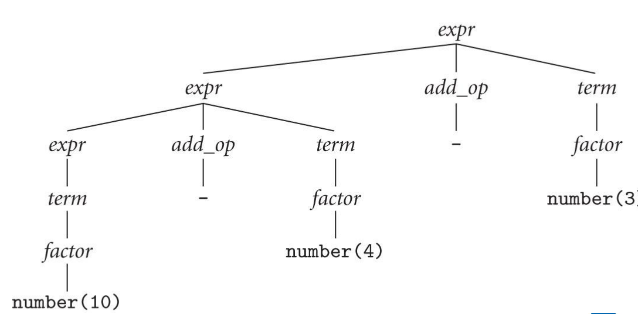
*id * id + id*

`slope * x + intercept`

Improved CFG Example

- This CFG handles *precedence* and *associativity* according to our intuitive expectations.
  - 3+4*5 means
    - 3+(4*5) = 23
    - *not* (3+4)*5 = 35.
  - 10-4-3 means
    - (10-4)-3 = 3
    - *not* 10-(4-3) = 9.





Improved CFG Example

- OK, so it works. But *why* does it work?
- Operator *precedence* is enforced by having nested rules that permit the repetition of only certain operators at each precedence level.
  - *expr* is “looser” than *term* as it occurs higher in the CFG definition.
  - Therefore the *add_op* operators are “looser” than the *mul_op* operators.
- Operator *associativity* is enforced by having the *expr* and *term* rules recurse on their left side rather than the right.
  - Therefore operations group from the left instead of the right.

expr ⟶ *term* | *expr add_op term*

term ⟶ *factor* | *term mul_op factor*

factor ⟶ `id` | `number` | `-` *factor* | `(` *expr* `)`

add_op ⟶ `+` | `-`

mul_op ⟶ `*` | `/`

Improved CFG Example Comments …

- We “fixed” the grammar’s ambiguity by introducing additional productions to keep the operators separate.
- Suppose we want to add *another* operator at yet *another* level of precedence?
  - … and *another* and *another* and … ?
- How far might this have to go? How many levels?

Precedence

- Lots and lots of operators.
- Table goes from highest *precedence* down to lowest.
- Languages try to organize precedence to ease expression writing.

Fortran	Pascal	C	Ada
		`++`, `--` (post-inc., dec.)	
`**`	`not`	`++`, `--` (pre-inc., dec.), `+`, `-` (unary), `&`, `*` (address, contents of), `!`, `~` (logical, bit-wise not)	`abs` (absolute value), `not`, `**`
`*`, `/`	`*`, `/`, `div`, `mod`, `and`	`*` (binary), `/`, `%` (modulo division)	`*`, `/`, `mod`, `rem`
`+`, `-` (unary and binary)	`+`, `-` (unary and binary), `or`	`+`, `-` (binary)	`+`, `-` (unary)
		`<<`, `>>` (left and right bit shift)	`+`, `-` (binary), `&` (concatenation)
`.eq.`, `.ne.`, `.lt.`, `.le.`, `.gt.`, `.ge.` (comparisons)	`<`, `<=`, `>`, `>=`, `=`, `<>`, `IN`	`<`, `<=`, `>`, `>=` (inequality tests)	`=`, `/=` , `<`, `<=`, `>`, `>=`
`.not.`		`==`, `!=` (equality tests)	
		`&` (bit-wise and)	
		`^` (bit-wise exclusive or)	
		`|` (bit-wise inclusive or)	
`.and.`		`&&` (logical and)	`and`, `or`, `xor` (logical operators)
`.or.`		`||` (logical or)	
`.eqv.`, `.neqv.` (logical comparisons)		`?:` (if…then…else)	
		`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `^=`, `|=` (assignment)	
		`,` (sequencing)	

Precedence

- C's ***15*** levels of operator precedence are tricky to remember.
  - But once one knows and uses them effectively, one hardly ever needs to use parentheses for grouping.
  - Watch out! The shift operators (`<<`, `>>`) are lower precedence than the add and subtract (`+`, `-`) operators! `1 << 3 + 1 << 5` is `(1 << (3+1)) << 5`.
- Ada has only ***6*** levels, but the `and` and `or` operators are at the same level of precedence.
  - Hard to believe, but `A or B and C` means `( A or B ) and C`.
- Pascal's ***4*** levels have `and` at a higher precedence than the comparisons.
  - `A < B and C < D` means `A < ( B and C ) < D`, a static error.

WTF?

- Wow.
- Imagine having to make up ***15*** levels of production rules just to get C's operator precedence to work correctly.
  - And then start worrying about *associativity*.

- There has *got* to be a better way to do this!
- Also, how do we get from those *Production Rules* to an actual program?
  - That is, how do we get from a *grammar* to a *parser*?

[*C's Precedence Levels in the Grammar*]

```
primary_expression
  : IDENTIFIER
  | I_CONSTANT | F_CONSTANT | ENUMERATION_CONSTANT
  | STRING_LITERAL | FUNC_NAME
  | '(' expression ')'
  | generic_selection ;

generic_selection
  : GENERIC '(' assignment_expression ',' generic_assoc_list ')' ;

generic_assoc_list
  : generic_association
  | generic_assoc_list ',' generic_association ;

generic_association
  : type_name ':' assignment_expression
  | DEFAULT ':' assignment_expression ;

postfix_expression
  : primary_expression
  | postfix_expression '[' expression ']'
  | postfix_expression '(' ')'
  | postfix_expression '(' argument_expression_list ')'
  | postfix_expression '.' IDENTIFIER
  | postfix_expression PTR_OP IDENTIFIER
  | postfix_expression INC_OP
  | postfix_expression DEC_OP
  | '(' type_name ')' '{' initializer_list '}'
  | '(' type_name ')' '{' initializer_list ',' '}' ;

argument_expression_list
  : assignment_expression
  | argument_expression_list ',' assignment_expression ;

unary_expression
  : postfix_expression
  | INC_OP unary_expression
  | DEC_OP unary_expression
  | unary_operator cast_expression
  | SIZEOF unary_expression
  | SIZEOF '(' type_name ')'
  | ALIGNOF '(' type_name ')' ;

unary_operator
  : '&' | '*' | '+' | '-' | '~' | '!' ;

cast_expression
  : unary_expression
  | '(' type_name ')' cast_expression ;

multiplicative_expression
  : cast_expression
  | multiplicative_expression '*' cast_expression
  | multiplicative_expression '/' cast_expression
  | multiplicative_expression '%' cast_expression ;

additive_expression
  : multiplicative_expression
  | additive_expression '+' multiplicative_expression
  | additive_expression '-' multiplicative_expression ;

shift_expression
  : additive_expression
  | shift_expression LEFT_OP additive_expression
  | shift_expression RIGHT_OP additive_expression ;

relational_expression
  : shift_expression
  | relational_expression '<' shift_expression
  | relational_expression '>' shift_expression
  | relational_expression LE_OP shift_expression
  | relational_expression GE_OP shift_expression ;

equality_expression
  : relational_expression
  | equality_expression EQ_OP relational_expression
  | equality_expression NE_OP relational_expression ;

and_expression
  : equality_expression
  | and_expression '&' equality_expression ;

exclusive_or_expression
  : and_expression
  | exclusive_or_expression '^' and_expression ;

inclusive_or_expression
  : exclusive_or_expression
  | inclusive_or_expression '|' exclusive_or_expression ;

logical_and_expression
  : inclusive_or_expression
  | logical_and_expression AND_OP inclusive_or_expression ;

logical_or_expression
  : logical_and_expression
  | logical_or_expression OR_OP logical_and_expression ;

conditional_expression
  : logical_or_expression
  | logical_or_expression '?' expression ':' conditional_expression ;

assignment_expression
  : conditional_expression
  | unary_expression assignment_operator assignment_expression ;

assignment_operator
  : '=' | MUL_ASSIGN | DIV_ASSIGN | MOD_ASSIGN | ADD_ASSIGN | SUB_ASSIGN
  | LEFT_ASSIGN | RIGHT_ASSIGN | AND_ASSIGN | XOR_ASSIGN | OR_ASSIGN ;

expression
  : assignment_expression
  | expression ',' assignment_expression ;
```

Excerpted from http://www.quut.com/c/ANSI-C-grammar-y.html

Some Context-Free Grammar Theory

- As CFGs are a *formal notation* (as are the *regular expressions*), what can we do with them *mechanically*?
  - That is, *automatically* and *without human intelligence*.
- This is important as it helps us understand for which operations we can construct *tools* that will process a CFG in a useful way.
  - Like getting a *parser* from a *grammar*.

Some Context-Free Grammar Theory

- Unfortunately, many useful properties of CFGs are *undecidable*.
  - ✖ Does a CFG generate *all* possible strings of its *terminals*?
  - ✖ Do two CFGs generate the *same* language?
  - ✖ Does a CFG generate *all* strings another CFG generates?
  - ✖ Does a CFG generate *any* string another CFG generates?
  - ✖ Is the *language* a CFG generates *regular*?
  - ✖ Is a CFG *ambiguous*?
- [Take a *theory* class to understand the *why* of these.]

Context-Free Grammar Theory

- On the other hand, some incredibly useful properties *are* decidable.
  - ✔ Is the language a CFG generates *empty*?
  - ✔ Is the language a CFG generates *finite*?
  - ✔ Is a *CFG* regular? (Either *left* or *right*.)
    - ***Not*** "Is the CFG's *language* regular?" as that's *undecidable*.
  - ✔ Does a CFG *generate* a given string?
    - That is, can we *parse* a given string using a given CFG?
  - ✔ Is a non-terminal of a CFG *reachable*, *productive*, or *nullable*?
  - ✔ Is a CFG `LL(1)`? `LR(1)`? `LR(`*k*`)`?

