Programming Language Syntax

CSE 307 – Principles of Programming Languages

Stony Brook University

http://www.cs.stonybrook.edu/~cse307

Programming Languages Syntax

- **Computer languages must be precise:**
  - Both their form (syntax) and meaning (semantics) must be specified without ambiguity, so that both programmers and computers can tell what a program is supposed to do.
  - Example: the syntax of Arabic numerals:
    - A ***digit*** "is": 0 |(or) 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    - A ***non_zero_digit*** "is" 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    - A ***natural_number*** (>0)"is"a ***non_zero_digit*** followed by other ***digit***s (a number that doesn't start with 0) = the regular expression "***non_zero_digit digit****"
- **Specifying the syntax for programming languages has 2 parts: Regular Expressions (RE) and Context-Free Grammars**

Regular Expressions

- A ***regular expression*** is one of the following:
  - a character
  - the empty string, denoted by ε
  - two regular expressions concatenated
    - E.g., **`name -> letter letter`**
  - two regular expressions separated by **|** (i.e., or),
    - E.g., **`name -> letter ( letter | digit )`**
  - a regular expression followed by the Kleene star <u>(concatenation of zero or more strings)</u>
    - E.g., **`name -> letter ( letter | digit )*`**

Regular Expressions

- RE example: the syntax of numeric constants can be defined with regular expressions:

A *digit* "is"	**0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**
A ***number*** "is"	***integer* | *real***
An ***integer*** "is"	***digit digit****
A ***real*** "is"	***integer exponent***
	| *decimal* (*exponent* | ε)
A ***decimal*** "is"	***digit* (.*digit* | *digit*.) *digit****
An ***exponent*** "is"	**(e | E) (+ | - | ε) *integer***

Regular Expressions

- **Regular expressions work well for defining tokens**
  - **They are unable to specify nested constructs**
    - **For example, a context free grammar in BNF form to define arithmetical expressions:**

expr → id | number | - expr | (expr) | expr op expr

op → + | - | * | /

- **Same number of open and closed parenthesis cannot be represented**
  - Proof that 0^n1^n cannot be represented using RE is in the CD extension of the textbook using contradiction and the pigeonhole principle

Chomsky Hierarchy

- ***Context Free Languages*** are **strictly more powerful** than Regular Expressions, BUT, Regular Expressions are **way faster to recognize**, so
  - Regular Expressions are used to create tokens, atoms of the syntax tree.
- ***Chomsky Hierarchy:***
  - Type-3: Regular Language - Finite Automata/Regex (Lexer, Scanner, Tokenizer)
  - Type-2: Context-Free Language - Pushdown Automata (Parser)
  - Type-1: Context-Sensitive Language - Turing Machine w/ Tape * Input
  - Type-0: Unrestricted Language - Turing Machine
    - Types 0 and 1 usually too slow for practical use
    - Type 2 maybe, maybe not for practical use , $O(N^3)$ in worst case
    - Type 3 are fast (linear time to recognize tokens), but not expressive enough

Context-Free Grammars (CFG)

- ***Backus–Naur Form (BNF) notation*** for CFG:

expr → id | number | - expr | (expr) | expr op expr

op → + | - | * | /

  - Each of the rules in a CFG is known as a ***production***.
  - The symbols on the left-hand sides of the productions are ***nonterminals*** (or ***variables***)

- A CFG consists of:
  - a set of terminals T (that cannot appear on the left-hand side of any production)
  - a set of non-terminals N
  - a start symbol S (a non-terminal), and
  - a set of productions

Context-Free Grammars (CFG)

- John Backus was the inventor of Fortran (won the ACM Turing Award in 1977).
- John Backus and Peter Naur used the BNF form for Algol.
  - Peter Naur also won the ACM Turing Award in 2005 for *Report on the Algorithmic Language ALGOL 60*.
- BNF was named by Donald Knuth

Context-Free Grammars (CFG)

- The Kleene star `*` and meta-level parentheses of regular expressions do not change the expressive power of the notation

```
id_list → id ( , id )*
```

is shorthand for

```
id_list → id id_list_tail
id_list_tail → , id id_list_tail
id_list_tail → ε
```

or the left-recursive version

```
id_list → id
id_list → id_list , id
```

Context-Free Grammars (CFG)

- From RE to BNF notation:
  - Consider the RE: **`a*( b a* b )*`**
    - Start with **`a*`**:

```
As -> a As
    | ε
```

Same with **`( b a* b )*`**. It is:

```
S -> b As b S
    | ε
```

Now you concatenate them into a single non-terminal:

```
G -> As S
```

Context-Free Grammars (CFG)

- ***Derivations and Parse Trees***: A context-free grammar shows us how to ***generate*** a syntactically valid string of terminals
  1. Begin with the start symbol.
  2. Choose a production with the start symbol on the left-hand side; replace the start symbol with the right-hand side of that production.
  3. Now choose a nonterminal **A** in the resulting string, choose a production **P** with **A** on its left-hand side, and replace **A** with the right-hand side of **P**
  - **Repeat this process until no non-terminals remain**
    - The replacement strategy named ***right-most derivation*** chooses at each step to replace the right-most nonterminal with the right-hand side of some production.
      - There are many other possible derivations, including ***left-most*** and options in between.

Context-Free Grammars (CFG)

- Example: we can use our grammar for expressions to generate the string "*slope* * *x* + *intercept*":

expr ⇒ expr op expr
⇒ expr op id
⇒ expr + id
⇒ expr op expr + id
⇒ expr op id + id
⇒ expr * id + id
⇒ id * id + id
⇒ id(*slope*)* id(*x*)+ id(*intercept*)

Grammar:

expr → id | number
| - expr | (expr)
| expr op expr

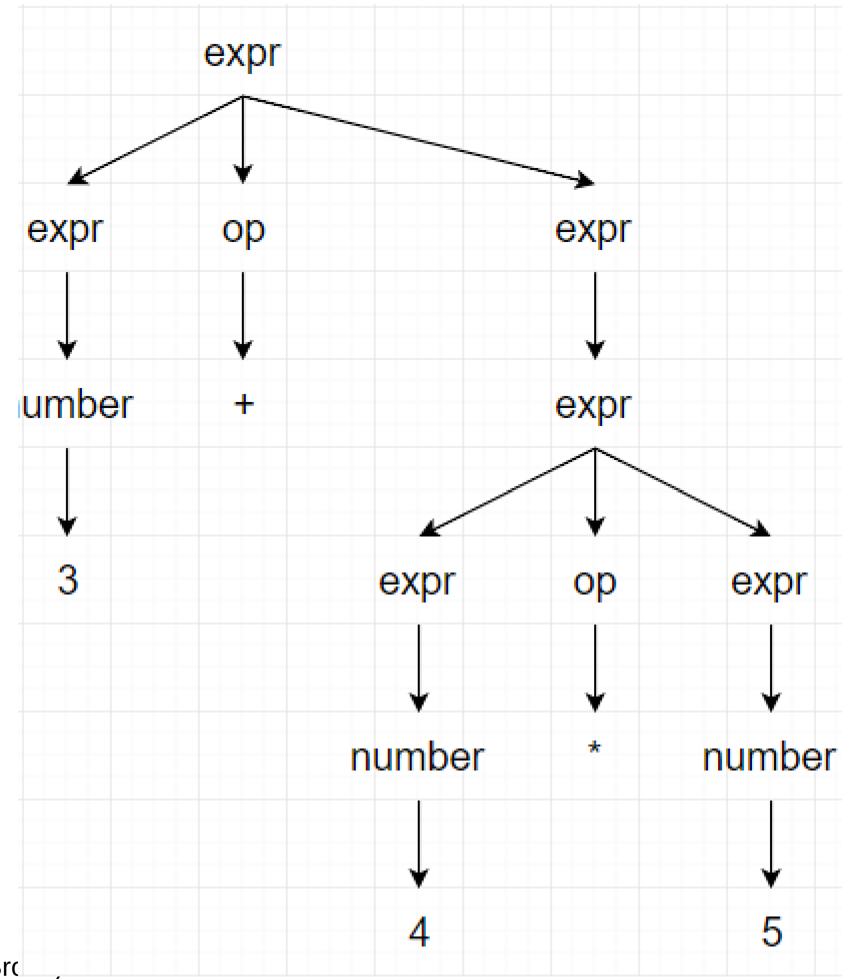
op → + | - | * | /

Notes: The ⇒ metasymbol is often pronounced "*derives*"

- A series of replacement operations that shows how to derive a string of terminals from the start symbol is called a ***derivation***
- Each string of symbols along the way is called a ***sentential form***
- The final sentential form, consisting of only terminals, is called the ***yield*** of the derivation

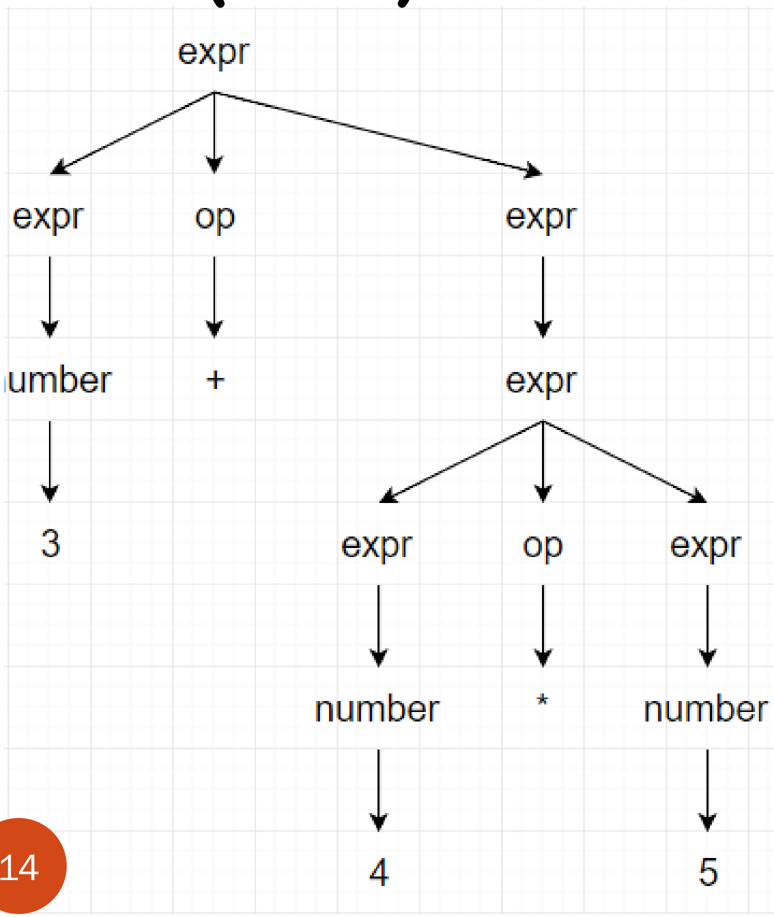
Derivations and Parse Trees

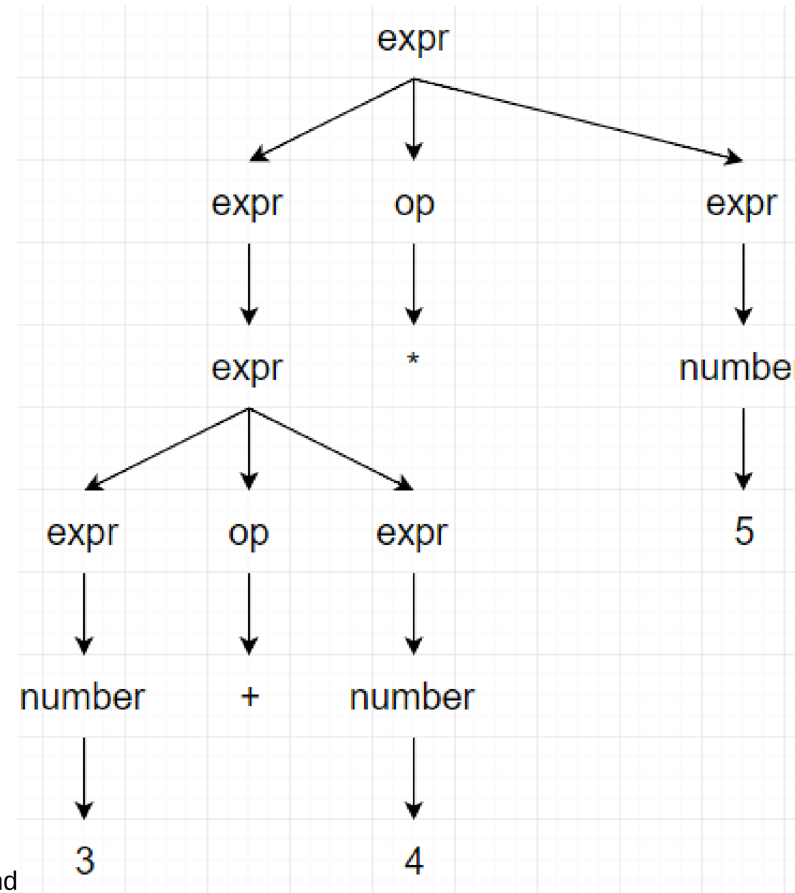
- We can represent a derivation graphically as a ***parse tree***
  - The root of the parse tree is the start symbol of the grammar
  - The leaves are its yield
  - Each node with its children represent a production
- E.g., The parse tree for the expression grammar for **`3 + 4 * 5`** is:



Derivations and Parse Trees

- The previous grammar was ***ambiguous*** (it can generate multiple parse trees for **`3+4*5`**): one corresponds to **`3+(4*5)`** and one corresponds to **`(3+4)*5`**

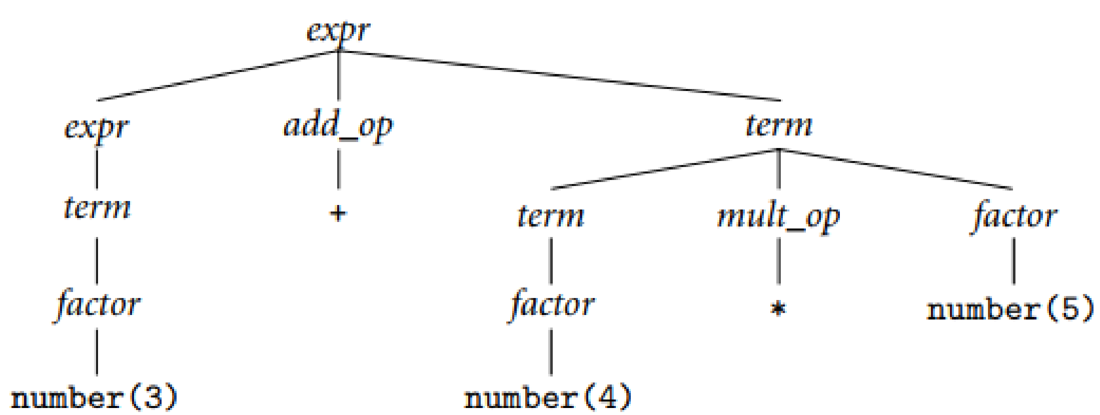




Context free grammars

- A better version of our expression grammar should include precedence and associativity:

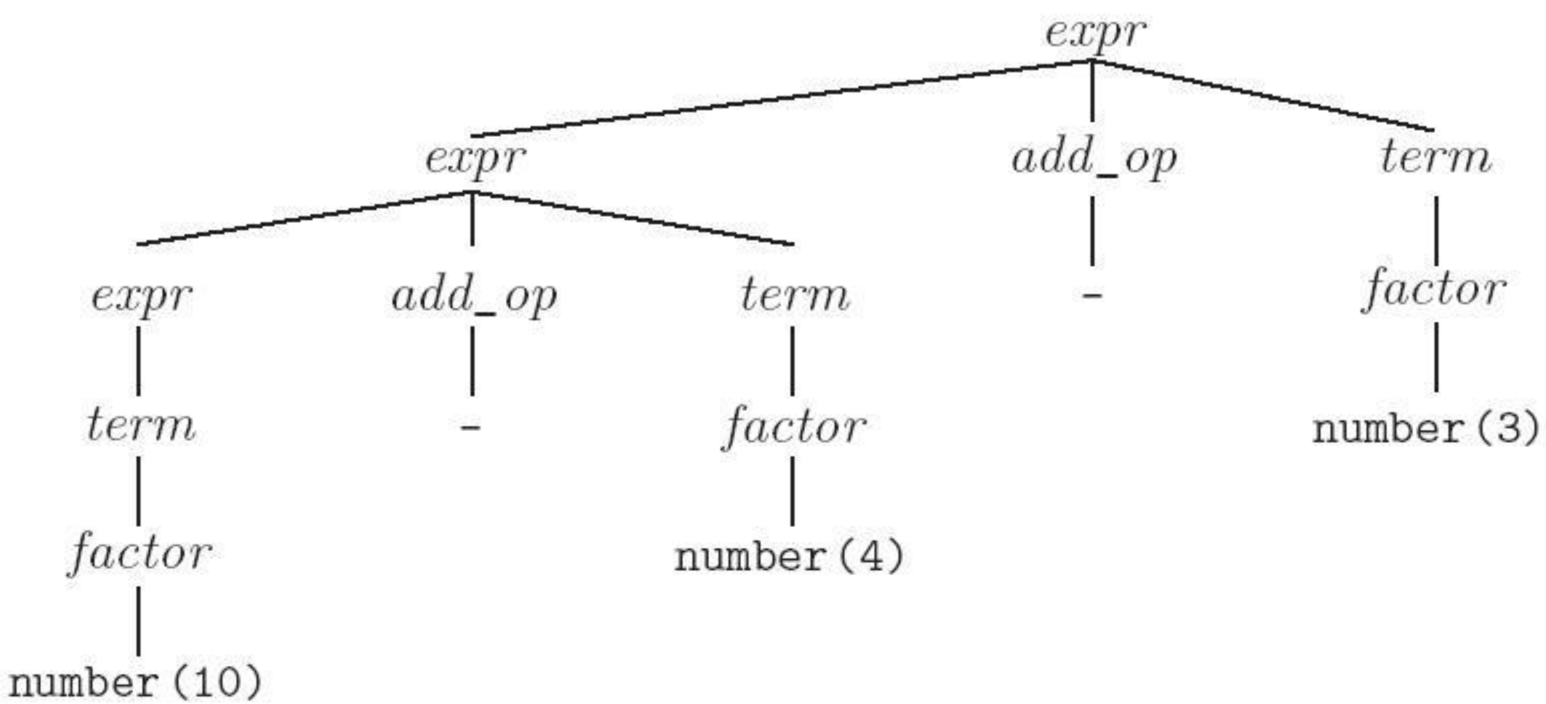
$$
\begin{aligned}
&\textit{expr} \longrightarrow \textit{term} \mid \textit{expr add\_op term} \\
&\textit{term} \longrightarrow \textit{factor} \mid \textit{term mult\_op factor} \\
&\textit{factor} \longrightarrow \textbf{id} \mid \textbf{number} \mid \textbf{-}\,\textit{factor} \mid \textbf{(}\ \textit{expr}\ \textbf{)} \\
&\textit{add\_op} \longrightarrow \textbf{+} \mid \textbf{-} \\
&\textit{mult\_op} \longrightarrow \textbf{*} \mid \textbf{/}
\end{aligned}
$$



Parse tree for 3 + 4 * 5, with precedence

Context free grammars

- Parse tree for expression grammar for 10 - 4 - 3



  - has ***left associativity***

Scanning

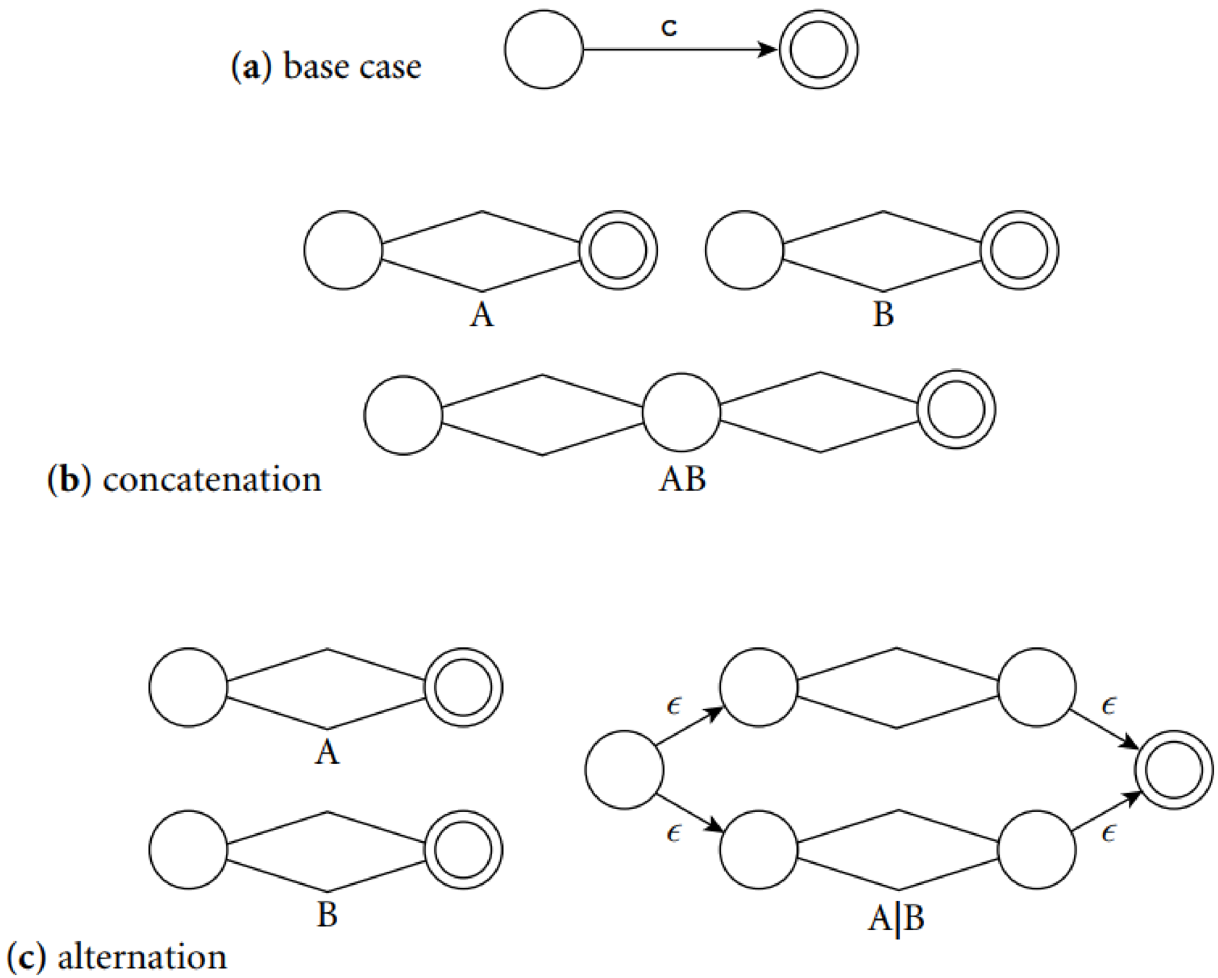
- The scanner and parser for a programming language are responsible for discovering the syntactic structure of a program (i.e., the ***syntax analysis***)
- The ***scanner/lexer*** is responsible for
  - tokenizing source
  - removing comments
  - (often) dealing with pragmas (i.e., significant comments)
  - saving text of identifiers, numbers, strings
  - saving source locations (file, line, column) for error messages

Scanning

- The Scanner turns a program into a string of tokens
- It matches **regular expressions** to a program and creates a list of tokens
  - However, there are two syntaxes for regular expressions: EBNF and Perl-style Regex
- Scanners tend to be built three ways:
  - Writing / Generating a finite automaton from REs
  - Scanner code (usually realized as nested if/case statements)
  - Table-driven DFA
- Writing / Generating a finite automaton generally yields the fastest, most compact code by doing lots of special-purpose things, although good automatically-generated scanners come very close

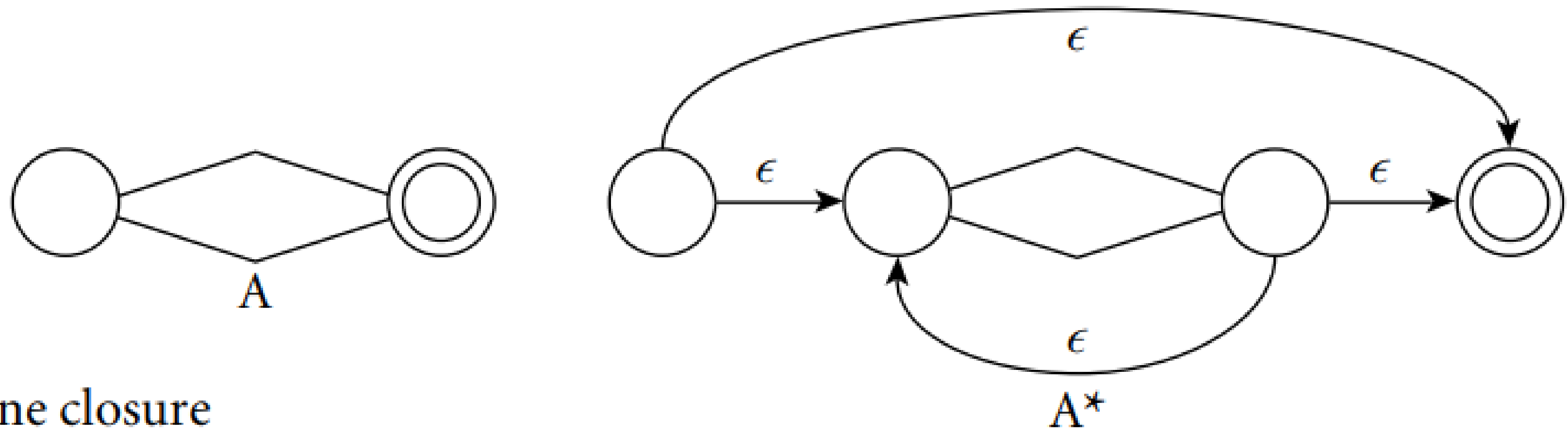
Scanning

- Construction of an NFA equivalent to a given regular expression: cases



Scanning

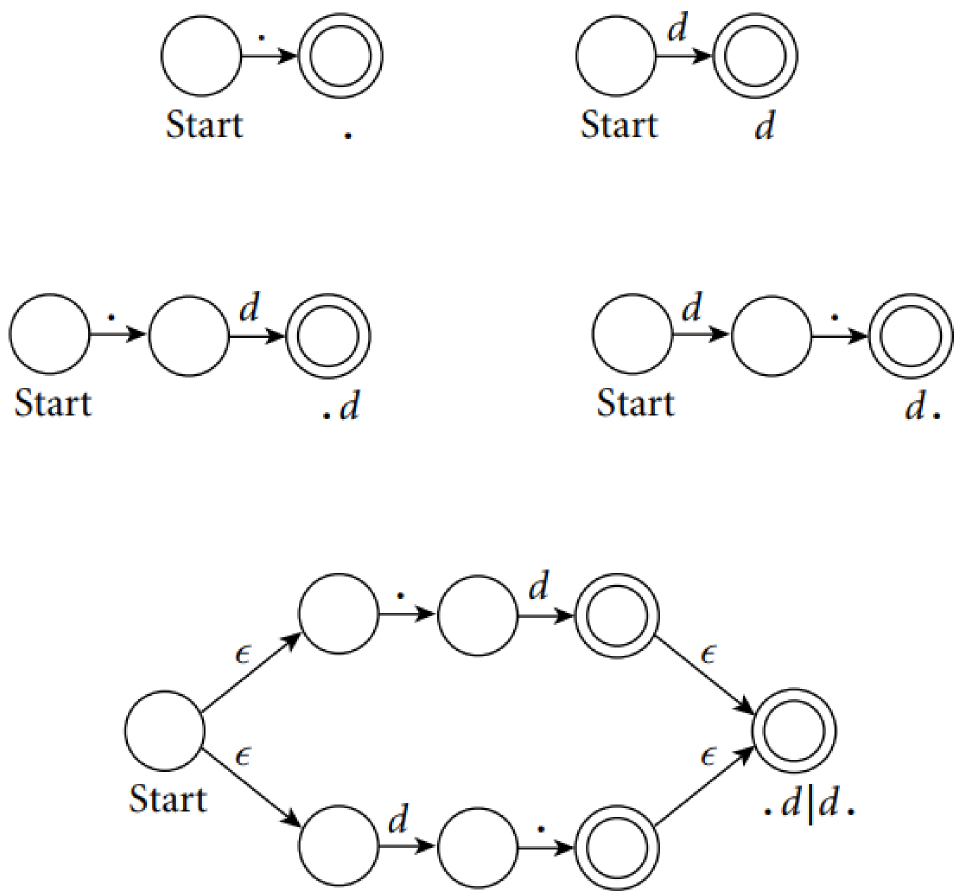
- Construction of an NFA equivalent to a given regular expression: cases