Predictive Parsing

- So what's that last item, `LL(1)`, mean?
- Essentially, can we *automatically* construct a parser from the CFG that will *efficiently* create an *unambiguous* parse tree from *any* given string?
  - Or tell us the string is *unparseable*.
- `LL(1)` ≡ ***L****eft-to-right* scan of the input, producing a ***L****eftmost* derivation, using only ***1*** token of lookahead.
- This sort of parser never has to *backtrack*.
  - It can parse in time *linear* in the size of the input string.

Predictive Parsing Example

- Let's go back to our improved CFG for expressions.
- It's *unambiguous* already.
- [*id* and *number* will be token categories returned by the scanner.]

expr ⟶ *term* | *expr add_op term*
term ⟶ *factor* | *term mul_op factor*
factor ⟶ *id* | *number* | - *factor* | (*expr*)
add_op ⟶ + | -
mul_op ⟶ * | /

Predictive Parsing Example

- Eliminate explicit alternation to make the production rules more obvious.
- Just make a rule for each case of alternation.

expr ⟶ *expr add_op term*
expr ⟶ *term*
term ⟶ *term mul_op factor*
term ⟶ *factor*
factor ⟶ *id*
factor ⟶ *number*
factor ⟶ - *factor*
factor ⟶ (*expr*)
add_op ⟶ +
add_op ⟶ -
mul_op ⟶ *
mul_op ⟶ /

Predictive Parsing Example

- The *expr* and *term* productions exhibit *left recursion*.
- While the grammar is not ambiguous, it's *non-deterministic* without arbitrary lookahead.
  - `(1+2+3)` vs. `(1+2+3)+4`
  - *expr* ⟶ *term* in the first case but *expr* ⟶ *expr add_op term* in the second.

expr ⟶ *expr add_op term*
expr ⟶ *term*

term ⟶ *term mul_op factor*
term ⟶ *factor*

factor ⟶ *id*
factor ⟶ *number*
factor ⟶ - *factor*
factor ⟶ (*expr*)
add_op ⟶ +
add_op ⟶ -
mul_op ⟶ *
mul_op ⟶ /

Predictive Parsing Example

- *Predictive Parsing* requires that we can *predict* from the next token† which production rule to use.
- We will have to eliminate the left recursion.
  - This can be done mechanically.

expr ⟶ *expr add_op term*
expr ⟶ *term*

term ⟶ *term mul_op factor*
term ⟶ *factor*

factor ⟶ *id*
factor ⟶ *number*
factor ⟶ - *factor*
factor ⟶ (*expr*)
add_op ⟶ +
add_op ⟶ -
mul_op ⟶ *
mul_op ⟶ /

†Actually, it doesn't have to be the *next* token. It could be the *next k* tokens. However, *k* has to be a constant, so ambiguous cases can always be constructed.

[*Eliminating Left Recursion*]

- Rule sets of the form
  - $X \longrightarrow X\alpha$ (where α does not start with X)
  - $X \longrightarrow \beta$

  generate strings of the form $\beta\alpha^*$.
- This can be replaced by
  - $X \longrightarrow \beta X'$
  - $X' \longrightarrow \alpha X'$
  - $X' \longrightarrow \varepsilon$

  moving the recursion from the left side to the right side.

This method isn't restricted to a single production. For example,

$X \rightarrow X\alpha_1$		$X \rightarrow \beta_1 X'$
$X \rightarrow X\alpha_2$		$X \rightarrow \beta_2 X'$
$X \rightarrow X\alpha_3$	$\Rightarrow$	$X \rightarrow \beta_3 X'$
$X \rightarrow \beta_1$		$X' \rightarrow \alpha_1 X'$
$X \rightarrow \beta_2$		$X' \rightarrow \alpha_2 X'$
$X \rightarrow \beta_3$		$X' \rightarrow \alpha_3 X'$
		$X' \rightarrow \varepsilon$

Predictive Parsing Example

- Eliminate left recursion by rewriting the *expr* and *term* production rules.
- We introduce non-terminals *expr1* and *term1* to handle the right recursion.

expr ⟶ *term expr1*
expr1 ⟶ *add_op term expr1*
expr1 ⟶ ε

term ⟶ *factor term1*
term1 ⟶ *mul_op factor term1*
term1 ⟶ ε

factor ⟶ *id*
factor ⟶ *number*
factor ⟶ - *factor*
factor ⟶ (*expr*)
add_op ⟶ +
add_op ⟶ -
mul_op ⟶ *
mul_op ⟶ /

Predictive Parsing Example

- Let's number the final set of production rules to make them easier to refer to as we continue the processing.

1. *expr* ⟶ *term expr1*
2. *expr1* ⟶ *add_op term expr1*
3. *expr1* ⟶ ε
4. *term* ⟶ *factor term1*
5. *term1* ⟶ *mul_op factor term1*
6. *term1* ⟶ ε
7. *factor* ⟶ *id*
8. *factor* ⟶ *number*
9. *factor* ⟶ - *factor*
10. *factor* ⟶ (*expr*)
11. *add_op* ⟶ +
12. *add_op* ⟶ -
13. *mul_op* ⟶ *
14. *mul_op* ⟶ /

Predictive Parsing Example

- To predict which production rule to use, we have to know three items for each non-terminal *X*.

- NULLABLE(*X*): Does *X* ever derive ε?
- FIRST(*X*): Which terminals can appear *first* in a string derived from *X*?
- FOLLOW(*X*): Which terminals can appear *immediately after X*?

1. *expr* ⟶ *term expr1*
2. *expr1* ⟶ *add_op term expr1*
3. *expr1* ⟶ ε
4. *term* ⟶ *factor term1*
5. *term1* ⟶ *mul_op factor term1*
6. *term1* ⟶ ε
7. *factor* ⟶ *id*
8. *factor* ⟶ *number*
9. *factor* ⟶ - *factor*
10. *factor* ⟶ (*expr*)
11. *add_op* ⟶ +
12. *add_op* ⟶ -
13. *mul_op* ⟶ *
14. *mul_op* ⟶ /

[*NULLABLE*]

- Computing NULLABLE(*X*) is a ~~tedious~~ straightforward though iterative process.
- For our grammar, the only NULLABLE items are *expr1* and *term1*.
  - They derive ε directly.
  - No other non-terminal derives either of these non-terminals without also including something else.

```
for each terminal and non-terminal X
  NULLABLE[X] ⟵ False

repeat
  for each rule X ⟶ Y1 Y2 … Yk
    if k = 0
      NULLABLE[X] ⟵ True

    else if Y1 Y2 … Yk are all NULLABLE
      NULLABLE[X] ⟵ True

until NULLABLE didn't change.
```

[*FIRST*]

- As with NULLABLE, computing FIRST(*X*) is an ~~incredibly tedious~~ straightforward though iterative process.
- After iterating, for a while, we find these FIRST sets.

FIRST(add_op) = { '+', '-' }
FIRST(mul_op) = { '*', '/' }
FIRST(factor) = { id, number, '-', '(' }
FIRST(term1) = { '*', '/' }
FIRST(term) = { id, number, '-', '(' }
FIRST(expr1) = { '+', '-' }
FIRST(expr) = { id, number, '-', '(' }

```
for each terminal X
  FIRST[X] ⟵ { X }

for each non-terminal X
  FIRST[X] ⟵ {}

repeat
  for each rule X ⟶ Y1 Y2 … Yk
    for i ⟵ 1 .. k
      if i = 1
        FIRST[X] ⟵ FIRST[X] ∪ FIRST[Yi]

      elif Y1 Y2 … Yi-1 are all NULLABLE
        FIRST[X] ⟵ FIRST[X] ∪ FIRST[Yi]

until FIRST didn't change.
```

[*FOLLOW*]

- Yes, computing FOLLOW(*X*) is a~~n incredibly, mind-bogglingly tedious~~ straightforward though iterative process.
- After iterating for a while, we find these FOLLOW sets.

FOLLOW(add_op) = { id, number, '-', '(' }
FOLLOW(mul_op) = { id, number, '-', '(' }
FOLLOW(factor) = { '*', '/', '+', '-', ')', EOF }
FOLLOW(term1) = { '+', '-', ')', EOF }
FOLLOW(term) = { '+', '-', ')', EOF }
FOLLOW(expr1) = { ')', EOF }
FOLLOW(expr) = { ')', EOF }

```
for each terminal and non-terminal X
  FOLLOW[X] ⟵ {}

repeat
  for each rule X ⟶ Y1 Y2 … Yk
    for i ⟵ 1 .. k
      if i = k
        FOLLOW[Yi] ⟵ FOLLOW[Yi] ∪ FOLLOW[X]

      elif Yi+1 … Yk are all NULLABLE
        FOLLOW[Yi] ⟵ FOLLOW[Yi] ∪ FOLLOW[X]

      for j ⟵ i+1 .. k
        if i+1 = j
          FOLLOW[Yi] ⟵ FOLLOW[Yi] ∪ FIRST[Yj]
        elif Yi+1 … Yj-1 are all NULLABLE
          FOLLOW[Yi] ⟵ FOLLOW[Yi] ∪ FIRST[Yj]

until FOLLOW didn't change.
```

Predictive Parsing Example

- Given the NULLABLE and FIRST and FOLLOW sets, we can now construct a *parse table* for our CFG.
- This table tells us which rule to use when we are trying to parse a given *non-terminal* and are seeing a given *terminal*.
- We have one row for each *non-terminal* and one column for each *terminal*.

```
for each terminal X
  for each non-terminal Y
    TABLE[X,Y] ⟵ {}

for each rule X ⟶ δ
  for each T ∈ FIRST[δ]
    TABLE[X,T] ⟵ TABLE[X,T] ∪ {X⟶δ}

  if NULLABLE[δ]
    for each T ∈ FOLLOW[X]
      TABLE[X,T] ⟵ TABLE[X,T] ∪ {X⟶δ}
```

Predictive Parsing Example

- For our CFG, we get this parse table.