(d) Kleene closure

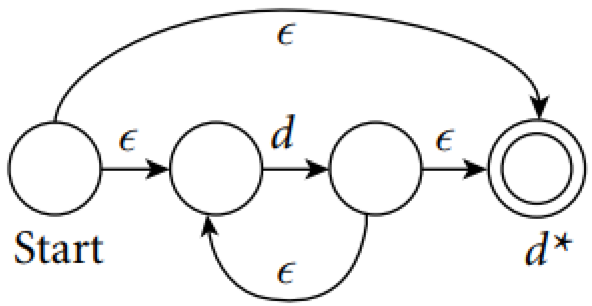
Scanning

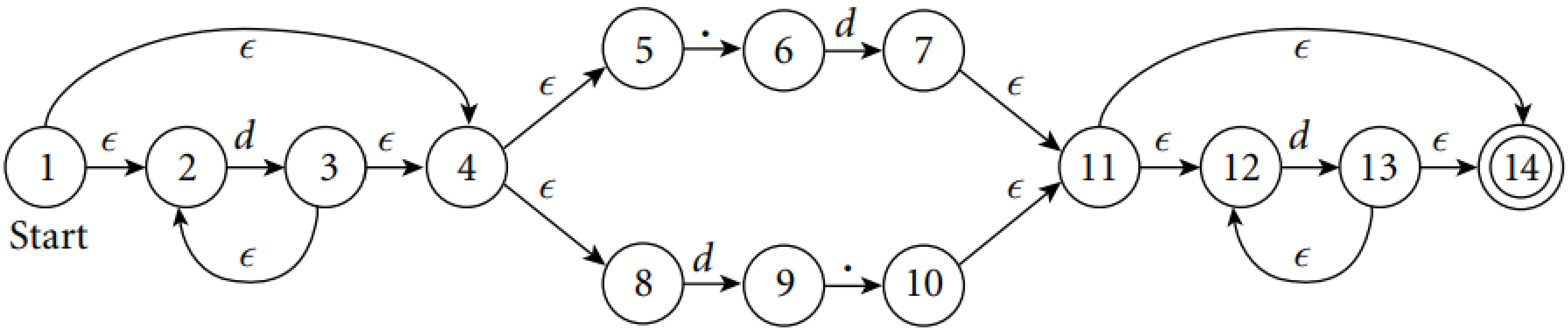
- Construction of an NFA equivalent to the regular expression **`d* ( .d | d. ) d*`**



Scanning

- Construction of an NFA equivalent to the regular expression **`d* ( .d | d. ) d*`**





Scanning

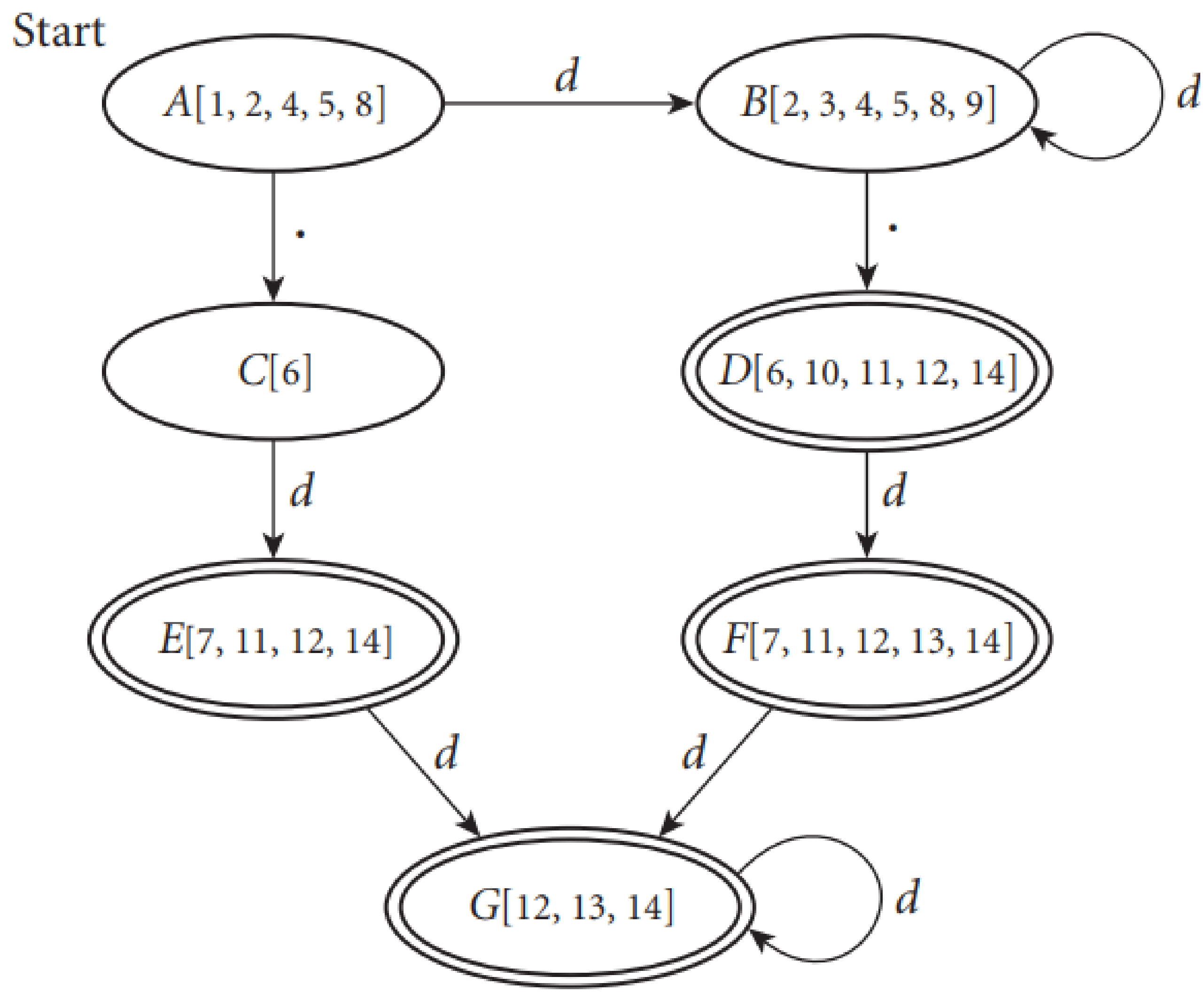
- From an NFA to a DFA:
  - Reason: With no way to "guess" the right transition to take from any given state, any practical implementation of an NFA would need to explore **all possible transitions concurrently or via backtracking**
  - We can instead build a DFA from that NFA:
    - The state of the DFA after reading any input will be the set of states that the NFA might have reached on the same input
      - Our example: Initially, before it consumes any input, the NFA may be in **State 1**, or it may make <u>epsilon transitions</u> to **States 2, 4, 5, or 8**
        - We thus create an initial **State A** for our DFA to represent this set: **1,2,4,5,8**

Scanning

- On an input of **d**, our NFA may move from **State 2** to **State 3**, or from **State 8** to **State 9**.
  - It has no other transitions on this input from any of the states in A.
  - From **State 3**, however, the NFA may make epsilon transitions to any of **States 2, 4, 5, or 8**.
  - We therefore create DFA **State B: 2, 3, 4, 5, 8, 9**
- On a **.**,our NFA may move from **State 5** to **State 6**
  - There are no other transitions on this input from any of the states in A, and there are no epsilon transitions out of **State 6**.
  - We therefore create the singleton DFA **State C: 6**
- **We continue the process until we find all the states and transitions in the DFA (it is a finite process – Why?)**

Scanning

- The DFA equivalent to our previous NFA:



Scanning

- Suppose we are building an ***ad-hoc (hand-written) scanner*** for a Calculator:

```
assign → :=
plus → +
minus → -
times → *
div → /
lparen → (
rparen → )
id → letter ( letter | digit )*
number → digit digit *
    | digit * ( . digit | digit . ) digit *
comment → /* ( non-* | * non-/ )* */
         | // ( non-newline )* newline
```

Scanning

- We read the characters one at a time with look-ahead

  skip any initial white space (spaces, tabs, and newlines)

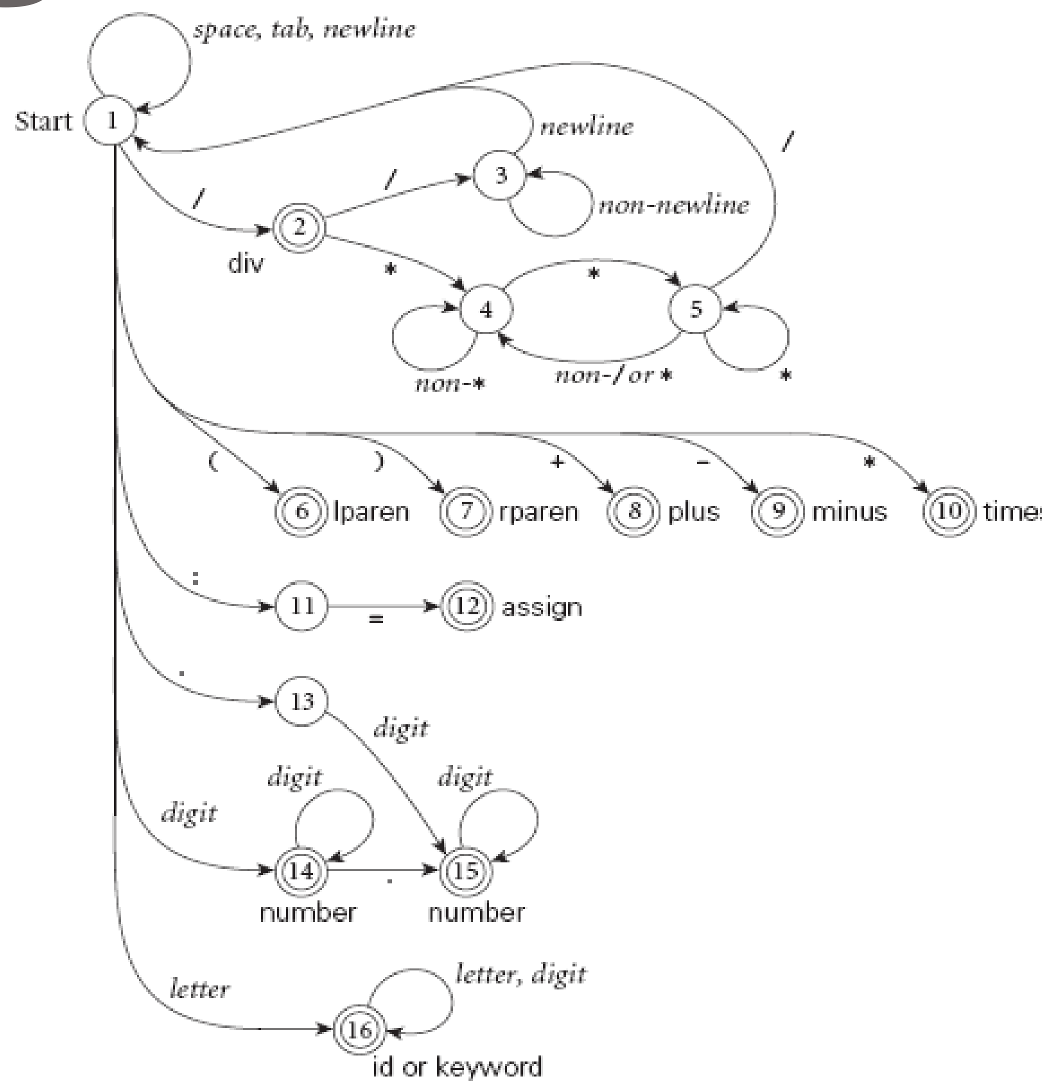
```
if cur_char ∈ {‘(’, ‘)’, ‘+’, ‘-’, ‘*’}
      return the corresponding single-character token
if cur_char = ‘:’
      read the next character
      if it is ‘=’ then return assign else announce an error
if cur_char = ‘/’
      peek at the next character
      if it is ‘*’ or ‘/’
        read additional characters until “*/” or newline
             is seen, respectively
      jump back to top of code
else return div
```

Scanning

```
if cur_char = .
     read the next character
     if it is a digit
            read any additional digits
            return number
     else announce an error
if cur_char is a digit
     read any additional digits and at most one decimal point
     return number
if cur_char is a letter
     read any additional letters and digits
     check to see whether the resulting string is read or
write
     if so then return the corresponding token
     else return id
else announce an error
```

Scanning

- Pictorial representation of a scanner for calculator tokens, in the form of a finite automaton



Scanning

- We run the machine over and over to get one token after another
  - Nearly universal rule:
    - always take the longest possible token from the input thus `foobar` is `foobar` and never `f` or `foo` or `foob`
    - more to the point, `3.14159` is a real constant and never `3`, `.`, and `14159`

Scanning

- The rule about longest-possible tokens means you return only when the next character can't be used to continue the current token
  - the next character will generally need to be saved for the next token
- In some cases, you may need to peek at **more than one character of look-ahead** in order to know whether to proceed
  - In Pascal, for example, when you have a **`3`** and you a see a dot
    - do you proceed (in hopes of getting **`3.14`**)? or
    - do you stop (in fear of getting **`3..5`**)? (declaration of arrays in Pascal, e.g., "**`array [1..6] of Integer`**")

Scanning

- Writing a pure DFA as a set of nested case statements is a surprisingly useful programming technique
  - use **`perl`**, **`awk`**, **`sed`**
- Table-driven DFA is what **`lex`** and **`scangen`** produce
  - **`lex`** (**`flex`**) in the form of C code
  - **`scangen`** in the form of numeric tables and a separate driver

Perl-style Regexp

- Learning by examples:

  abcd - concatenation

  a(b|c)d - grouping

  a(b|c)*d - can apply a number of repeats to char or group

  ? = 0-1

  ***** = 0-inf

  + = 1-inf

  [bc] - character class

  [a-zA-Z0-9_] - ranges

  . - matches any character.

  \a - alpha

  \d - numeric

  \w - word (alpha, num, _)

  \s - whitespace

Perl-style Regexp

- Learning by examples:

  How do we write a regexp that matches floats?