	+	-	*	/	id	number	(	)	EOF
add_op	11	12							
mul_op			13	14					
factor		9			7	8	10		
term1	6	6	5	5				6	6
term		4			4	4	4		
expr1	2	2						3	3
expr		1			1	1	1		

Predictive Parsing Example

- For example, if we are trying to parse an *expr* and we are seeing the `(` token, we use production rule `1`.
- Any box that does not have a rule number indicates an error condition for that non-terminal / terminal combination.
  - E.g., trying to parse a *mul_op* and seeing the `-` token is an error.
- Since no box has more than one rule, we are confirmed in our thinking that the grammar is *unambiguous*.

Predictive Parsing Example

- Great, we have the parse table. How do we get a parser?
- It's not difficult. We just write a routine for each non-terminal.
- The parse table tells us what to do in each case.
  - At right are two of the seven required routines.

```
def EXPR() :
  token = peekToken()

  if token in FIRST_EXPR :
    value = TERM() + EXPR1()

  else :
    print( 'Error!  EXPR saw %s when expecting %s.'
      % ( token, FIRST_EXPR ) )
    raise ValueError

  return value

def EXPR1() :
  token = peekToken()

  if token in [ PLUS, MINUS ] :
    value = ADD_OP() + TERM() + EXPR1()

  elif token in [ RPAREN, EOF ] :
    value = 'ε'

  else :
    print( 'Error!  EXPR1 saw %s when expecting %s.'
      % ( token, FIRST_EXPR1 ) )
    raise ValueError

  return value
```

Predictive Parsing Example

- For each non-terminal, we *peek* at the next token and then take the action recorded in the parse table.
- E.g., if `EXPR()` sees `-`, `id`, `number`, or `(`, it knows it's supposed to use rule 1, *expr* ⟶ *term expr1*.
- It therefore calls `TERM()` and then `EXPR1()`.
- Anything else is an error.


```
def EXPR() :
  token = peekToken()

  if token in FIRST_EXPR :
    value = TERM() + EXPR1()

  else :
    print( 'Error!  EXPR saw %s when expecting %s.'
      % ( token, FIRST_EXPR ) )
    raise ValueError

  return value

def EXPR1() :
  token = peekToken()

  if token in [ PLUS, MINUS ] :
    value = ADD_OP() + TERM() + EXPR1()

  elif token in [ RPAREN, EOF ] :
    value = 'ε'

  else :
    print( 'Error!  EXPR1 saw %s when expecting %s.'
      % ( token, FIRST_EXPR1 ) )
    raise ValueError

  return value
```

Predictive Parsing Example

- Eventually we get to routines that have to *consume* tokens after peeking at them.
- `advanceToken()` moves to the next token.
- `eat()` ensures that the token matches its argument, then moves across it.

```
def FACTOR( indent ) :
  token = peekToken()

  if token in [ ID, NUMBER ] :
    advanceToken()
    value = token

  elif token == MINUS :
    advanceToken()
    value = MINUS + FACTOR()

  elif token == LPAREN :
    advanceToken()
    value = LPAREN + EXPR() + RPAREN
    eat( RPAREN )

  else :
    print( 'Error!  FACTOR saw %s when expecting %s.'
      % ( token, FIRST_FACTOR ) )
    raise ValueError

  return value
```

Predictive Parsing Example

- This is known as a *recursive descent* parser.
- Writing this kind of parser by hand is very common …
  - When the grammar is simple!
- … which is the main reason we spend the time exploring the method.


```
def FACTOR( indent ) :
  token = peekToken()

  if token in [ ID, NUMBER ] :
    advanceToken()
    value = token

  elif token == MINUS :
    advanceToken()
    value = MINUS + FACTOR()

  elif token == LPAREN :
    advanceToken()
    value = LPAREN + EXPR() + RPAREN
    eat( RPAREN )

  else :
    print( 'Error!  FACTOR saw %s when expecting %s.'
      % ( token, FIRST_FACTOR ) )
    raise ValueError

  return value
```

[*Table-Driven Predictive Parsing*]

- By the way, we do not have to hand-code the parser.
- A general method exists that parses *directly* from the table.
- Simpler than using hand-coded routines, but can be inefficient.
- (*If we're going to use a generator, why not a more powerful one?*)

```
push( startSymbol )
token ← readNextToken()

repeat
  X ← pop()

  if terminal( X ) or X = EOF
    if X = token
      token ← readNextToken()
    else
      // Needed a token and current one didn't match.
      error()

  elif TABLE[ X, token ] is empty
    // No rule for this non-terminal / token pair.
    error()

  else
    // Have a rule to use. Push its RHS onto
    // the stack in reverse order.
    for Y in reverse( RHS( TABLE[ X, token ] ) )
      push( Y )

until X = EOF
```

Predictive Parsing Example

- And, the parser *really*, *really* works!

```
#----------
Trying ['number', '-', 'number', 'EOF'] ...
FACTOR:     number
TERM1 :     ε
TERM  :   numberε
ADD_OP:     -
FACTOR:       number
TERM1 :       ε
TERM  :     numberε
EXPR1 :     ε
EXPR1 :   -numberεε
EXPR  : numberε-numberεε

Success!
  numberε-numberεε
```

```
#----------
Trying ['number', '+', 'number',
        '*', 'number', 'EOF'] ...
FACTOR:     number
TERM1 :     ε
TERM  :   numberε
ADD_OP:     +
FACTOR:       number
MUL_OP:         *
FACTOR:         number
TERM1 :         ε
TERM1 :       *numberε
TERM  :     number*numberε
EXPR1 :     ε
EXPR1 :   +number*numberεε
EXPR  : numberε+number*numberεε

Success!
  numberε+number*numberεε
```

```
#----------
Trying ['number', '/', 'number',
        '-', 'number', 'EOF'] ...
FACTOR:     number
MUL_OP:       /
FACTOR:       number
TERM1 :       ε
TERM1 :     /numberε
TERM  :   number/numberε
ADD_OP:     -
FACTOR:       number
TERM1 :       ε
TERM  :     numberε
EXPR1 :     ε
EXPR1 :   -numberεε
EXPR  : number/numberε-numberεε

Success!
  number/numberε-numberεε
```

Predictive Parsing Example

- Even with *errors* and *complex* cases!

```
#----------
Trying [')', 'EOF'] ...
Error!  EXPR saw ) when expecting ['-', 'id', 'number', '('].
Parse error!

#----------
Trying ['-', '*', 'EOF'] ...
Error!  FACTOR saw * when expecting ['-', 'id', 'number', '('].
Parse error!

#----------
Trying ['number', '-', '+', 'EOF'] ...
FACTOR:     number
TERM1 :     ε
TERM  :   numberε
ADD_OP:     -
Error!  TERM saw + when expecting ['-', 'id', 'number', '('].
Parse error!
```

```
#----------
Trying ['(', 'number', '+', 'number',
        ')', '*', 'number', 'EOF'] ...
FACTOR:         number
TERM1 :           ε
TERM  :         numberε
ADD_OP:           +
FACTOR:             number
TERM1 :               ε
TERM  :           numberε
EXPR1 :           ε
EXPR1 :         +numberεε
EXPR  :       numberε+numberεε
FACTOR:     (numberε+numberεε)
MUL_OP:         *
FACTOR:       number
TERM1 :       ε
TERM1 :     *numberε
TERM  :   (numberε+numberεε)*numberε
EXPR1 :   ε
EXPR  : (numberε+numberεε)*numberεε

Success!
  (numberε+numberεε)*numberεε
```

Predictive Parsing Example

- So what are those ε characters?
- That's where *expr1* or *term1* derived the empty string (ε).
  - Rules 3 and 6.
- Remember, we got rid of *left* recursion by converting it to *right* recursion.

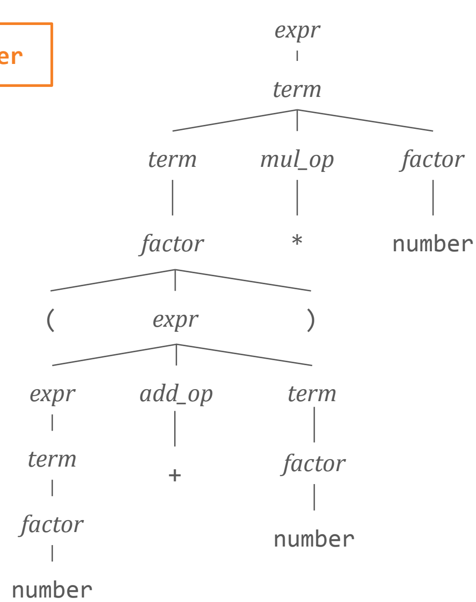
1. *expr* ⟶ *term expr1*
2. *expr1* ⟶ *add_op term expr1*
3. *expr1* ⟶ ε
4. *term* ⟶ *factor term1*
5. *term1* ⟶ *mul_op factor term1*
6. *term1* ⟶ ε
7. *factor* ⟶ *id*
8. *factor* ⟶ *number*
9. *factor* ⟶ - *factor*
10. *factor* ⟶ (*expr*)
11. *add_op* ⟶ +
12. *add_op* ⟶ -
13. *mul_op* ⟶ *
14. *mul_op* ⟶ /

Predictive Parsing Example

- Well, that was *painful*.
- Also, the eventual parsing we got from the `LL(1)` method is not exactly what we might have expected.
  - We had to change our grammar to eliminate left recursion.
- Did you notice the weird way the productions came out?

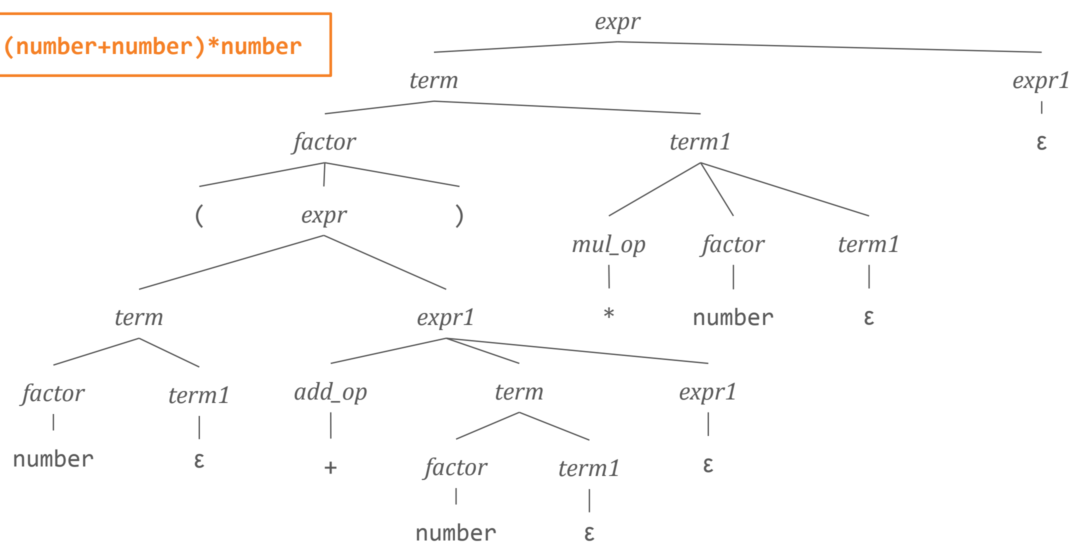
Expected Parse Tree for (number+number)*number

`(number+number)*number`



Actual Parse Tree

`(number+number)*number`



Predictive Parsing Example

- Pretty awful, huh?
- Well, no matter, we can always fix it up in the action routines.
- *But* … do we *want* to?
- ***Again***, there has *got* to be a better way!

LR(k) Parsing

- There is a better way.
- It's called **LR(k)** parsing.

- LR(k) ≡ ***L**eft-to-right* scan of the input, producing a ***R**ightmost* derivation, using ***k*** tokens of lookahead.

LL(1) vs LR(k) Parsing

- The LL(1) Predictive Parsing that we did was *top-down*.
  - We used the production rules from *left* to *right*.

- LR(k) parsing is *bottom-up*.
  - We use the production rules from *right* to *left*.

- As with the table-driven predictive parsing, we use a *stack* with LR(k) parsing.
  - But we use the stack in a more sophisticated way.

LR(k) Parsing

- We use a *stack* and the *input stream* to decide what to do.
- The first k tokens in the input stream form the *lookahead*.

- Depending on the stack and the lookahead, the parser takes one of two possible actions:

  > *Shift* : Push the first input token onto the stack.

  > *Reduce* : Select some grammar rule, e.g., $X \longrightarrow A\ B\ C$; pop C, B, A off the stack; and push X onto the stack.

LR(k) Parsing

- Initially the *stack* is empty and the *lookahead* is the first k tokens from the *input stream*.

- If the parser makes it to the point where it can *shift* the *EOF* marker onto the stack, it has *accepted* the input as valid.

LR(k) Parsing

- The decision to *reduce* is made based on the top few items on the stack, which need to match the RHS of a production rule. Three cases can occur.

① The top few items do not match the RHS of *any* rule of the grammar.

- This is *easy* to resolve.
- Just *shift* another token from the input stream onto the stack and look again.

LR(k) Parsing

② The top few items match the RHS of *one* rule of the grammar.

- This is *easy* to resolve as well.
- Just *reduce* by popping off the stack items corresponding to the RHS of the rule.
  - The popped items are known as a *handle*.
- Then push the LHS of the rule (a non-terminal) onto the stack.

LR(k) Parsing

③ The top few items match the RHS of *multiple* rules of the grammar.

- This is *not* so easy to resolve.
- The grammar is *ambiguous* so we don't know which reduction to make.
- This is known as a *reduce/reduce* ambiguity.

LR(k) Parsing

- So how do we find such issues in the grammar?
  - And are there other kinds of problems?

- We have to produce the `ACTION` and `GOTO` tables and see what problems show up along the way.
- Producing these tables is similar to what we did with *Predictive Parsing*.
- Aside from *reduce/reduce* conflicts, there are *shift/reduce* conflicts, which occur when it's not clear whether to shift a token onto the stack or to reduce what's already there.

LR(k) Parsing

- We analyze the grammar by building *item sets* based on the rules and how far we've gotten into a parse.
- For example, the *item*
  - `E -> E . + T`

  indicates we using the rule E -> E + T, we have seen an E, and that we are expecting a '+' token next.
- An *item set* is a collection of *items* that are all in the same state.

Analyzing an Example Grammar

- We'll start with a simplified version of the classic expression grammar.
  - Only one add op, +.
  - Only one mul op, *.
- It's unambiguous, so it shouldn't have any analysis problems.

```
S → E
E → E + T
E → T
T → T * F
T → F
F → id
F → ( E )
```

Analyzing an Example Grammar

```
Grammar

Rule 0     S' -> S
Rule 1     S -> E
Rule 2     E -> E + T
Rule 3     E -> T
Rule 4     T -> T * F
Rule 5     T -> F
Rule 6     F -> id
Rule 7     F -> ( E )
```

```
Terminals, with rules where they appear

(                : 7
*                : 4
+                : 2
)                : 7
error            :
id               : 6

Nonterminals, with rules where they appear

E                : 1 2 7
F                : 4 5
S                : 0
T                : 2 3 4
```

Analyzing an Example Grammar

```
state 0

    (0) S' -> . S
    (1) S -> . E
    (2) E -> . E + T
    (3) E -> . T
    (4) T -> . T * F
    (5) T -> . F
    (6) F -> . id
    (7) F -> . ( E )

    id              shift and go to state 5
    (               shift and go to state 6

    S                              shift and go to state 1
    E                              shift and go to state 2
    T                              shift and go to state 3
    F                              shift and go to state 4
```

Analyzing an Example Grammar

```
state 1

    (0) S' -> S .



state 2

    (1) S -> E .
    (2) E -> E . + T

    $end            reduce using rule 1 (S -> E .)
    +               shift and go to state 7
```

Analyzing an Example Grammar

```
state 3

    (3) E -> T .
    (4) T -> T . * F

    +        reduce using rule 3 (E -> T .)
    $end     reduce using rule 3 (E -> T .)
    )        reduce using rule 3 (E -> T .)
    *        shift and go to state 8


state 4

    (5) T -> F .

    *        reduce using rule 5 (T -> F .)
    +        reduce using rule 5 (T -> F .)
    $end     reduce using rule 5 (T -> F .)
    )        reduce using rule 5 (T -> F .)
```

Analyzing an Example Grammar

```
state 5

    (6) F -> id .

    *        reduce using rule 6 (F -> id .)
    +        reduce using rule 6 (F -> id .)
    $end     reduce using rule 6 (F -> id .)
    )        reduce using rule 6 (F -> id .)
```

Analyzing an Example Grammar

```
state 6

    (7) F -> ( . E )
    (2) E -> . E + T
    (3) E -> . T
    (4) T -> . T * F
    (5) T -> . F
    (6) F -> . id
    (7) F -> . ( E )

    id          shift and go to state 5
    (           shift and go to state 6

    E                                   shift and go to state 9
    T                                   shift and go to state 3
    F                                   shift and go to state 4
```

Analyzing an Example Grammar

```
state 7

    (2) E -> E + . T
    (4) T -> . T * F
    (5) T -> . F
    (6) F -> . id
    (7) F -> . ( E )

    id          shift and go to state 5
    (           shift and go to state 6

    T                                   shift and go to state 10
    F                                   shift and go to state 4
```

Analyzing an Example Grammar

```
state 8

    (4) T -> T * . F
    (6) F -> . id
    (7) F -> . ( E )

    id          shift and go to state 5
    (           shift and go to state 6

    F                                   shift and go to state 11

state 9

    (7) F -> ( E . )
    (2) E -> E . + T

    )           shift and go to state 12
    +           shift and go to state 7
```

Analyzing an Example Grammar

```
state 10

    (2) E -> E + T .
    (4) T -> T . * F

    +               reduce using rule 2 (E -> E + T .)
    $end            reduce using rule 2 (E -> E + T .)
    )               reduce using rule 2 (E -> E + T .)
    *               shift and go to state 8


state 11

    (4) T -> T * F .

    *               reduce using rule 4 (T -> T * F .)
    +               reduce using rule 4 (T -> T * F .)
    $end            reduce using rule 4 (T -> T * F .)
    )               reduce using rule 4 (T -> T * F .)
```

Analyzing an Example Grammar

```
state 12

    (7) F -> ( E ) .

    *               reduce using rule 7 (F -> ( E ) .)
    +               reduce using rule 7 (F -> ( E ) .)
    $end            reduce using rule 7 (F -> ( E ) .)
    )               reduce using rule 7 (F -> ( E ) .)
```

Analyzing an Example Grammar

ACTION Table

```
0: {'id': 5, '(': 6},
1: {'$': 0},
2: {'$': -1, '+': 7},
3: {'$': -3, '+': -3, ')': -3, '*': 8},
4: {'$': -5, '+': -5, ')': -5, '*': -5},
5: {'$': -6, '+': -6, ')': -6, '*': -6},
6: {'id': 5, '(': 6},
7: {'id': 5, '(': 6},
8: {'id': 5, '(': 6},
9: {'+': 7, ')': 12},
10: {'$': -2, '+': -2, ')': -2, '*': 8},
11: {'$': -4, '+': -4, ')': -4, '*': -4},
12: {'$': -7, '+': -7, ')': -7, '*': -7},
```

GOTO Table

```
0: {'S': 1, 'E': 2, 'T': 3, 'F': 4},
6: {'E': 9, 'T': 3, 'F': 4},
7: {'T': 10, 'F': 4},
8: {'F': 11}
```

Analyzing an Example Grammar

- This example grammar is straightforward.
  - There are no ambiguities.
  - No shift/reduce conflicts.
  - No reduce/reduce conflicts.
- How boring.
- Let’s try another grammar, this time with a problem.