  `digit*(.digit|digit.)digit*`

  `\d*(\.\d|\d \.)\d*`

Parsing

- The ***parser*** calls the scanner to get the tokens, assembles the tokens together into a ***syntax tree***, and passes the tree (perhaps one subroutine at a time) to the later phases of the compiler (this process is called ***syntax-directed translation***).
- Most use a context-free grammar (CFG)

Parsing

- It turns out that for any CFG we can create a parser that runs in `O(n`^{3}`)` time (e.g., Earley's algorithm and the Cocke-Younger-Kasami (CYK) algorithm)
  - `O(n`^{3}`)` time is clearly unacceptable for a parser in a compiler - too slow even for a program of 100 tokens (~1,000,000 cycles)

Parsing

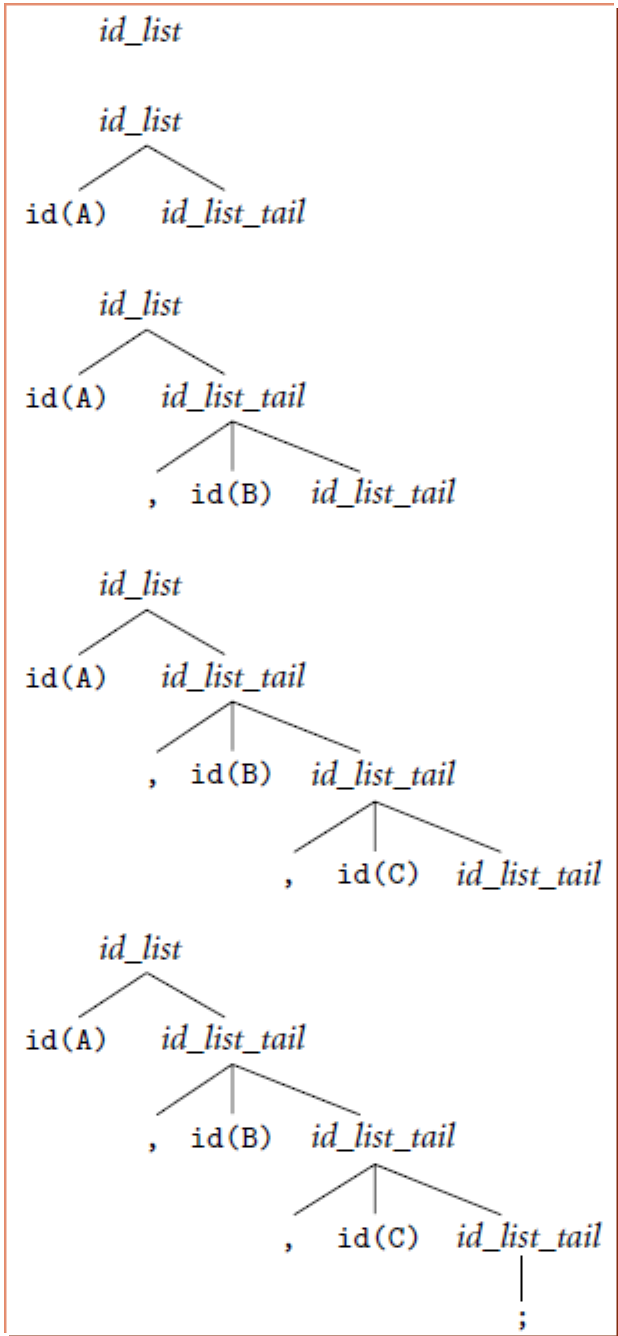
- Fortunately, there are large classes of grammars for which we can build parsers that run in linear time
  - The two most important classes are called LL and LR
    - LL stands for ***Left-to-right, Leftmost derivation***
      - Leftmost derivation - work on the left side of the parse tree
    - LR stands for ***Left-to-right, Rightmost derivation***
      - Rightmost derivation - work on the right side of the tree
  - LL parsers are also called '*top-down*', or '*predictive*' parsers
  - LR parsers are also called '*bottom-up*', or '*shift-reduce*' parsers

Top-down parsing (LL)

Consider a grammar for a comma separated list of identifiers, terminated by a semicolon:

id_list* → id *id_list_tail
id_list_tail* → , id *id_list_tail
***id_list_tail* → ;**

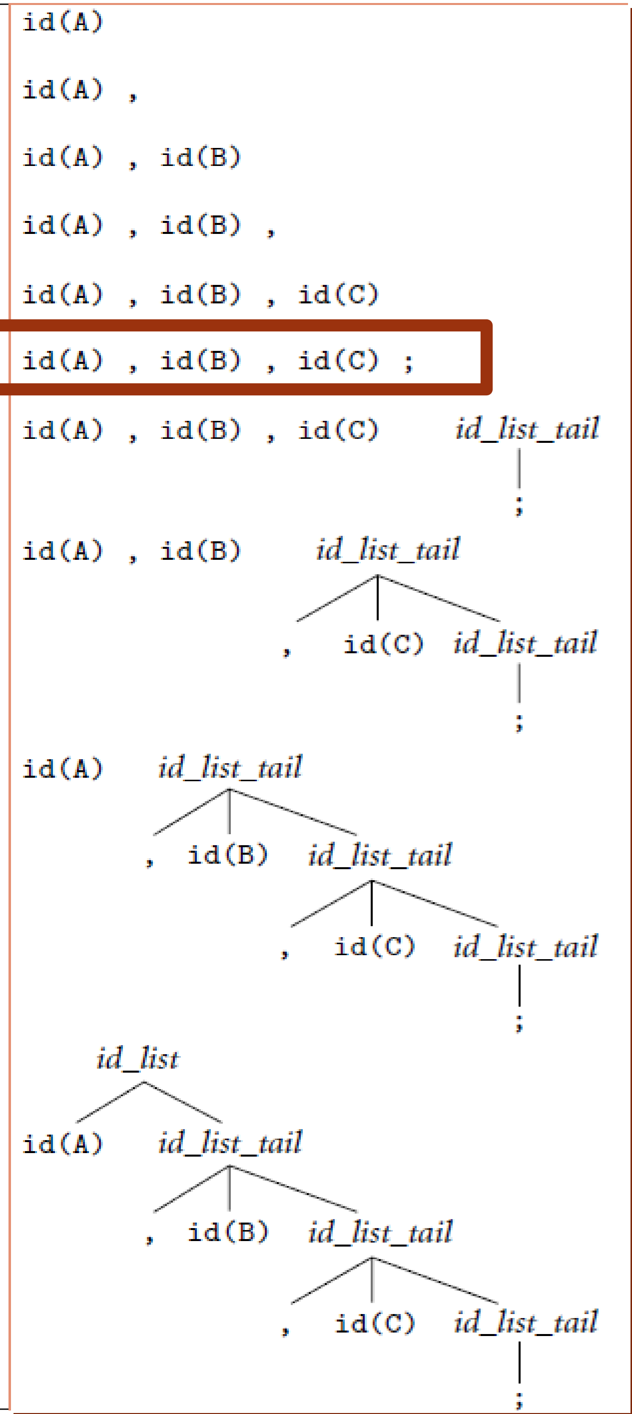
- The top-down construction of a parse tree for the string: "A, B, C;" starts from the root and applies rules and tried to identify nodes.



Bottom-up parsing (LR)

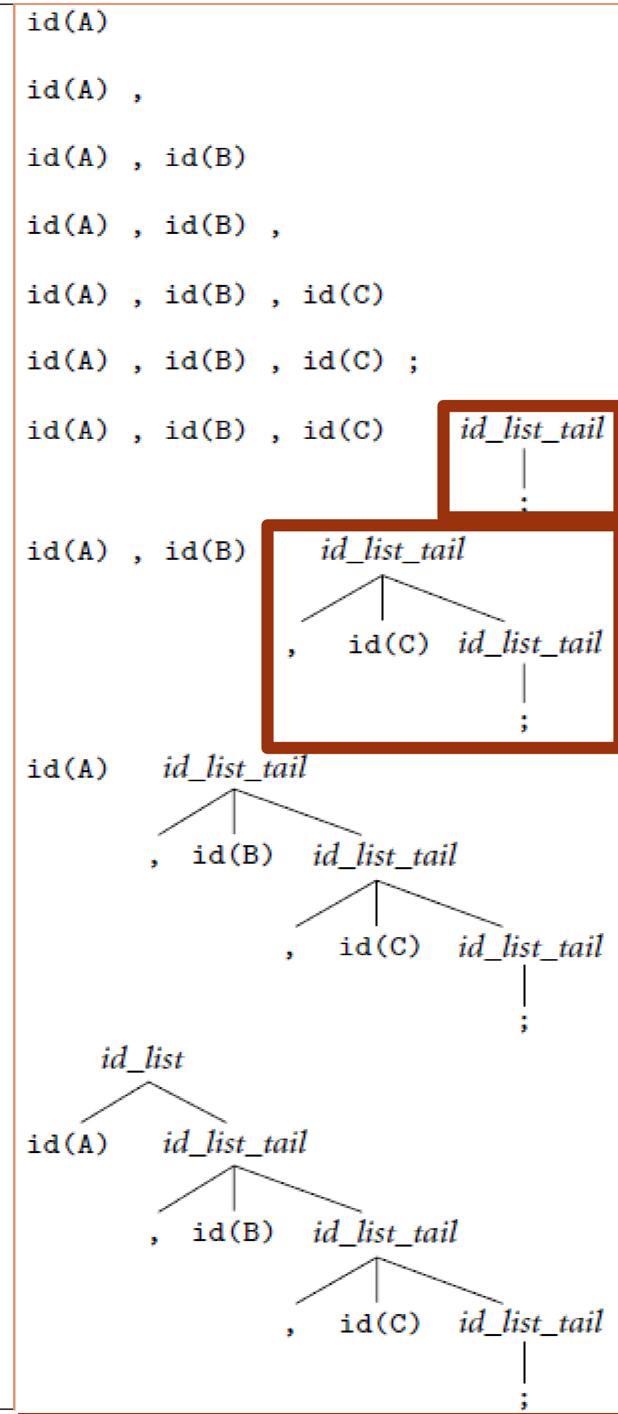
id_list* → id *id_list_tail
id_list_tail* → , id *id_list_tail
***id_list_tail* → ;**

- The bottom-up construction of a parse tree for the same string: "A, B, C;"
- The parser finds the left-most leaf of the tree is an id. The next leaf is a comma. The parser continues in this fashion, shifting new leaves from the scanner into a forest of partially completed parse tree fragments.



Bottom-up parsing (LR)

- The bottom-up construction realizes that **some of those fragments constitute a complete right-hand side.**
- In this grammar, that occur when the parser has seen the semicolon—the right-hand side of ***id_list_tail***. With this right-hand side in hand, the parser **reduces** the semicolon to an ***id_list_tail***.
- It then **reduces** "**, id *id_list_tail***" into another ***id_list_tail***.
- After doing this one more time it is able to reduce "**id *id_list_tail***" into the root of the parse tree, ***id_list***.



Parsing

- The number in LL(1), LL(2), …, indicates how many tokens of look-ahead are required in order to parse
  - Almost all real compilers use **one token** of look-ahead
- **LL grammars requirements:**
  - **no left recursion**
  - **no common prefixes**
- Every LL(1) grammar is also LR(1), though right recursion in production tends to require very deep stacks and complicates semantic analysis

An LL(1) grammar

```
program      → stmt_list  $$(end of file)
stmt_list    → stmt stmt_list
             | ε
stmt   → id := expr
             | read id
             | write expr
expr   → term term_tail
term_tail    → add_op term term_tail
             | ε
term   → factor fact_tailt
fact_tail    → mult_op   factor   fact_tail
             | ε
factor       → ( expr )
             | id
             | number
 add_op      → +
             | -
 mult_op     → *
             | /
```

LL Parsing

- This grammar captures associativity and precedence, but most people don't find it as pretty
  - for one thing, the operands of a given operator aren't in a Right Hand Side (RHS) together!
  - however, the simplicity of the parsing algorithm makes up for this weakness
  - The first parsers were LL
- How do we parse a string with this grammar?
  - by building the parse tree incrementally

LL Parsing

- Example (the average program):

```
read A
read B
sum := A + B
write sum
write sum / 2  $$
```

- We keep a stack of non-terminals with the start symbol inserted
- We start at the top and predict needed productions on the basis of the current "left-most" non-terminal in the tree and the current input token

LL Parsing

- Table-driven LL parsing: you have a big loop in which you repeatedly look up an action in a two-dimensional table based on current leftmost non-terminal and current input token
- The actions are:

  (1) match a terminal

  (2) predict a production

  OR

  (3) announce a syntax error

LL Parsing

- First, unfold the production rules to collect for each production the possible tokens that could start it

PREDICT

1. *program* ⟶ *stmt_list* `$$` {`id`, `read`, `write`, `$$`}
2. *stmt_list* ⟶ *stmt* *stmt_list* {`id`, `read`, `write`}
3. *stmt_list* ⟶ ϵ {`$$`}
4. *stmt* ⟶ `id` `:=` *expr* {`id`}
5. *stmt* ⟶ `read` `id` {`read`}
6. *stmt* ⟶ `write` *expr* {`write`}
7. *expr* ⟶ *term* *term_tail* {`(`, `id`, `number`}
8. *term_tail* ⟶ *add_op* *term* *term_tail* {`+`, `-`}
9. *term_tail* ⟶ ϵ {`)`, `id`, `read`, `write`, `$$`}
10. *term* ⟶ *factor* *factor_tail* {`(`, `id`, `number`}
11. *factor_tail* ⟶ *mult_op* *factor* *factor_tail* {`*`, `/`}
12. *factor_tail* ⟶ ϵ {`+`, `-`, `)`, `id`, `read`, `write`, `$$`}
13. *factor* ⟶ `(` *expr* `)` {`(`}
14. *factor* ⟶ `id` {`id`}
15. *factor* ⟶ `number` {`number`}
16. *add_op* ⟶ `+` {`+`}
17. *add_op* ⟶ `-` {`-`}
18. *mult_op* ⟶ `*` {`*`}
19. *mult_op* ⟶ `/` {`/`}

LL Parsing

- Construct the ***prediction table***: for each possible input token and the left-most nonterminal, what is the possible production rule that will be used?
  - The non-terminal will be "used", while the RHS of the production is added to the stack.

PREDICT

1. *program* ⟶ *stmt_list* `$$` {`id`, `read`, `write`, `$$`}
2. *stmt_list* ⟶ *stmt* *stmt_list* {`id`, `read`, `write`}
3. *stmt_list* ⟶ ϵ {`$$`}
4. *stmt* ⟶ `id` `:=` *expr* {`id`}
5. *stmt* ⟶ `read` `id` {`read`}
6. *stmt* ⟶ `write` *expr* {`write`}
7. *expr* ⟶ *term* *term_tail* {`(`, `id`, `number`}
8. *term_tail* ⟶ *add_op* *term* *term_tail* {`+`, `-`}
9. *term_tail* ⟶ ϵ {`)`, `id`, `read`, `write`, `$$`}
10. *term* ⟶ *factor* *factor_tail* {`(`, `id`, `number`}
11. *factor_tail* ⟶ *mult_op* *factor* *factor_tail* {`*`, `/`}
12. *factor_tail* ⟶ ϵ {`+`, `-`, `)`, `id`, `read`, `write`, `$$`}
13. *factor* ⟶ `(` *expr* `)` {`(`}
14. *factor* ⟶ `id` {`id`}
15. *factor* ⟶ `number` {`number`}
16. *add_op* ⟶ `+` {`+`}
17. *add_op* ⟶ `-` {`-`}
18. *mult_op* ⟶ `*` {`*`}
19. *mult_op* ⟶ `/` {`/`}

Top-of-stack nonterminal	Current input token: id	number	read	write	:=	(	)	+	-	*	/	$$
program	1	–	1	1	–	–	–	–	–	–	–	1
stmt_list	2	–	2	2	–	–	–	–	–	–	–	3
stmt	4	–	5	6	–	–	–	–	–	–	–	–
expr	7	7	–	–	–	7	–	–	–	–	–	–
term_tail	9	–	9	9	–	–	9	8	8	–	–	9
term	10	10	–	–	–	10	–	–	–	–	–	–
factor_tail	12	–	12	12	–	–	12	12	12	11	11	12
factor	14	15	–	–	–	13	–	–	–	–	–	–
add_op	–	–	–	–	–	–	–	16	17	–	–	–
mult_op	–	–	–	–	–	–	–	–	–	18	19	–

LL Parsing

- LL(1) parse table for parsing for calculator language

```
read A
read B
sum := A + B
write sum
write sum / 2 $$
```

PREDICT

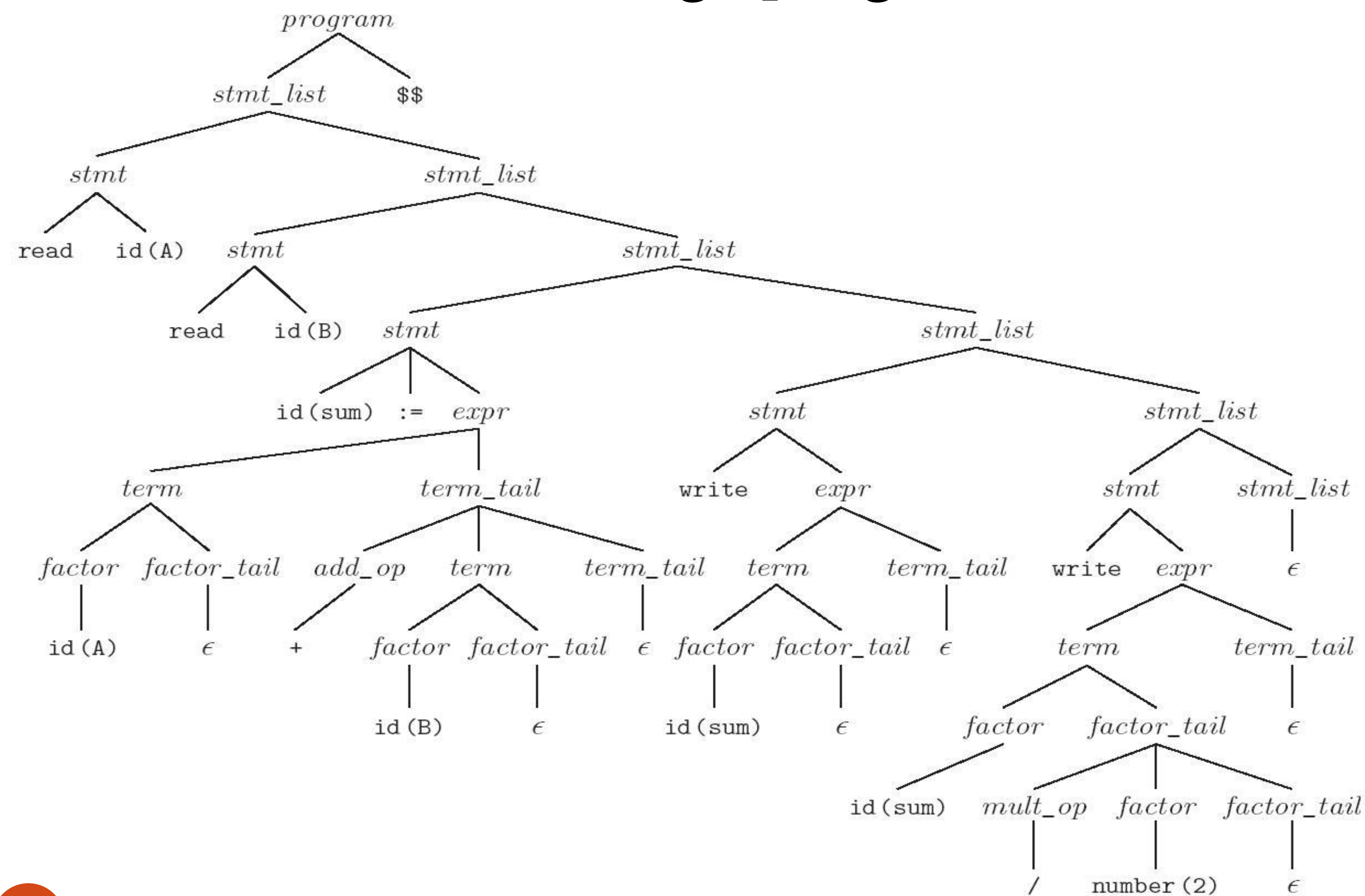
1. *program* → *stmt_list* $$ {id, read, write, $$}
2. *stmt_list* → *stmt* *stmt_list* {id, read, write}
3. *stmt_list* → ϵ {$$}
4. *stmt* → id := *expr* {id}
5. *stmt* → read id {read}
6. *stmt* → write *expr* {write}
7. *expr* → *term* *term_tail* {(, id, number}
8. *term_tail* → *add_op* *term* *term_tail* {+, -}
9. *term_tail* → ϵ {), id, read, write, $$}
10. *term* → *factor* *factor_tail* {(, id, number}
11. *factor_tail* → *mult_op* *factor* *factor_tail* {*, /}
12. *factor_tail* → ϵ {+, -,), id, read, write, $$}
13. *factor* → (*expr*) {(}
14. *factor* → id {id}
15. *factor* → number {number}
16. *add_op* → + {+}
17. *add_op* → - {-}
18. *mult_op* → * {*}
19. *mult_op* → / {/}

Top-of-stack	Current input token										
nonterminal	id	number	read	write	:=	(	)	+	-	*	/	$$
program	1	–	1	1	–	–	–	–	–	–	–	1
stmt_list	2	–	2	2	–	–	–	–	–	–	–	3
stmt	4	–	5	6	–	–	–	–	–	–	–	–
expr	7	7	–	–	–	7	–	–	–	–	–	–
term_tail	9	–	9	9	–	–	9	8	8	–	–	9
term	10	10	–	–	–	10	–	–	–	–	–	–
factor_tail	12	–	12	12	–	–	12	12	12	11	11	12
factor	14	15	–	–	–	13	–	–	–	–	–	–
add_op	–	–	–	–	–	–	–	16	17	–	–	–
mult_op	–	–	–	–	–	–	–	–	–	18	19	–

Parse stack	Input stream	Comment
program	read A read B ...	
stmt_list $$	read A read B ...	predict *program* ⟶ *stmt_list* $$
stmt stmt_list $$	read A read B ...	predict *stmt_list* ⟶ *stmt stmt_list*
read id *stmt_list* $$	read A read B ...	predict *stmt* ⟶ read id
id *stmt_list* $$	A read B ...	match read
stmt_list $$	read B sum := ...	match id
stmt stmt_list $$	read B sum := ...	predict *stmt_list* ⟶ *stmt stmt_list*
read id *stmt_list* $$	read B sum := ...	predict *stmt* ⟶ read id
id *stmt_list* $$	B sum := ...	match read
stmt_list $$	sum := A + B ...	match id
stmt stmt_list $$	sum := A + B ...	predict *stmt_list* ⟶ *stmt stmt_list*
id := *expr stmt_list* $$	sum := A + B ...	predict *stmt* ⟶ id := *expr*
:= *expr stmt_list* $$	:= A + B ...	match id
expr stmt_list $$	A + B ...	match :=
term term_tail stmt_list $$	A + B ...	predict *expr* ⟶ *term term_tail*
factor factor_tail term_tail stmt_list $$	A + B ...	predict *term* ⟶ *factor factor_tail*
id *factor_tail term_tail stmt_list* $$	A + B ...	predict *factor* ⟶ id
factor_tail term_tail stmt_list $$	+ B write sum ...	match id
term_tail stmt_list $$	+ B write sum ...	predict *factor_tail* ⟶ ϵ
add_op term term_tail stmt_list $$	+ B write sum ...	predict *term_tail* ⟶ *add_op term term_tail*
+ *term term_tail stmt_list* $$	+ B write sum ...	predict *add_op* ⟶ +
term term_tail stmt_list $$	B write sum ...	match +
factor factor_tail term_tail stmt_list $$	B write sum ...	predict *term* ⟶ *factor factor_tail*
id *factor_tail term_tail stmt_list* $$	B write sum ...	predict *factor* ⟶ id
factor_tail term_tail stmt_list $$	write sum ...	match id
term_tail stmt_list $$	write sum write ...	predict *factor_tail* ⟶ ϵ
stmt_list $$	write sum write ...	predict *term_tail* ⟶ ϵ
stmt stmt_list $$	write sum write ...	predict *stmt_list* ⟶ *stmt stmt_list*
write *expr stmt_list* $$	write sum write ...	predict *stmt* ⟶ write *expr*

expr stmt_list `$$`	`sum write sum / 2`	match `write`
term term_tail stmt_list `$$`	`sum write sum / 2`	predict *expr* $\longrightarrow$ *term term_tail*
factor factor_tail term_tail stmt_list `$$`	`sum write sum / 2`	predict *term* $\longrightarrow$ *factor factor_tail*
`id` *factor_tail term_tail stmt_list* `$$`	`sum write sum / 2`	predict *factor* $\longrightarrow$ `id`
factor_tail term_tail stmt_list `$$`	`write sum / 2`	match `id`
term_tail stmt_list `$$`	`write sum / 2`	predict *factor_tail* $\longrightarrow \epsilon$
stmt_list `$$`	`write sum / 2`	predict *term_tail* $\longrightarrow \epsilon$
stmt stmt_list `$$`	`write sum / 2`	predict *stmt_list* $\longrightarrow$ *stmt stmt_list*
`write` *expr stmt_list* `$$`	`write sum / 2`	predict *stmt* $\longrightarrow$ `write` *expr*
expr stmt_list `$$`	`sum / 2`	match `write`
term term_tail stmt_list `$$`	`sum / 2`	predict *expr* $\longrightarrow$ *term term_tail*
factor factor_tail term_tail stmt_list `$$`	`sum / 2`	predict *term* $\longrightarrow$ *factor factor_tail*
`id` *factor_tail term_tail stmt_list* `$$`	`sum / 2`	predict *factor* $\longrightarrow$ `id`
factor_tail term_tail stmt_list `$$`	`/ 2`	match `id`
mult_op factor factor_tail term_tail stmt_list `$$`	`/ 2`	predict *factor_tail* $\longrightarrow$ *mult_op factor factor_tail*
`/` *factor factor_tail term_tail stmt_list* `$$`	`/ 2`	predict *mult_op* $\longrightarrow$ `/`
factor factor_tail term_tail stmt_list `$$`	`2`	match `/`
`number` *factor_tail term_tail stmt_list* `$$`	`2`	predict *factor* $\longrightarrow$ `number`
factor_tail term_tail stmt_list `$$`		match `number`
term_tail stmt_list `$$`		predict *factor_tail* $\longrightarrow \epsilon$
stmt_list `$$`		predict *term_tail* $\longrightarrow \epsilon$
`$$`		predict *stmt_list* $\longrightarrow \epsilon$

Parse tree for the average program



LL Parsing

- **Problems trying to make a grammar LL(1)**
  - **left recursion**
    - example:

```
id_list     → id_list , id
id_list     → id
```

    - **we can get rid of all left recursion mechanically in any grammar**

```
id_list          → id  id_list_tail
id_list_tail     → , id id_list_tail
id_list_tail     →  ε
```

LL Parsing

- **Problems trying to make a grammar LL(1)**
  - **common prefixes**
    - example:

```
stmt → id := expr
     | id ( arg_list )
```

    - **we can eliminate left-factor mechanically = "*left-factoring*"**

```
stmt → id  id_stmt_tail
id_stmt_tail → := expr
             | ( arg_list)
```

LL Parsing

- Eliminating left recursion and common prefixes still does NOT make a grammar LL
  - there are infinitely many non-LL LANGUAGES, and the mechanical transformations work on them just fine
- Problems trying to make a grammar LL(1)
  - the"dangling else" problem prevents grammars from being LL(1) (or in fact LL(k) for any k)
    - the following natural (Pascal) grammar fragment is ambiguous:

```
stmt → if cond then_clause else_clause
       | other_stuff
then_clause → then  stmt
else_clause → else  stmt   | ε
```

Example String: "**if C1 then if C2 then S1 else S2**"

Ambiguity: the else can be paired with either if then!!!

LL Parsing

- Desired effect: pair the else with the nearest then.
- The less natural grammar fragment:

```
stmt  → balanced_stmt | unbalanced_stmt
balanced_stmt → if cond then balanced_stmt
                    else balanced_stmt
              | other_stuff
unbalanced_stmt → if cond then stmt
              | if cond then balanced_stmt
                    else unbalanced_stmt
```

- A **balanced_stmt** is one with the same number of **then**s and **else**s.
- An **unbalanced_stmt** has more **then**s.

LL Parsing

- The usual approach, whether top-down OR bottom-up, is to use the ambiguous grammar together with a disambiguating rule that says:
  - else goes with the closest then or
  - more generally, the first of two possible productions is the one to predict (or reduce)

```
stmt → if cond then_clause else_clause
        | other_stuff
then_clause → then  stmt
else_clause → else  stmt   | ε
```

LL Parsing

- Better yet, languages (since Pascal) generally employ **explicit end-markers**, which eliminate this problem.
- In Modula-2, for example, one says:

```
if A = B then
    if C = D then E := F end
else
    G := H
end
```

- Ada says '**`end if`**'; other languages say '**`fi`**'

LL Parsing

- One problem with end markers is that they tend to bunch up. In Pascal you say

```
if A = B then …
else if A = C then …
else if A = D then …
else if A = E then …
else ...;
```

  - With end markers this becomes

```
if A = B then …
else if A = C then …
else if A = D then …
else if A = E then …
else ...;
end; end; end; end; end; end; …
```

LR Parsing

- LR parsers are almost always **table-driven**:
  - like a table-driven LL parser, an LR parser uses a **big loop** in which it repeatedly inspects a two-dimensional table to find out what action to take
  - unlike the LL parser, however, the LR driver has non-trivial **state** (like a DFA), and the table is indexed by current input token and current **state**
    - also the stack contains a record of what has been seen SO FAR (NOT what is expected)

LR Parsing

- LR keeps the **roots of its partially completed subtrees** on a stack
  - When it accepts a new token from the scanner, it ***shifts*** the token into the stack
  - When it recognizes that the top few symbols on the stack constitute a right-hand side, it ***reduces*** those symbols to their left-hand side by popping them off the stack and pushing the left-hand side in their place

LR Parsing

- ***Rightmost (canonical) derivation*** for the identifiers grammar:

Stack contents (roots of partial trees)	Remaining input
ϵ	`A, B, C;`
`id (A)`	`, B, C;`
`id (A) ,`	`B, C;`
`id (A) , id (B)`	`, C;`
`id (A) , id (B) ,`	`C;`
`id (A) , id (B) , id (C)`	`;`
`id (A) , id (B) , id (C) ;`	
`id (A) , id (B) , id (C)` *id_list_tail*	
`id (A) , id (B)` *id_list_tail*	
`id (A)` *id_list_tail*	
id_list	

LR Parsing

- LR(1) grammar for the calculator language:

1. *program* $\longrightarrow$ *stmt_list* `$$`
2. *stmt_list* $\longrightarrow$ *stmt_list stmt*
3. *stmt_list* $\longrightarrow$ *stmt*
4. *stmt* $\longrightarrow$ `id := ` *expr*
5. *stmt* $\longrightarrow$ `read id`
6. *stmt* $\longrightarrow$ `write` *expr*
7. *expr* $\longrightarrow$ *term*
8. *expr* $\longrightarrow$ *expr add_op term*
9. *term* $\longrightarrow$ *factor*
10. *term* $\longrightarrow$ *term mult_op factor*
11. *factor* $\longrightarrow$ `(` *expr* `)`
12. *factor* $\longrightarrow$ `id`
13. *factor* $\longrightarrow$ `number`
14. *add_op* $\longrightarrow$ `+`
15. *add_op* $\longrightarrow$ `-`
16. *mult_op* $\longrightarrow$ `*`
17. *mult_op* $\longrightarrow$ `/`

LR Parsing

- Example (the average program):

```
read A
read B
sum := A + B
write sum
write sum / 2  $$
```

LR Parsing

- When we begin execution, the parse stack is **empty** and we are at the beginning of the production for program:

  `program → . stmt_list $$`

  - When augmented with a **`.`**, a production is called an *LR item*
  - This original item (**`program → . stmt_list $$`**) is called the ***basis*** of the list.