Grammar with a Shift/Reduce Conflict

- Our grammar is simple, merely demonstrating how one might represent an `if` statement.
- We have two kinds of `if` statement, one with an `else` clause and one without.
- The ambiguity is obvious. Let's see what the analysis reports.

```
S → if E then S else S
S → if E then S
S → E
E → ID
```

Grammar with a Shift/Reduce Conflict

```
Grammar

Rule 0     S' -> S
Rule 1     S -> IF E THEN S ELSE S
Rule 2     S -> IF E THEN S
Rule 3     S -> E
Rule 4     E -> ID
```

```
Terminals, with rules where they appear

ELSE                 : 1
ID                   : 4
IF                   : 1 2
THEN                 : 1 2
error                :

Nonterminals, with rules where they appear

E                    : 1 2 3
S                    : 1 1 2 0
```

Grammar with a Shift/Reduce Conflict

```
state 0

    (0) S' -> . S
    (1) S -> . IF E THEN S ELSE S
    (2) S -> . IF E THEN S
    (3) S -> . E
    (4) E -> . ID

    IF              shift and go to state 2
    ID              shift and go to state 4

    S                              shift and go to state 1
    E                              shift and go to state 3
```

Grammar with a Shift/Reduce Conflict

```
state 1

    (0) S' -> S .



state 2

    (1) S -> IF . E THEN S ELSE S
    (2) S -> IF . E THEN S
    (4) E -> . ID

    ID              shift and go to state 4

    E                              shift and go to state 5
```

Grammar with a Shift/Reduce Conflict

```
state 3

    (3) S -> E .

    $end            reduce using rule 3 (S -> E .)
    ELSE            reduce using rule 3 (S -> E .)


state 4

    (4) E -> ID .

    $end            reduce using rule 4 (E -> ID .)
    THEN            reduce using rule 4 (E -> ID .)
    ELSE            reduce using rule 4 (E -> ID .)
```

Grammar with a Shift/Reduce Conflict

```
state 5

    (1) S -> IF E . THEN S ELSE S
    (2) S -> IF E . THEN S

    THEN            shift and go to state 6

state 6

    (1) S -> IF E THEN . S ELSE S
    (2) S -> IF E THEN . S
    (1) S -> . IF E THEN S ELSE S
    (2) S -> . IF E THEN S
    (3) S -> . E
    (4) E -> . ID

    IF              shift and go to state 2
    ID              shift and go to state 4

    E                              shift and go to state 3
    S                              shift and go to state 7
```

Grammar with a Shift/Reduce Conflict

```
state 7

    (1) S -> IF E THEN S . ELSE S
    (2) S -> IF E THEN S .

  ! shift/reduce conflict for ELSE resolved as shift
    ELSE            shift and go to state 8
    $end            reduce using rule 2 (S -> IF E THEN S .)

  ! ELSE            [ reduce using rule 2 (S -> IF E THEN S .) ]
```

Grammar with a Shift/Reduce Conflict

```
state 8

    (1) S -> IF E THEN S ELSE . S
    (1) S -> . IF E THEN S ELSE S
    (2) S -> . IF E THEN S
    (3) S -> . E
    (4) E -> . ID

    IF              shift and go to state 2
    ID              shift and go to state 4

    E                              shift and go to state 3
    S                              shift and go to state 9
```

Grammar with a Shift/Reduce Conflict

```
state 9

    (1) S -> IF E THEN S ELSE S .

    $end            reduce using rule 1 (S -> IF E THEN S ELSE S .)
    ELSE            reduce using rule 1 (S -> IF E THEN S ELSE S .)

WARNING:
WARNING: Conflicts:
WARNING:
WARNING: shift/reduce conflict for ELSE in state 7 resolved as shift
```

Grammar with a Shift/Reduce Conflict

ACTION Table

```
0: { 'IF': 2, 'ID': 4 }
1: { '$end': 0 }
2: { 'ID': 4 }
3: { '$end': -3, 'ELSE': -3 }
4: { '$end': -4, 'ELSE': -4, 'THEN': -4 }
5: { 'THEN': 6 }
6: { 'IF': 2, 'ID': 4 }
7: { '$end': -2, 'ELSE': 8 }
8: { 'IF': 2, 'ID': 4 }
9: { '$end': -1, 'ELSE': -1 }
```

GOTO Table

```
0: { 'S': 1, 'E': 3 }
2: { 'E':  5 }
6: { 'S': 7, 'E': 3 }
8: { 'S': 9, 'E': 3 }
```

Grammar with a Shift/Reduce Conflict

- OK, so there was a shift/reduce conflict.
- What do we do about that?
- First, we have to decide what the correct, expected parse is supposed to be.
  - Normally, this would be to have the `else` clause bind with the closest available `if` that doesn't already have an `else` clause.
- Guess what? That's exactly what happens if we just resolve the shift/reduce conflict by *always* shifting!
  - That's the *default* behavior of the parser generator anyway.

Grammar with a Shift/Reduce Conflict

- So that wasn't very exciting after all.
- We could have rewritten the grammar so that this shift/reduce conflict doesn't occur, but why?
  - Or, we could have introduced a terminating marker for the `if` statement (as, e.g., Ada has, `end if`).
- OK, let's try another grammar, this one with *lots* of shift/reduce conflicts.

Another Grammar with Shift/Reduce Conflicts

- Another expression grammar, this time with multiple binary operators and no indication of precedence or associativity.
- The ambiguity is obvious. Let's see what the analysis reports.

```
E ⟶ INTEGER
E ⟶ E + E
E ⟶ E - E
E ⟶ E * E
E ⟶ E / E
E ⟶ ( E )
```

Another Grammar with Shift/Reduce Conflicts

```
Grammar

Rule 0     S' -> E
Rule 1     E -> INTEGER
Rule 2     E -> E PLUS E
Rule 3     E -> E MINUS E
Rule 4     E -> E MULTIPLY E
Rule 5     E -> E DIVIDE E
Rule 6     E -> LPAREN E RPAREN
```

```
Terminals, with rules where they appear

DIVIDE               : 5
INTEGER              : 1
LPAREN               : 6
MINUS                : 3
MULTIPLY             : 4
PLUS                 : 2
RPAREN               : 6
error                :

Nonterminals, with rules where they appear

E                    : 2 2 3 3 4 4 5 5 6 0
```

Another Grammar with Shift/Reduce Conflicts

```
state 0

    (0) S' -> . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                              shift and go to state 1
```

Another Grammar with Shift/Reduce Conflicts

```
state 1

    (0) S' -> E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7
```

Another Grammar with Shift/Reduce Conflicts

```
state 2

    (1) E -> INTEGER .

    PLUS            reduce using rule 1 (E -> INTEGER .)
    MINUS           reduce using rule 1 (E -> INTEGER .)
    MULTIPLY        reduce using rule 1 (E -> INTEGER .)
    DIVIDE          reduce using rule 1 (E -> INTEGER .)
    $end            reduce using rule 1 (E -> INTEGER .)
    RPAREN          reduce using rule 1 (E -> INTEGER .)
```

Another Grammar with Shift/Reduce Conflicts

```
state 3

    (6) E -> LPAREN . E RPAREN
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                              shift and go to state 8
```

Another Grammar with Shift/Reduce Conflicts

```
state 4

    (2) E -> E PLUS . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                               shift and go to state 9
```

Another Grammar with Shift/Reduce Conflicts

```
state 5

    (3) E -> E MINUS . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                               shift and go to state 10
```

Another Grammar with Shift/Reduce Conflicts

```
state 6

    (4) E -> E MULTIPLY . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                              shift and go to state 11
```

Another Grammar with Shift/Reduce Conflicts

```
state 7

    (5) E -> E DIVIDE . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                              shift and go to state 12
```

Another Grammar with Shift/Reduce Conflicts

```
state 8

    (6) E -> LPAREN E . RPAREN
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    RPAREN          shift and go to state 13
    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7
```

Another Grammar with Shift/Reduce Conflicts

```
state 9

    (2) E -> E PLUS E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

  ! shift/reduce conflict for PLUS resolved as shift
  ! shift/reduce conflict for MINUS resolved as shift
  ! shift/reduce conflict for MULTIPLY resolved as shift
  ! shift/reduce conflict for DIVIDE resolved as shift
    $end            reduce using rule 2 (E -> E PLUS E .)
    RPAREN          reduce using rule 2 (E -> E PLUS E .)
    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7

  ! PLUS            [ reduce using rule 2 (E -> E PLUS E .) ]
  ! MINUS           [ reduce using rule 2 (E -> E PLUS E .) ]
  ! MULTIPLY        [ reduce using rule 2 (E -> E PLUS E .) ]
  ! DIVIDE          [ reduce using rule 2 (E -> E PLUS E .) ]
```

Another Grammar with Shift/Reduce Conflicts

```
state 10

    (3) E -> E MINUS E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

  ! shift/reduce conflict for PLUS resolved as shift
  ! shift/reduce conflict for MINUS resolved as shift
  ! shift/reduce conflict for MULTIPLY resolved as shift
  ! shift/reduce conflict for DIVIDE resolved as shift
    $end            reduce using rule 3 (E -> E MINUS E .)
    RPAREN          reduce using rule 3 (E -> E MINUS E .)
    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7

  ! PLUS            [ reduce using rule 3 (E -> E MINUS E .) ]
  ! MINUS           [ reduce using rule 3 (E -> E MINUS E .) ]
  ! MULTIPLY        [ reduce using rule 3 (E -> E MINUS E .) ]
  ! DIVIDE          [ reduce using rule 3 (E -> E MINUS E .) ]
```