LR Parsing

- Since the **.** in this item is immediately in front of a nonterminal—namely **`stmt_list`** —we may be about to see the yield of that nonterminal coming up on the input.

```
program → . stmt_list $$
stmt_list → . stmt_list stmt
stmt_list → . stmt
```

LR Parsing

- Since **`stmt`** is a nonterminal, we may also be at the beginning of any production whose left-hand side is **`stmt`**:

```
program → . stmt_list $$
stmt_list → . stmt_list stmt
stmt_list → . stmt
stmt → . id := expr
stmt → . read id
stmt → . write expr
```

  - The additional items to the basis are its ***closure***.

LR Parsing

- Our upcoming token is a **`read`**
  - Once we shift it onto the stack, we know we are in the following state:

`stmt` → **`read . id`**

  - This state has a single basis item and an empty closure—the **`.`** precedes a terminal.
- After shifting the A, we have:

`stmt` → **`read id .`**

LR Parsing

- We now know that **`read id`** is the handle, and we must reduce.
  - The reduction **pops** two symbols off the parse stack and pushes a **`stmt`** in their place
  - Since one of the items in State 0 was

`stmt_list → . stmt`

we now have

`stmt_list → stmt .`

Again we must reduce: remove the **`stmt`** from the stack and push a **`stmt_list`** in its place.

LR Parsing

- Our new state:

```
program → stmt_list . $$
stmt_list → stmt_list . stmt
stmt → . id := expr
stmt → . read id
stmt → . write expr
```

	State	Transitions
0.	*program* ⟶ • *stmt_list* $$	on *stmt_list* shift and goto 2
	stmt_list ⟶ • *stmt_list* *stmt*	
	stmt_list ⟶ • *stmt*	on *stmt* shift and reduce (pop 1 state, push *stmt_list* on input)
	stmt ⟶ • `id` `:=` *expr*	on `id` shift and goto 3
	stmt ⟶ • `read` `id`	on `read` shift and goto 1
	stmt ⟶ • `write` *expr*	on `write` shift and goto 4
1.	*stmt* ⟶ `read` • `id`	on `id` shift and reduce (pop 2 states, push *stmt* on input)
2.	*program* ⟶ *stmt_list* • $$	on $$ shift and reduce (pop 2 states, push *program* on input)
	stmt_list ⟶ *stmt_list* • *stmt*	on *stmt* shift and reduce (pop 2 states, push *stmt_list* on input)
	stmt ⟶ • `id` `:=` *expr*	on `id` shift and goto 3
	stmt ⟶ • `read` `id`	on `read` shift and goto 1
	stmt ⟶ • `write` *expr*	on `write` shift and goto 4
3.	*stmt* ⟶ `id` • `:=` *expr*	on `:=` shift and goto 5
4.	*stmt* ⟶ `write` • *expr*	on *expr* shift and goto 6
	expr ⟶ • *term*	on *term* shift and goto 7
	expr ⟶ • *expr* *add_op* *term*	
	term ⟶ • *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ • *term* *mult_op* *factor*	
	factor ⟶ • `(` *expr* `)`	on `(` shift and goto 8
	factor ⟶ • `id`	on `id` shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • `number`	on `number` shift and reduce (pop 1 state, push *factor* on input)

5.	*stmt* ⟶ `id` `:=` • *expr*	on *expr* shift and goto 9
	expr ⟶ • *term*	on *term* shift and goto 7
	expr ⟶ • *expr add_op term*	
	term ⟶ • *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ • *term mult_op factor*	
	factor ⟶ • `(` *expr* `)`	on `(` shift and goto 8
	factor ⟶ • `id`	on `id` shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • `number`	on `number` shift and reduce (pop 1 state, push *factor* on input)
6.	*stmt* ⟶ `write` *expr* •	on FOLLOW(*stmt*) = {`id`, `read`, `write`, `$$`} reduce (pop 2 states, push *stmt* on input)
	expr ⟶ *expr* • *add_op term*	on *add_op* shift and goto 10
	add_op ⟶ • `+`	on `+` shift and reduce (pop 1 state, push *add_op* on input)
	add_op ⟶ • `-`	on `-` shift and reduce (pop 1 state, push *add_op* on input)
7.	*expr* ⟶ *term* •	on FOLLOW(*expr*) = {`id`, `read`, `write`, `$$`, `)`, `+`, `-`} reduce (pop 1 state, push *expr* on input)
	term ⟶ *term* • *mult_op factor*	on *mult_op* shift and goto 11
	mult_op ⟶ • `*`	on `*` shift and reduce (pop 1 state, push *mult_op* on input)
	mult_op ⟶ • `/`	on `/` shift and reduce (pop 1 state, push *mult_op* on input)

8.	$factor \longrightarrow (\bullet expr)$	on $expr$ shift and goto 12
	$expr \longrightarrow \bullet term$	on $term$ shift and goto 7
	$expr \longrightarrow \bullet expr\ add\_op\ term$	
	$term \longrightarrow \bullet factor$	on $factor$ shift and reduce (pop 1 state, push $term$ on input)
	$term \longrightarrow \bullet term\ mult\_op\ factor$	
	$factor \longrightarrow \bullet (expr)$	on (shift and goto 8
	$factor \longrightarrow \bullet$ `id`	on `id` shift and reduce (pop 1 state, push $factor$ on input)
	$factor \longrightarrow \bullet$ `number`	on `number` shift and reduce (pop 1 state, push $factor$ on input)
9.	$stmt \longrightarrow$ `id :=` $expr \bullet$	on FOLLOW ($stmt$) = {`id`, `read`, `write`, `$$`} reduce (pop 3 states, push $stmt$ on input)
	$expr \longrightarrow expr \bullet add\_op\ term$	on add_op shift and goto 10
	$add\_op \longrightarrow \bullet$ `+`	on `+` shift and reduce (pop 1 state, push add_op on input)
	$add\_op \longrightarrow \bullet$ `-`	on `-` shift and reduce (pop 1 state, push add_op on input)
10.	$expr \longrightarrow expr\ add\_op \bullet term$	on $term$ shift and goto 13
	$term \longrightarrow \bullet factor$	on $factor$ shift and reduce (pop 1 state, push $term$ on input)
	$term \longrightarrow \bullet term\ mult\_op\ factor$	
	$factor \longrightarrow \bullet (expr)$	on (shift and goto 8
	$factor \longrightarrow \bullet$ `id`	on `id` shift and reduce (pop 1 state, push $factor$ on input)
	$factor \longrightarrow \bullet$ `number`	on `number` shift and reduce (pop 1 state, push $factor$ on input)

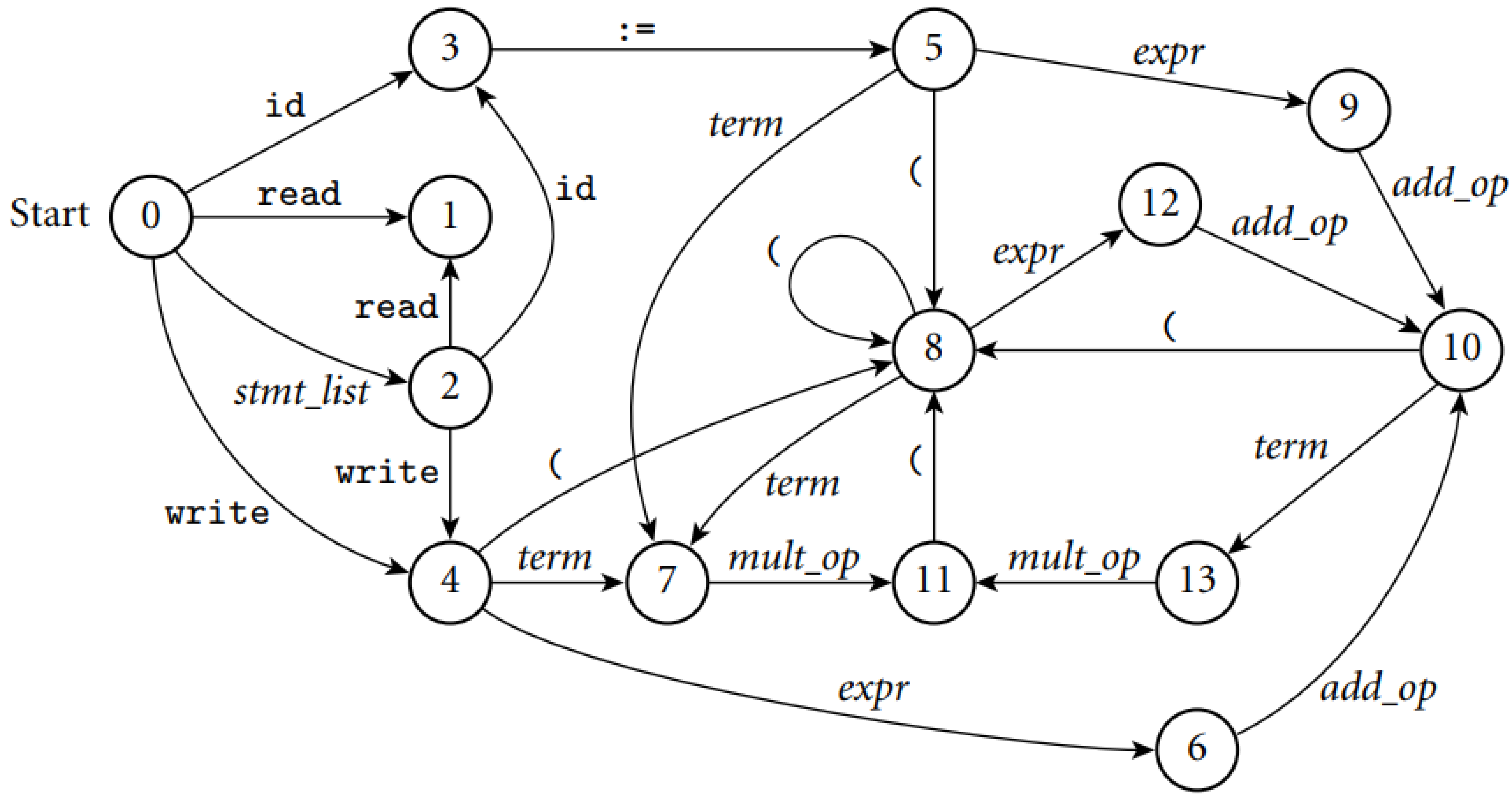
11.	*term* $\longrightarrow$ *term mult_op* $\bullet$ *factor*	on *factor* shift and reduce (pop 3 states, push *term* on input)
	factor $\longrightarrow$ $\bullet$ `(` *expr* `)`	on `(` shift and goto 8
	factor $\longrightarrow$ $\bullet$ `id`	on `id` shift and reduce (pop 1 state, push *factor* on input)
	factor $\longrightarrow$ $\bullet$ `number`	on `number` shift and reduce (pop 1 state, push *factor* on input)
12.	*factor* $\longrightarrow$ `(` *expr* $\bullet$ `)`	on `)` shift and reduce (pop 3 states, push *factor* on input)
	expr $\longrightarrow$ *expr* $\bullet$ *add_op term*	on *add_op* shift and goto 10
	add_op $\longrightarrow$ $\bullet$ `+`	on `+` shift and reduce (pop 1 state, push *add_op* on input)
	add_op $\longrightarrow$ $\bullet$ `-`	on `-` shift and reduce (pop 1 state, push *add_op* on input)
13.	*expr* $\longrightarrow$ *expr add_op term* $\bullet$	on FOLLOW(*expr*) = {`id`, `read`, `write`, `$$`, `)`, `+`, `-`} reduce (pop 3 states, push *expr* on input)
	term $\longrightarrow$ *term* $\bullet$ *mult_op factor*	on *mult_op* shift and goto 11
	mult_op $\longrightarrow$ $\bullet$ `*`	on `*` shift and reduce (pop 1 state, push *mult_op* on input)
	mult_op $\longrightarrow$ $\bullet$ `/`	on `/` shift and reduce (pop 1 state, push *mult_op* on input)

Top-of-stack state	Current input symbol																		
	sl	*s*	*e*	*t*	*f*	*ao*	*mo*	`id`	`lit`	`r`	`w`	`:=`	`(`	`)`	`+`	`-`	`*`	`/`	`$$`
0	s2	b3	–	–	–	–	–	s3	–	s1	s4	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	b5	–	–	–	–	–	–	–	–	–	–	–
2	–	b2	–	–	–	–	–	s3	–	s1	s4	–	–	–	–	–	–	–	b1
3	–	–	–	–	–	–	–	–	–	–	–	s5	–	–	–	–	–	–	–
4	–	–	s6	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
5	–	–	s9	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
6	–	–	–	–	–	s10	–	r6	–	r6	r6	–	–	–	b14	b15	–	–	r6
7	–	–	–	–	–	–	s11	r7	–	r7	r7	–	–	r7	r7	r7	b16	b17	r7
8	–	–	s12	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
9	–	–	–	–	–	s10	–	r4	–	r4	r4	–	–	–	b14	b15	–	–	r4
10	–	–	–	s13	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
11	–	–	–	–	b10	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
12	–	–	–	–	–	s10	–	–	–	–	–	–	–	b11	b14	b15	–	–	–
13	–	–	–	–	–	–	s11	r8	–	r8	r8	–	–	r8	r8	r8	b16	b17	r8

Table entries indicate whether to shift (s), reduce (r), or shift and then reduce (b). The accompanying number is the new state when shifting, or the production that has been recognized when (shifting and) reducing

Driver for a table-driven LR(1) parser

```
parse_stack.push(⟨null, start_state⟩)
cur_sym : symbol := scan                            –– get new token from scanner
loop
    cur_state : state := parse_stack.top.st         –– peek at state at top of stack
    if cur_state = start_state and cur_sym = start_symbol
        return                                      –– success!
    ar : action_rec := parse_tab[cur_state, cur_sym]
    case ar.action
        shift:
            parse_stack.push(⟨cur_sym, ar.new_state⟩)
            cur_sym := scan                         –– get new token from scanner
        reduce:
            cur_sym := prod_tab[ar.prod].lhs
            parse_stack.pop(prod_tab[ar.prod].rhs_len)
        shift_reduce:
            cur_sym := prod_tab[ar.prod].lhs
            parse_stack.pop(prod_tab[ar.prod].rhs_len–1)
        error:
            parse_error
```


Start
0
1
2
3
4
5
6
7
8
9
10
11
12
13
id
read
id
read
stmt_list
write
write
:=
term
(
expr
add_op
add_op
expr
(
(
term
(
term
(
term
mult_op
mult_op
expr
add_op

Parsing summary

- A **scanner** is a DFA
  - it can be specified with a state diagram
- An LL or LR **parser** is a PDA (push down automata)
  - a PDA can be specified with a ***state diagram*** and a stack
    - the state diagram looks just like a DFA state diagram, except the arcs are labeled with **`<input symbol, top-of-stack symbol>`** pairs, and in addition to moving to a new state the PDA has the option of pushing or popping a finite number of symbols onto/off the stack
  - Early's algorithm does NOT use PDAs, but dynamic programming

Actions

- We can run actions when a rule triggers:
  - Often used to construct an AST for a compiler.
  - For simple languages, can interpret code directly
  - We can use actions to fix the Top-Down Parsing problems

Programming

- A *compiler-compiler* (or *parser generator*, *compiler generator*) is a programming tool that creates a parser, interpreter, or compiler from some form of formal description of a language and machine
  - the input is a grammar (usually in BNF) of a programming language
  - the generated output is the source code of a parser
- Examples of parser generators:
  - classical parsing tools: **`lex`**, **`Yacc`**, **`bison`**, **`flex`**, **`ANTLR`**
  - **`PLY`**: python implementation of **`lex`** and **`yacc`**
  - Python **`TPG`** parser
  - **`ANTLR`** for python

Classic Parsing Tools

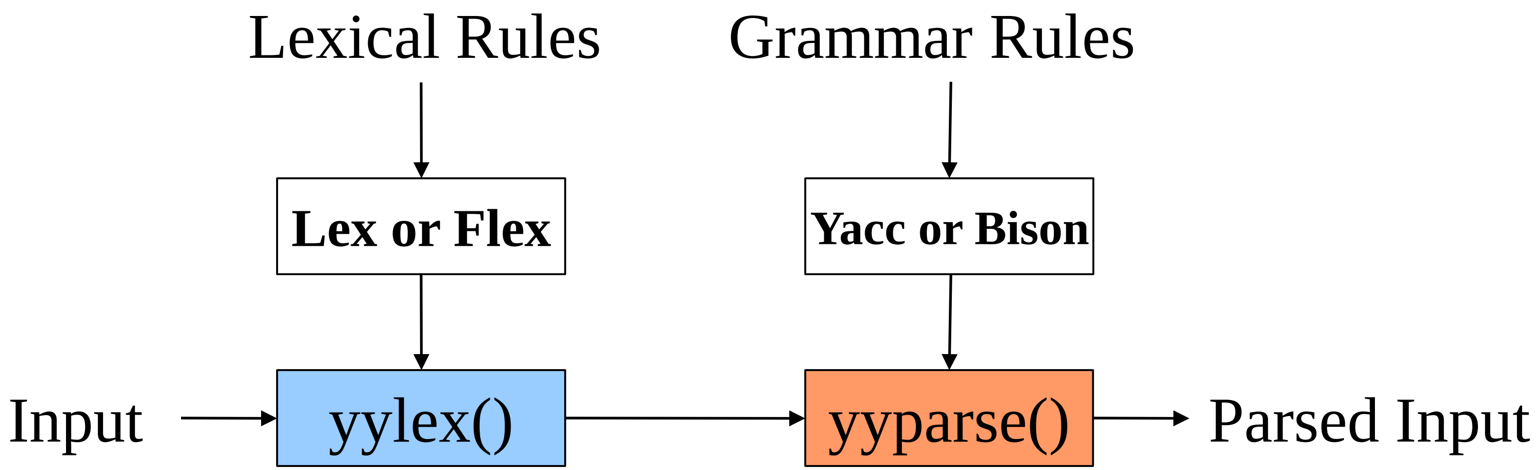
`lex` - original UNIX Lexical analysis (tokenizing) generator

- create a C function that will parse input according to a set of regular expressions

`yacc` - Yet Another Compiler Compiler (parsing)

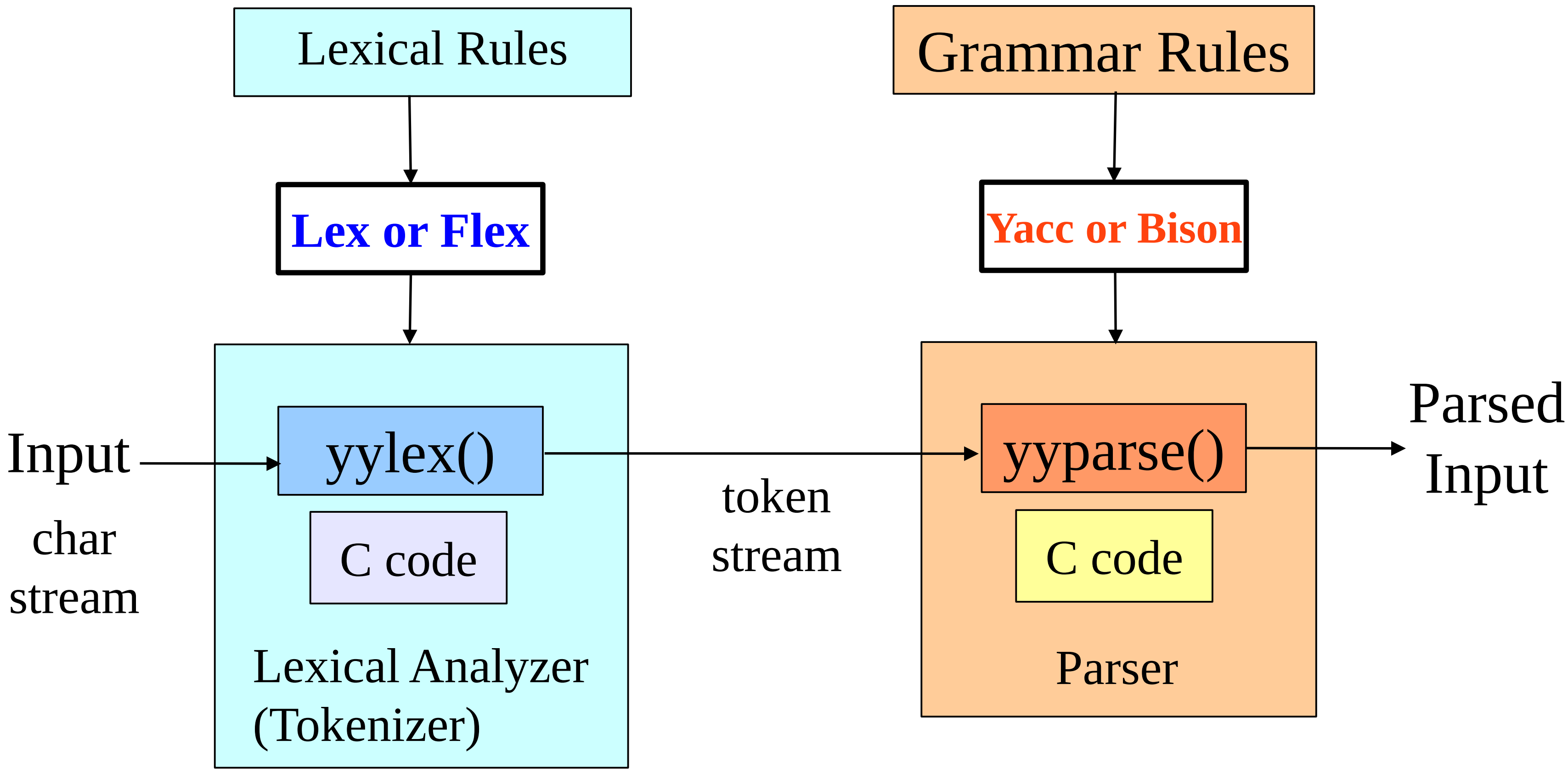
- generate a C program for a parser from BNF rules

`bison` and **`flex`** ("**`f`**ast **`lex`**") - more powerful, free versions of yacc and lex, from GNU Software Fnd'n.

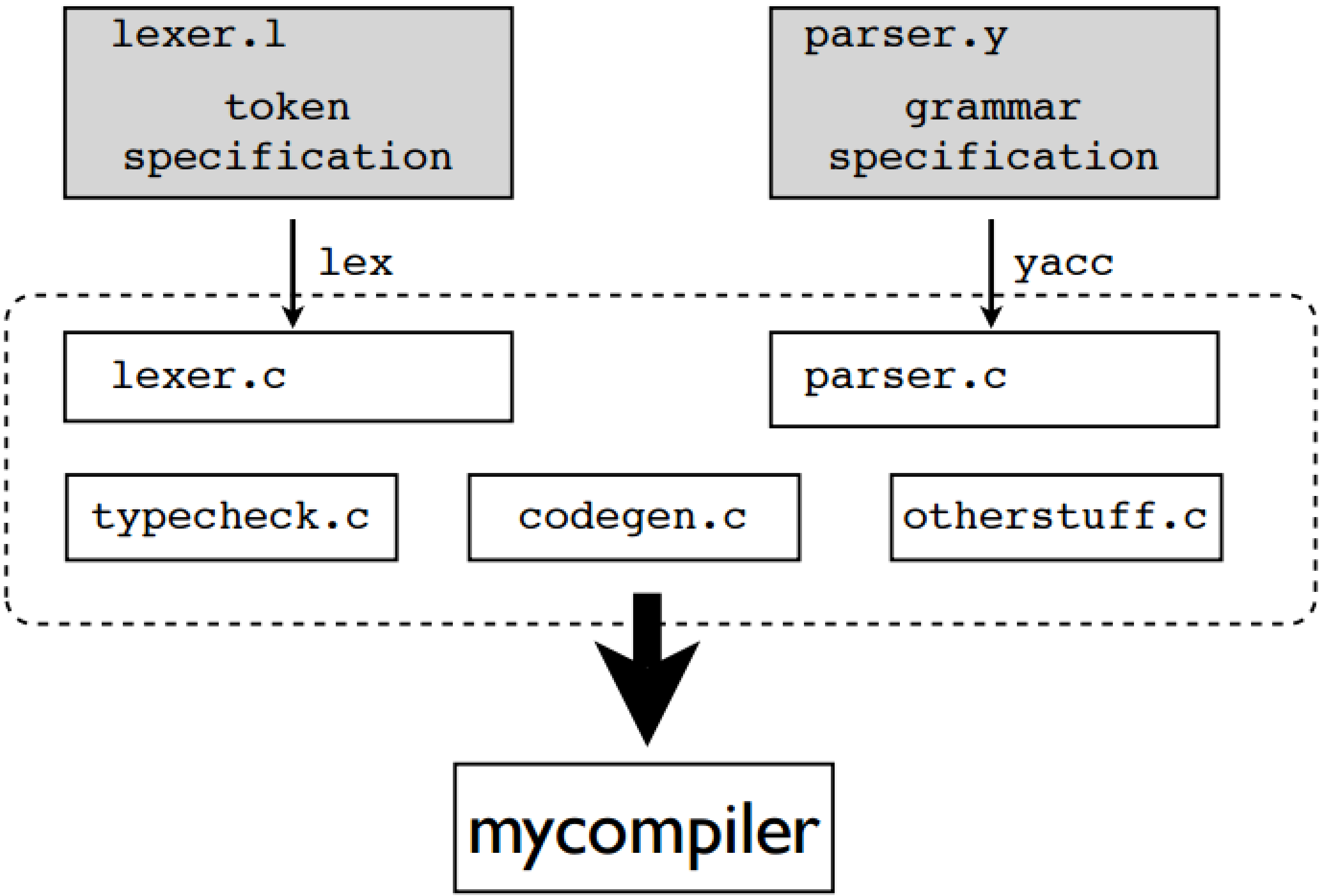


Classic Parsing Tools

- Lex and Yacc generate C code for your analyzer & parser

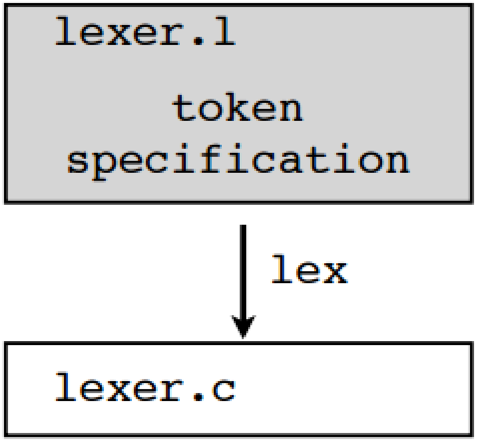


Lex and Yacc the big picture



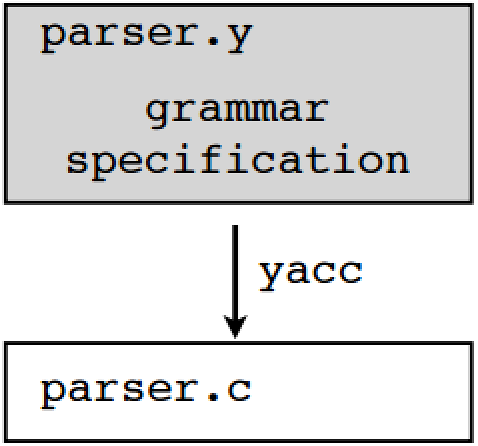
Lex Example

```
/* lexer.l */

%{
#include “header.h”
int lineno = 1;
%}
%%
[ \t]* ; /* Ignore whitespace */
\n { lineno++; }
[0-9]+ { yylval.val = atoi(yytext);
             return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.name = strdup(yytext);
                          return ID; }
\+ { return PLUS; }
- { return MINUS; }
\* { return TIMES; }
\/ { return DIVIDE; }
= { return EQUALS; }
%%
```



Yacc Example

```
/* parser.y */

%{
#include "header.h"
%}
%union {
        char *name;
        int val;
}
%token PLUS MINUS TIMES DIVIDE EQUALS
%token<name> ID;
%token<val> NUMBER;
%%
start : ID EQUALS expr;
expr : expr PLUS term
        | expr MINUS term
        | term
        ;
...
```



Bison Overview

`myparser.y`

BNF rules and actions for your grammar.

The programmer puts BNF rules and token rules for the parser he wants in a bison source file `myparser.y`

```
> bison myparser.y
```

run bison to create a C program (*.tab.c) containing a parser function.