Another Grammar with Shift/Reduce Conflicts

```
state 11

    (4) E -> E MULTIPLY E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

  ! shift/reduce conflict for PLUS resolved as shift
  ! shift/reduce conflict for MINUS resolved as shift
  ! shift/reduce conflict for MULTIPLY resolved as shift
  ! shift/reduce conflict for DIVIDE resolved as shift
    $end            reduce using rule 4 (E -> E MULTIPLY E .)
    RPAREN          reduce using rule 4 (E -> E MULTIPLY E .)
    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7

  ! PLUS            [ reduce using rule 4 (E -> E MULTIPLY E .) ]
  ! MINUS           [ reduce using rule 4 (E -> E MULTIPLY E .) ]
  ! MULTIPLY        [ reduce using rule 4 (E -> E MULTIPLY E .) ]
  ! DIVIDE          [ reduce using rule 4 (E -> E MULTIPLY E .) ]
```

Another Grammar with Shift/Reduce Conflicts

```
state 12

    (5) E -> E DIVIDE E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

  ! shift/reduce conflict for PLUS resolved as shift
  ! shift/reduce conflict for MINUS resolved as shift
  ! shift/reduce conflict for MULTIPLY resolved as shift
  ! shift/reduce conflict for DIVIDE resolved as shift
    $end            reduce using rule 5 (E -> E DIVIDE E .)
    RPAREN          reduce using rule 5 (E -> E DIVIDE E .)
    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7

  ! PLUS            [ reduce using rule 5 (E -> E DIVIDE E .) ]
  ! MINUS           [ reduce using rule 5 (E -> E DIVIDE E .) ]
  ! MULTIPLY        [ reduce using rule 5 (E -> E DIVIDE E .) ]
  ! DIVIDE          [ reduce using rule 5 (E -> E DIVIDE E .) ]
```

Another Grammar with Shift/Reduce Conflicts

```
state 13

    (6) E -> LPAREN E RPAREN .

    PLUS            reduce using rule 6 (E -> LPAREN E RPAREN .)
    MINUS           reduce using rule 6 (E -> LPAREN E RPAREN .)
    MULTIPLY        reduce using rule 6 (E -> LPAREN E RPAREN .)
    DIVIDE          reduce using rule 6 (E -> LPAREN E RPAREN .)
    $end            reduce using rule 6 (E -> LPAREN E RPAREN .)
    RPAREN          reduce using rule 6 (E -> LPAREN E RPAREN .)
```

Another Grammar with Shift/Reduce Conflicts

```
WARNING:
WARNING: Conflicts:
WARNING:
WARNING: shift/reduce conflict for PLUS in state 9 resolved as shift
WARNING: shift/reduce conflict for MINUS in state 9 resolved as shift
WARNING: shift/reduce conflict for MULTIPLY in state 9 resolved as shift
WARNING: shift/reduce conflict for DIVIDE in state 9 resolved as shift
WARNING: shift/reduce conflict for PLUS in state 10 resolved as shift
WARNING: shift/reduce conflict for MINUS in state 10 resolved as shift
WARNING: shift/reduce conflict for MULTIPLY in state 10 resolved as shift
WARNING: shift/reduce conflict for DIVIDE in state 10 resolved as shift
WARNING: shift/reduce conflict for PLUS in state 11 resolved as shift
WARNING: shift/reduce conflict for MINUS in state 11 resolved as shift
WARNING: shift/reduce conflict for MULTIPLY in state 11 resolved as shift
WARNING: shift/reduce conflict for DIVIDE in state 11 resolved as shift
WARNING: shift/reduce conflict for PLUS in state 12 resolved as shift
WARNING: shift/reduce conflict for MINUS in state 12 resolved as shift
WARNING: shift/reduce conflict for MULTIPLY in state 12 resolved as shift
WARNING: shift/reduce conflict for DIVIDE in state 12 resolved as shift
```

Another Grammar with Shift/Reduce Conflicts

- This grammar has ***16*** shift/reduce conflicts (4×4 because of the operators) and we ***can't*** just always shift.
  - This would cause precedence problems in some cases.
    - E.g., + might happen before *.
- The key word here is *precedence*. We could rewrite the grammar to separate precedence levels (as in that C excerpt), but why?
- Most compiler-compilers however allow the specification of operator precedence (and associativity).

Specifying Precedence and Associativity in `bison`

- Use the `%left` directive.
  - In the *Definitions* section of the `.y` file.
- Here we state that `TOKEN_PLUS` and `TOKEN_MINUS` are lower precedence than `TOKEN_SLASH` and `TOKEN_STAR`.
  - They are all *left-to-right* associative.
- There are also `%right` and `%nonassoc` directives for those kinds of operators.

lower precedence

```
%left TOKEN_MINUS TOKEN_PLUS
%left TOKEN_SLASH TOKEN_STAR
```

higher precedence

Another Grammar …

- Inserting the directives and rerunning `bison` results in *no* shift/reduce conflicts at all.
- We still have the same number / structure of states, but now we *know* when to shift and when to reduce so that precedence and associativity are honored.

Another Grammar …

```
Grammar

Rule 0     S' -> E
Rule 1     E -> INTEGER
Rule 2     E -> E PLUS E
Rule 3     E -> E MINUS E
Rule 4     E -> E MULTIPLY E
Rule 5     E -> E DIVIDE E
Rule 6     E -> LPAREN E RPAREN
```

```
Terminals, with rules where they appear

DIVIDE               : 5
INTEGER              : 1
LPAREN               : 6
MINUS                : 3
MULTIPLY             : 4
PLUS                 : 2
RPAREN               : 6
error                :

Nonterminals, with rules where they appear

E                    : 2 2 3 3 4 4 5 5 6 0
```

Another Grammar …

```
state 0

    (0) S' -> . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                              shift and go to state 1
```

Another Grammar …

```
state 1

    (0) S' -> E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7
```

Another Grammar …

```
state 2

    (1) E -> INTEGER .

    PLUS            reduce using rule 1 (E -> INTEGER .)
    MINUS           reduce using rule 1 (E -> INTEGER .)
    MULTIPLY        reduce using rule 1 (E -> INTEGER .)
    DIVIDE          reduce using rule 1 (E -> INTEGER .)
    $end            reduce using rule 1 (E -> INTEGER .)
    RPAREN          reduce using rule 1 (E -> INTEGER .)
```

Another Grammar …

```
state 3

    (6) E -> LPAREN . E RPAREN
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                               shift and go to state 8
```

Another Grammar …

```
state 4

    (2) E -> E PLUS . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                               shift and go to state 9
```

Another Grammar …

```
state 5

    (3) E -> E MINUS . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                              shift and go to state 10
```

Another Grammar …

```
state 6

    (4) E -> E MULTIPLY . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                              shift and go to state 11
```

Another Grammar …

```
state 7

    (5) E -> E DIVIDE . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                              shift and go to state 12
```

Another Grammar …

```
state 8

    (6) E -> LPAREN E . RPAREN
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    RPAREN          shift and go to state 13
    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7
```

Another Grammar …

```
state 9

    (2) E -> E PLUS E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    PLUS            reduce using rule 2 (E -> E PLUS E .)
    MINUS           reduce using rule 2 (E -> E PLUS E .)
    $end            reduce using rule 2 (E -> E PLUS E .)
    RPAREN          reduce using rule 2 (E -> E PLUS E .)
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7

  ! MULTIPLY        [ reduce using rule 2 (E -> E PLUS E .) ]
  ! DIVIDE          [ reduce using rule 2 (E -> E PLUS E .) ]
  ! PLUS            [ shift and go to state 4 ]
  ! MINUS           [ shift and go to state 5 ]
```

Another Grammar …

```
state 10

    (3) E -> E MINUS E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    PLUS            reduce using rule 3 (E -> E MINUS E .)
    MINUS           reduce using rule 3 (E -> E MINUS E .)
    $end            reduce using rule 3 (E -> E MINUS E .)
    RPAREN          reduce using rule 3 (E -> E MINUS E .)
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7

  ! MULTIPLY        [ reduce using rule 3 (E -> E MINUS E .) ]
  ! DIVIDE          [ reduce using rule 3 (E -> E MINUS E .) ]
  ! PLUS            [ shift and go to state 4 ]
  ! MINUS           [ shift and go to state 5 ]
```

Another Grammar …

```
state 11

    (4) E -> E MULTIPLY E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    PLUS            reduce using rule 4 (E -> E MULTIPLY E .)
    MINUS           reduce using rule 4 (E -> E MULTIPLY E .)
    MULTIPLY        reduce using rule 4 (E -> E MULTIPLY E .)
    DIVIDE          reduce using rule 4 (E -> E MULTIPLY E .)
    $end            reduce using rule 4 (E -> E MULTIPLY E .)
    RPAREN          reduce using rule 4 (E -> E MULTIPLY E .)

  ! PLUS            [ shift and go to state 4 ]
  ! MINUS           [ shift and go to state 5 ]
  ! MULTIPLY        [ shift and go to state 6 ]
  ! DIVIDE          [ shift and go to state 7 ]
```

Another Grammar …

```
state 12

    (5) E -> E DIVIDE E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    PLUS            reduce using rule 5 (E -> E DIVIDE E .)
    MINUS           reduce using rule 5 (E -> E DIVIDE E .)
    MULTIPLY        reduce using rule 5 (E -> E DIVIDE E .)
    DIVIDE          reduce using rule 5 (E -> E DIVIDE E .)
    $end            reduce using rule 5 (E -> E DIVIDE E .)
    RPAREN          reduce using rule 5 (E -> E DIVIDE E .)

  ! PLUS            [ shift and go to state 4 ]
  ! MINUS           [ shift and go to state 5 ]
  ! MULTIPLY        [ shift and go to state 6 ]
  ! DIVIDE          [ shift and go to state 7 ]
```

Another Grammar …

```
state 13

    (6) E -> LPAREN E RPAREN .

    PLUS            reduce using rule 6 (E -> LPAREN E RPAREN .)
    MINUS           reduce using rule 6 (E -> LPAREN E RPAREN .)
    MULTIPLY        reduce using rule 6 (E -> LPAREN E RPAREN .)
    DIVIDE          reduce using rule 6 (E -> LPAREN E RPAREN .)
    $end            reduce using rule 6 (E -> LPAREN E RPAREN .)
    RPAREN          reduce using rule 6 (E -> LPAREN E RPAREN .)
```