The programmer must also supply a tokenizer named yylex( )

`myparser.tab.c`

parser source code

`yylex.c`

tokenizer function in C

```
> gcc -o myprog myparser.tab.c yylex.c
```

`myprog`

executable program

PLY

- PLY: Python Lex-Yacc = an implementation of lex and yacc parsing tools for Python by David Beazley: http://www.dabeaz.com/ply/
- A bit of history:
  - Yacc : ~1973. Stephen Johnson (AT&T)
  - Lex : ~1974. Eric Schmidt and Mike Lesk (AT&T)
  - PLY: 2001

PLY

- PLY is not a code generator
- PLY consists of two Python modules

  ply.lex = A module for writing lexers

    Tokens specified using regular expressions

    Provides functions for reading input text

  ply.yacc = A module for writing grammars

  - You simply import the modules to use them
    - The grammar must be in a file

PLY

- ply.lex example:

```
import ply.lex as lex
tokens = [ ‘NAME’,’NUMBER’,’PLUS’,’MINUS’,’TIMES’,
 ’DIVIDE’, EQUALS’ ]
t_ignore = ‘ \t’
t_PLUS = r’\+’
t_MINUS = r’-’
t_TIMES = r’\*’
t_DIVIDE = r’/’
t_EQUALS = r’=’
t_NAME = r’[a-zA-Z_][a-zA-Z0-9_]*’

def t_NUMBER(t):
    r’\d+’
    t.value = int(t.value)
    return t

lex.lex() # Build the lexer
```

tokens list specifies all of the possible tokens

Each token has a matching declaration of the form t_*TOKNAME*

Functions are used when special action code must execute

Builds the lexer by creating a master regular expression

PLY

- Two functions: input() and token()

```
lex.lex() # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break

     # Use token
```

input() feeds a string into the lexer

token() returns the next token or None

```
tok.type
tok.value
tok.line
tok.lexpos
```

t_NAME = r'[a-zA-Z_][a-zA-Z0-9_]*'

matching text

Position in input text

PLY

token information imported from lexer

```
import ply.yacc as yacc
import mylexer                    # Import lexer information
tokens = mylexer.tokens           # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''
def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''
def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''
def p_factor(p):
    '''factor : NUMBER'''

yacc.yacc() # Build the parser
data = "x = 3*4+5*6"
yacc.parse(data) # Parse some text
```

grammar rules encoded as functions with names p_*rulename*

docstrings contain grammar rules from BNF

PLY

- PLY uses LR-parsing
  - *Shift-reduce* parsing
    - Input tokens are shifted onto a parsing stack

X = 3 * 4 + 5	**->**	
= 3 * 4 + 5	**->**	**NAME**
3 * 4 + 5	**->**	**NAME =**
*** 4 + 5**	**->**	**NAME = NUM**

    - This continues until a complete grammar rule appears on the top of the stack

reduce **factor : NUM**

*** 4 + 5**	**->**	**NAME = factor**

PLY

- During reduction, rule functions are invoked

```
def p_factor(p):
    'factor : NUMBER'
```

- Parameter p contains grammar symbol values

```
def p_factor(p):
    'factor : NUMBER'
```

p[0] p[1]

PLY

- Rule functions generally process values on right hand side of grammar rule
- Result is then stored in left hand side
- Results propagate up through the grammar
- PLY does Bottom-up parsing

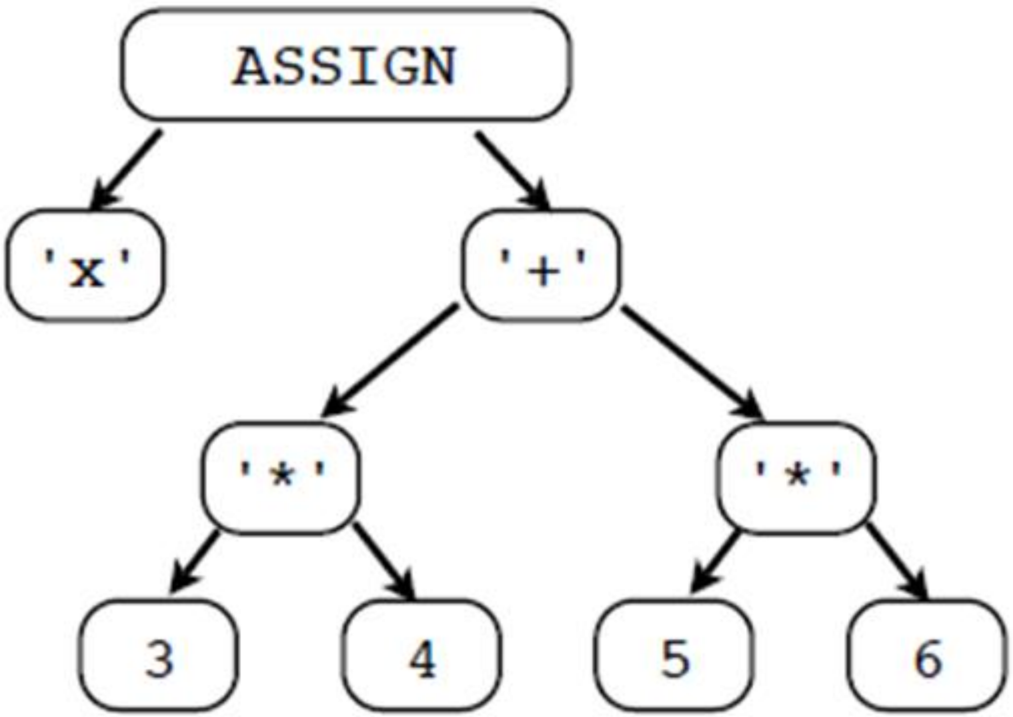
PLY Calculator Example

```
def p_assign(p):
    '''assign : NAME EQUALS expr'''
    vars[p[1]] = p[3]

def p_expr_plus(p):
    '''expr : expr PLUS term'''
    p[0] = p[1] + p[3]

def p_term_mul(p):
    '''term : term TIMES factor'''
    p[0] = p[1] * p[3]

def p_term_factor(p):
    '''term : factor'''
    p[0] = p[1]

def p_factor(p):
    '''factor : NUMBER'''
    p[0] = p[1]
```

Build a parse tree using tuples

```
def p_assign(p):
    '''assign : NAME EQUALS expr'''
    p[0] = ('ASSIGN',p[1],p[3])

def p_expr_plus(p):
    '''expr : expr PLUS term'''
    p[0] = ('+',p[1],p[3])

def p_term_mul(p):
    '''term : term TIMES factor'''
    p[0] = ('*',p[1],p[3])

def p_term_factor(p):
    '''term : factor'''
    p[0] = p[1]

def p_factor(p):
    '''factor : NUMBER'''
    p[0] = ('NUM',p[1])
```

```
>>> t = yacc.parse("x = 3*4 + 5*6")
>>> t
('ASSIGN','x',('+',
                    ('*',('NUM',3),('NUM',4)),
                    ('*',('NUM',5),('NUM',6))
               )
)
```



PLY Precedence Specifiers

- Precedence Specifiers (most precedence at bottom):

```
precedence = (
        ('left','PLUS','MINUS'),
        ('left','TIMES','DIVIDE'),
        ('nonassoc','UMINUS'),
)
def p_expr_uminus(p):
        'expr : MINUS expr %prec UMINUS'
        p[0] = -p[1]
...
```

PLY Best Documentation

- Google Mailing list/group:

  http://groups.google.com/group/ply-hack

TPG

- TGP is a lexical and syntactic parser generator for Python.
  - YACC is too complex to use in simple cases (calculators, configuration files, small programming languages, …).
  - You can also add Python code directly into grammar rules and build abstract syntax trees while parsing.

Python TPG Lexer

- Toy Parser Generator (TPG): http://cdsoft.fr/tpg
  - Syntax:

```
token <name> <regex> <function> ;
separator <name> <regex>;
```

  - Example:

```
token integer '\d+' int;
token float '\d+\.\d*|\.\d+' float;
token rbrace '{';
separator space '\s+';
```

Python TPG Lexer

- Embed TPG in Python:

```
import tpg
class Calc:
  r"""
  separator spaces: '\s+' ;
  token number: '\d+' ;
  token add: '[+-]' ;
  token mul: '[*/]' ;
  """
```

Try it in Python: download TGP from http://cdsoft.fr/tpg

TPG example

- Defining the grammar:
  - Non-terminal productions:

```
START -> Expr ;
Expr -> Term ( add Term )* ;
Term -> Fact ( mul Fact )* ;
Fact -> number | '\(' Expr '\)' ;
```

TPG example

```
import tpg
class Calc:
  r"""
  separator spaces: '\s+' ;
  token number: '\d+' ;
  token add: '[+-]' ;
  token mul: '[*/]' ;
  START -> Expr ;
  Expr -> Term ( add Term )* ;
  Term -> Fact ( mul Fact )* ;
  Fact -> number | '\(' Expr '\)' ;
  """
```

TPG example

- Reading the input and returning values:

```
separator spaces: '\s+' ;
token number: '\d+' int ;
token add: '[+-]' make_op;
token mul: '[*/]' make_op;
```

- Transform tokens into defined operations:

```
def make_op(s):
  return {
  '+': lambda x,y: x+y,
  '-': lambda x,y: x-y,
  '*': lambda x,y: x*y,
  '/': lambda x,y: x/y,
  }[s]
```

TPG example

- After a terminal symbol is recognized we will store it in a Python variable: for example to save a number in a variable **`n`**: **`number/n`**.
- Include Python code example:

```
Expr/t -> Term/t ( add/op Term/f $t=op(t,f)$ )* ;
Term/f -> Fact/f ( mul/op Fact/a $f=op(f,a)$ )* ;
Fact/a -> number/a | '\(' Expr/a '\)' ;
```

```
import math                          # Simple calculator calc.py
import operator
import string
import tpg
def make_op(s):
      return {
            '+': lambda x,y: x+y,
            '-': lambda x,y: x-y,
            '*': lambda x,y: x*y,
            '/': lambda x,y: x/y,
      }[s]
class Calc(tpg.Parser):
      r"""
      separator spaces: '\s+' ;
      token number: '\d+' int ;
      token add: '[+-]' make_op ;
      token mul: '[*/]' make_op ;
      START/e -> Term/e ;
```

```
      Term/t -> Fact/t ( add/op Fact/f $ t = op(t,f) $ )* ;
      Fact/f -> Atom/f ( mul/op Atom/a $ f = op(f,a) $ )* ;
      Atom/a -> number/a | '\(' Term/a '\)' ;
      """
calc = Calc()

if tpg.__python__ == 3:
    operator.div = operator.truediv
    raw_input = input

expr = raw_input('Enter an expression: ')
print(expr, '=', calc(expr))
```

```
#!/usr/bin/env python
# Larger example: scientific_calc.py
import math
import operator
import string
import tpg
if tpg.__python__ == 3:
    operator.div = operator.truediv
    raw_input = input
def make_op(op):
    return {
        '+'   : operator.add,
        '-'   : operator.sub,
        '*'   : operator.mul,
        '/'   : operator.div,
        '%'   : operator.mod,
        '^'   : lambda x,y:x**y,
        '**'  : lambda x,y:x**y,
        'cos' : math.cos,
        'sin' : math.sin,
        'tan' : math.tan,
        'acos': math.acos,
```

```
        'asin': math.asin,
        'atan': math.atan,
        'sqr' : lambda x:x*x,
        'sqrt': math.sqrt,
        'abs' : abs,
        'norm': lambda x,y:math.sqrt(x*x+y*y),
    }[op]
class Calc(tpg.Parser, dict):
    r"""
        separator space '\s+' ;
        token pow_op    '\^|\*\*' $ make_op
        token add_op    '[+-]'    $ make_op
        token mul_op    '[*/%]'   $ make_op
        token funct1    '(cos|sin|tan|acos|asin|atan|sqr|sqrt|abs)\b' $ make_op
        token funct2    '(norm)\b' $ make_op
        token real      '(\d+\.\d*|\d*\.\d+)([eE][-+]?\d+)?|\d+[eE][-+]?\d+'
                 $ float
        token integer   '\d+' $ int
        token VarId     '[a-zA-Z_]\w*'
         ;
```

```
START/e ->
        'vars'                    $ e=self.mem()
    |   VarId/v '=' Expr/e        $ self[v]=e
    |   Expr/e
;
Var/$self.get(v,0)$ -> VarId/v ;
Expr/e -> Term/e ( add_op/op Term/t     $ e=op(e,t)
                 )*
;
Term/t -> Fact/t ( mul_op/op Fact/f     $ t=op(t,f)
                 )*
;
Fact/f ->
        add_op/op Fact/f                $ f=op(0,f)
    |   Pow/f
;
Pow/f -> Atom/f ( pow_op/op Fact/e      $ f=op(f,e)
                )?
;
```

```
    Atom/a ->
            real/a
        |   integer/a
        |   Function/a
        |   Var/a
        |   '\(' Expr/a '\)'
    ;
    Function/y ->
            funct1/f '\(' Expr/x '\)'                  $ y = f(x)
        |   funct2/f '\(' Expr/x1 ',' Expr/x2 '\)'  $ y = f(x1,x2)
    ;
"""
def mem(self):
    vars = sorted(self.items())
    memory = [ "%s = %s"%(var, val) for (var, val) in vars ]
    return "\n\t" + "\n\t".join(memory)
```

```
print("Calc (TPG example)")
calc = Calc()
while 1:
    l = raw_input("\n:")
    if l:
        try:
            print(calc(l))
        except Exception:
            print(tpg.exc())
    else:
        break
```

AntLR

ANother **T**ool for **L**anguage **R**ecognition is an LL(k) parser and translator generator tool

which can create

- lexers
- parsers
- abstract syntax trees (AST's)

in which you describe the language grammatically

and in return receive a program that can recognize and translate that language

Tasks Divided

- Lexical Analysis (scanning)
- Semantic Analysis (parsing)
- Tree Generation
  - Abstract Syntax Tree (AST) is a structure which keeps information in an easily traversable form (such as operator at a node, operands at children of the node)
  - ignores form-dependent superficial details
- Code Generation

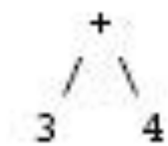
The Java Code

- The code to invoke the parser:

```
import java.io.*;
class Main {
  public static void main(String[] args) {
    try {
      // use DataInputStream to grab bytes
      MyLexer lexer = new MyLexer(
            new DataInputStream(System.in));
      MyParser parser = new MyParser(lexer);
      int x = parser.expr();
      System.out.println(x);
    } catch(Exception e) {
      System.err.println("exception: "+e);
    }
  }
}
```

Abstract Syntax Trees

- Abstract Syntax Tree: Like a parse tree, without unnecessary information
- Two-dimensional trees that can encode the structure of the input as well as the input symbols
- An AST for (3+4) might be represented as



  - No parentheses are included in the tree!