Another Grammar …

ACTION Table

```
0: {'INTEGER': 2, 'LPAREN': 3}
1: {'$end': 0, 'PLUS': 4, 'MINUS': 5,
   'MULTIPLY': 6, 'DIVIDE': 7}
2: {'$end': -1, 'PLUS': -1, 'MINUS': -1,
   'MULTIPLY': -1, 'DIVIDE': -1, 'RPAREN': -1}
3: {'INTEGER': 2, 'LPAREN': 3}
4: {'INTEGER': 2, 'LPAREN': 3}
5: {'INTEGER': 2, 'LPAREN': 3}
6: {'INTEGER': 2, 'LPAREN': 3}
7: {'INTEGER': 2, 'LPAREN': 3}
8: {'PLUS': 4, 'MINUS': 5, 'MULTIPLY': 6,
   'DIVIDE': 7, 'RPAREN': 13}
9: {'$end': -2, 'PLUS': -2, 'MINUS': -2,
   'MULTIPLY': 6, 'DIVIDE': 7, 'RPAREN': -2}
10: {'$end': -3, 'PLUS': -3, 'MINUS': -3,
    'MULTIPLY': 6, 'DIVIDE': 7, 'RPAREN': -3}
11: {'$end': -4, 'PLUS': -4, 'MINUS': -4,
    'MULTIPLY': -4, 'DIVIDE': -4, 'RPAREN': -4}
12: {'$end': -5, 'PLUS': -5, 'MINUS': -5,
    'MULTIPLY': -5, 'DIVIDE': -5, 'RPAREN': -5}
13: {'$end': -6, 'PLUS': -6, 'MINUS': -6,
    'MULTIPLY': -6, 'DIVIDE': -6, 'RPAREN': -6}
```

GOTO Table

```
0: {'E': 1}
3: {'E': 8}
4: {'E': 9}
5: {'E': 10}
6: {'E': 11}
7: {'E': 12}
```

Another Grammar …

- Now there are no conflicts, precedence and associativity are properly honored, and the parse tree will be as expected.

- However, *shift/reduce* is not the only kind of conflict that can occur.
- There's also *reduce/reduce*.

Grammar with a Reduce/Reduce Conflict

- A contrived example grammar, but this is the kind of rule structure that leads to reduce/reduce conflicts.

- The ambiguity is obvious. Let's see what the analysis reports.

```
A → B c d
A → E c f
B → x y
E → x y
```

Grammar with a Reduce/Reduce Conflict

```
Grammar

Rule 0     S' -> A
Rule 1     A -> B c d
Rule 2     A -> E c f
Rule 3     B -> x y
Rule 4     E -> x y
```

```
Terminals, with rules where they appear

c                    : 1 2
d                    : 1
error                :
f                    : 2
x                    : 3 4
y                    : 3 4

Nonterminals, with rules where they appear

A                    : 0
B                    : 1
E                    : 2
```

Grammar with a Reduce/Reduce Conflict

```
state 0

    (0) S' -> . A
    (1) A -> . B c d
    (2) A -> . E c f
    (3) B -> . x y
    (4) E -> . x y

    x               shift and go to state 4

    A                              shift and go to state 1
    B                              shift and go to state 2
    E                              shift and go to state 3
```

Grammar with a Reduce/Reduce Conflict

```
state 1

    (0) S' -> A .



state 2

    (1) A -> B . c d

    c               shift and go to state 5


state 3

    (2) A -> E . c f

    c               shift and go to state 6
```

Grammar with a Reduce/Reduce Conflict

```
state 4

    (3) B -> x . y
    (4) E -> x . y

    y               shift and go to state 7


state 5

    (1) A -> B c . d

    d               shift and go to state 8
```

Grammar with a Reduce/Reduce Conflict

```
state 6

    (2) A -> E c . f

    f               shift and go to state 9


state 7

    (3) B -> x y .
    (4) E -> x y .

  ! reduce/reduce conflict for c resolved using rule 3 (B -> x y .)
    c               reduce using rule 3 (B -> x y .)

  ! c               [ reduce using rule 4 (E -> x y .) ]
```

Grammar with a Reduce/Reduce Conflict

```
state 8

    (1) A -> B c d .

    $end            reduce using rule 1 (A -> B c d .)


state 9

    (2) A -> E c f .

    $end            reduce using rule 2 (A -> E c f .)
```

Grammar with a Reduce/Reduce Conflict

```
WARNING:
WARNING: Conflicts:
WARNING:
WARNING: reduce/reduce conflict in state 7 resolved using rule (B -> x y)
WARNING: rejected rule (E -> x y) in state 7
WARNING: Rule (E -> x y) is never reduced
```

Grammar with a Reduce/Reduce Conflict

- To resolve the reduce/reduce conflict, the parser generator *arbitrarily* picked one of the rules and reduced by it.
- While letting a shift/reduce conflict be resolved by picking shift over reduce *might* be the proper resolution, letting the parser generator *arbitrarily* pick a rule to reduce by is ***not*** the proper way to resolve this sort of conflict.
- Reduce/reduce conflicts ***must always*** be investigated and ***resolved*** by refactoring / restating that part of the grammar.

Grammar with a Reduce/Reduce Conflict

- Fixing the contrived example is kind of pointless.
  - It's *contrived*.
- We'll look at reduce/reduce conflicts some more as we develop the grammar for our language.

```
A → B c d
A → E c f
B → x y
E → x y
```

Table-Driven Shift-Reduce Parsing

- The generalized *Shift-Reduce* parser is similar to the one for *Predictive Parsing*.
- We again stay in a loop looking at tokens and taking actions until we succeed or fail.
- Here, there are *three* kinds of entries in the `ACTION` table: *shift*, *reduce*, and *accept*.
- There's also a `GOTO` table for resetting the state after a reduce.

```
push(0)
token ← readNextToken()

loop
  s ← top()

  if ACTION[ s, token ] = 'si'
    push( i )    // Shift onto stack
    token ← readNextToken()

  elif ACTION[ s, token ] = 'ri'
    // Reduce by rule i: X → A1 A2 … An
    pop() n times
    s ← top()
    push( GOTO[ s, X ] )

  elif ACTION[ s, token ] = 'a'
    break    // Successful parse!

  else
    // Unexpected state / token pair.
    error()
end
```

LR(k) Parsing

- The parser decides between *shifting* and *reducing* via a DFA.
  - This DFA is not *parsing* the input as a DFA is too weak to parse a CFG.
  - The DFA is applied to the contents of the *stack*.

- The possible DFA actions are:
  - *Shift*(n) : Advance one token; push n onto the stack.
  - *Reduce*(m) : Pop as many items as the number of symbols on the RHS of CFG production rule m. For the state now on the top of the stack, look up the LHS of rule m to get the next state n. Push n onto the stack.
  - *Accept* : Stop parsing and report success.
  - *Error* : Stop parsing and report failure.

A Simple CFG …

1 $S \rightarrow S \,;\, S$

2 $S \rightarrow \mathrm{id} := E$

3 $S \rightarrow \mathrm{print}\ (L)$

4 $E \rightarrow \mathrm{id}$

5 $E \rightarrow \mathrm{num}$

6 $E \rightarrow E + E$

7 $E \rightarrow (S , E)$

8 $L \rightarrow E$

9 $L \rightarrow L , E$

A Simple CFG's LR Parsing Table

	id	num	print	;	,	+	:=	(	)	$	*S*	*E*	*L*
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9	s20	s10						s8				g15	g14
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
12				s3	s18								
13				r3	r3					r3			
14					s19				s13				
15					r8				r8				
16	s20	s10						s8				g17	
17				r6	r6	s16			r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

s*n* Shift into state *n*.
g*n* Go to state *n*.
r*m* Reduce by rule *m*.
a Accept

Blanks indicate *error*.

LR Parsing Example

```
a := 7;
b := c + (d := 5 + 6, d)
```

Stack	*Input*	*Action*
$_1$	a := 7 ; b := c + (d := 5 + 6 , d) $	*shift*
$_1\ \text{id}_4$	:= 7 ; b := c + (d := 5 + 6 , d) $	*shift*
$_1\ \text{id}_4 :=_6$	7 ; b := c + (d := 5 + 6 , d) $	*shift*
$_1\ \text{id}_4 :=_6 \text{num}_{10}$	; b := c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow \text{num}$
$_1\ \text{id}_4 :=_6 E_{11}$	; b := c + (d := 5 + 6 , d) $	*reduce* $S \rightarrow \text{id} := E$
$_1\ S_2$	; b := c + (d := 5 + 6 , d) $	*shift*
$_1\ S_2 ;_3$	b := c + (d := 5 + 6 , d) $	*shift*
$_1\ S_2 ;_3 \text{id}_4$	:= c + (d := 5 + 6 , d) $	*shift*
$_1\ S_2 ;_3 \text{id}_4 :=_6$	c + (d := 5 + 6 , d) $	*shift*
$_1\ S_2 ;_3 \text{id}_4 :=_6 \text{id}_{20}$	+ (d := 5 + 6 , d) $	*reduce* $E \rightarrow \text{id}$
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11}$	+ (d := 5 + 6 , d) $	*shift*
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16}$	(d := 5 + 6 , d) $	*shift*
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} (_8$	d := 5 + 6 , d) $	*shift*
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} (_8 \text{id}_4$	:= 5 + 6 , d) $	*shift*
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} (_8 \text{id}_4 :=_6$	5 + 6 , d) $	*shift*
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} (_8 \text{id}_4 :=_6 \text{num}_{10}$	+ 6 , d) $	*reduce* $E \rightarrow \text{num}$
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} (_8 \text{id}_4 :=_6 E_{11}$	+ 6 , d) $	*shift*
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} (_8 \text{id}_4 :=_6 E_{11} +_{16}$	6 , d) $	*shift*
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} (_8 \text{id}_4 :=_6 E_{11} +_{16} \text{num}_{10}$	, d) $	*reduce* $E \rightarrow \text{num}$
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} (_8 \text{id}_4 :=_6 E_{11} +_{16} E_{17}$	, d) $	*reduce* $E \rightarrow E + E$
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} (_8 \text{id}_4 :=_6 E_{11}$	, d) $	*reduce* $S \rightarrow \text{id} := E$
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} (_8 S_{12}$	, d) $	*shift*
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} (_8 S_{12} ,_{18}$	d) $	*shift*
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} (_8 S_{12} ,_{18} \text{id}_{20}$	) $	*reduce* $E \rightarrow \text{id}$
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} (_8 S_{12} ,_{18} E_{21}$	) $	*shift*
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} (_8 S_{12} ,_{18} E_{21})_{22}$	$	*reduce* $E \rightarrow (S, E)$
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11} +_{16} E_{17}$	$	*reduce* $E \rightarrow E + E$
$_1\ S_2 ;_3 \text{id}_4 :=_6 E_{11}$	$	*reduce* $S \rightarrow \text{id} := E$
$_1\ S_2 ;_3 S_5$	$	*reduce* $S \rightarrow S; S$
$_1\ S_2$	$	*accept*

LR(*k*) Parsing

- OK, enough theory.

- No, I am not going to drag you through how to process the CFG to get the DFA for parsing.
  - That's the job of a *compiler-compiler* tool such as `ply`, `yacc`, `bison`, etc.
  - FYI, at last check, Wikipedia's list of *Deterministic Context-Free Language Parser Generators* list had ***98*** entries.

2022 Sep 09, https://en.wikipedia.org/wiki/Comparison_of_parser_generators

The `bison` Parser Generator

- The point is that someone else has already done all of that work for you.

- Just use the `bison` parser generator so we can concentrate on the *language* instead of the tool that got us there.

bison calc Example

- A barebones "desk calculator".
- Add +, Subtract -, Multiply *, and Divide / operators.
  - And Unary + and -, Exponent ^
- Variables to hold results.
  - And some built-in constants (e.g., `pi`).
  - And some flags to control the display (`showParseTree`, `showRomanInts`).

```
> 1 + 1
"1 + 1" ==>
 (LITERAL INTEGER 2)
> r = 1 + 2
"r = 1 + 2" ==>
 (LITERAL INTEGER 3)
> showParseTree = 1
"showParseTree = 1" ==>
 (LITERAL INTEGER 1)
> pi * r^2
 (BOP MULTIPLY
  (ID "pi")
  (BOP EXPONENT
   (ID "r")
   (LITERAL INTEGER 2)
  )
 )
"pi * r^2" ==>
 (LITERAL REAL 2.8274333882308138e+01)
> showParseTree = 0
"showParseTree = 0" ==>
 (LITERAL INTEGER 0)
> showRomanInts = 1
"showRomanInts = 1" ==>
 (LITERAL INTEGER 1)
> 0xabc + 0rXVIII
"0xabc + 0rXVIII" ==>
 (LITERAL INTEGER 0rMMDCCLXVI)
>
```

Parse Tree

- The output of the *Syntactic Analyzer* phase is a *Parse Tree*.
  - This tree represents the *structure* of the particular input token stream as determined by the language's grammar rules.
  - *Unique* for a given stream of tokens.
    - If not, the grammar is ambiguous and needs to be fixed.
- It's normally called a *Raw* (or *Concrete*) tree in that its leaves are the tokens and its internal nodes correspond to the applied production rules.

Parse Tree

- Correspondence is not always *exact*.
  - Not all tokens may be represented.
  - Alterations / Simplifications may have been made.

- There can be a *blurring* of the line between the *Parse Tree* and the *Abstract Syntax Tree* (*AST*).
  - ASTs will be considered in *Semantic Analysis*.

Syntactic Error Recovery

- What to do when a syntax error is detected?
  - Complain and die? — Most obvious answer but very unfriendly. Probably would like to see *other* errors.
  - However, have to avoid an avalanche of cascading errors.
- Exactly what's possible depends on how the parser was generated.
  - Different compiler compilers have (very) different error detection, control, and recovery mechanisms.
  - Hand-written parsers have the best flexibility!

UNIVERSITY OF TEXAS ARLINGTON

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Language / Grammar Hierarchy Restatement

All Languages
Non-computable even with Turing Machine.

Recognizable Languages
Turing Machine will halt on YES, *may loop infinitely otherwise.*

Decidable Languages
Turing Machine will always halt, answering YES *or* NO.

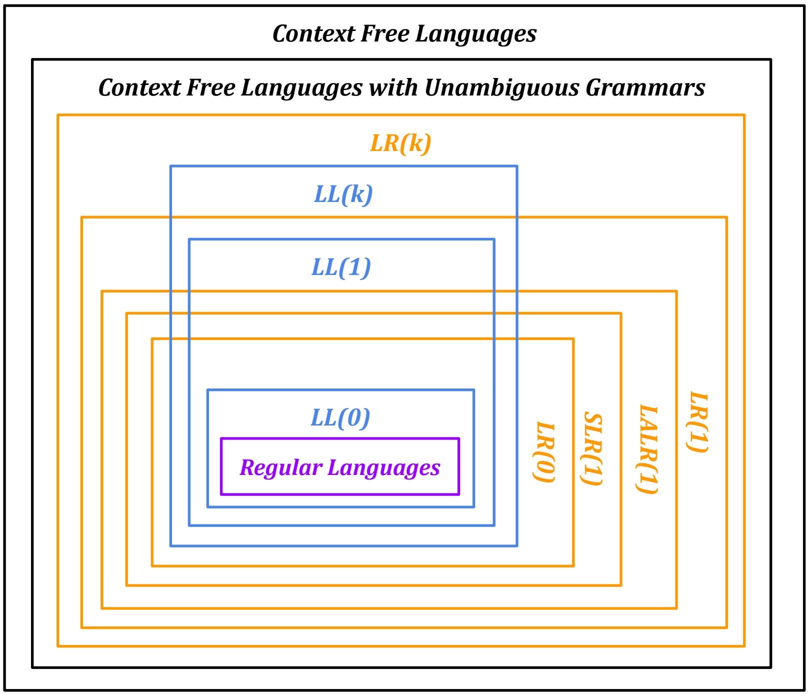
Context-Sensitive Languages
Linear-Bounded Automaton

Context-Free Languages
[See detail on next slide.]

"Recognizable"
AKA *"Turing Recognizable"*
AKA *"Recursively Enumerable"*
AKA *"Semi-Decidable"*

"Decidable"
AKA *"Turing Decidable"*
AKA *"Recursive"*

"Linear Bounded Automaton"
A Turing Machine whose tape is finite in length, limited to some *constant* times the length of the input.



Arbitrary *Context-Free Languages* require *ambiguous* grammars and a *Nondeterministic Pushdown Automaton* for parsing. However even the *Unambiguous CFL* subset requires an NDPDA.

It's not until one gets to the `LR(k)` subset that a *Deterministic PDA* suffices.

A DPDA is required until one reaches the *Regular Languages*. At that point, an FSM suffices.

Interesting observation: The computational power of a *Nondeterministic* Turing Machine is the same as that of a *Deterministic* TM. Similarly for FSM: *Nondeterministic* FSM and *Deterministic* FSM have the same computational power.

However, this is *not* true for Pushdown Automata. *Nondeterministic* PDA have strictly more computational power than *Deterministic* PDA. (Why? Think about it. :)

Really interesting observation: Even though a Turing Machine is strictly more powerful than even an NDPDA, if we give even a *Deterministic* PDA *two* stacks instead of just *one* it's suddenly *equivalent* to a Turing Machine in computational power. (Why? Think about it. :)

Why all those Subsets in the CFL Set?

- We've already defined `LL(k)` and `LR(k)`.
  - `LL(k)` : ***L***eft-to-Right scan of the input, ***L***eft-most derivation, ***k*** units of lookahead.
  - `LR(k)` : ***L***eft-to-Right scan of the input, ***R***ight-most derivation, ***k*** units of lookahead.
- These subsets (and their specific sub-subsets `LL(0)`, `LL(1)`, `LR(0)`, `LR(1)`) are useful in that we have algorithms that can produce *parsers* from *grammars* that describe these kinds of languages.

What about SLR and LALR?

Donald Knuth

- While we have methods for making parsers from these kinds of grammars …
  - Thank you, Donald Knuth! "*On the Translation of Languages from Left to Right*", *Information and Control*, v8 n6, pp. 607–639.
- … the size of the tables generated can be *immense*.
- `SLR` (*Simple LR*) and `LALR` (*Look-Ahead LR*) reduce the size of the tables by *combining* certain transitions.

https://commons.wikimedia.org/wiki/File:Donald_Knuth_1965.png

What about SLR and LALR?

- If transitions can be combined, why not always do that?
- ***Conflicts!***
- `Shift` / `Reduce` `:(` and `Reduce` / `Reduce` `=:o` conflicts that didn't previously exist might arise.
- It's a tradeoff.
  - If your language is one for which an unconflicted `SLR(1)` or `LALR(1)` grammar can be constructed, go for it!
  - If not, you'll have to restate your grammar or use a more powerful parsing technique.
    - *Please!* Don't just ignore the conflicts. `:)`

What about the Parser Generators?

- `bison` by default generates `LALR(1)` parsers.
  - It can also generate `IELR(1)`, `LR(1)`, and `GLR` parsers.
  - Target languages are C, C++, and Java.
- `yacc` generates `LALR(1)` parsers only.
  - Target language is C.
- `ply` generates `LALR(1)` parsers only.
  - Target language is Python.

https://en.wikipedia.org/wiki/Comparison_of_parser_generators

UNIVERSITY OF TEXAS ARLINGTON

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING