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 digits (a number that doesn't start with 0) =

the regular expression "non_zero_digit digit*"

- **Specifying the syntax for programming languages: Regular Expressions 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 (concatenation of zero or more strings)
    - E.g., name -> letter (letter | digit)*

Regular Expressions

- RE example: the syntax of numeric constants:

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
A digit "is"	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Regular Expressions

- **Regular expressions work well for defining tokens.**
  - **They are unable to specify nested constructs.**
    - **For example, a grammar 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 create tokens, atoms of the syntax tree.
- Chomsky Hierarchy:
  - Type-0: Unrestricted Language - Turing Machine
  - Type-1: Context-Sensitive Language - Turing Machine w/ Tape * Input
  - Type-2: Context-Free Language - Pushdown Automata - Scanner
  - Type-3: Regular Language - Finite Automata/Regex – Lexer
    - 0 and 1 usually too slow for practical use. 2 maybe, maybe not. 3 are fast.

Context-Free Grammars (CFG)

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

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

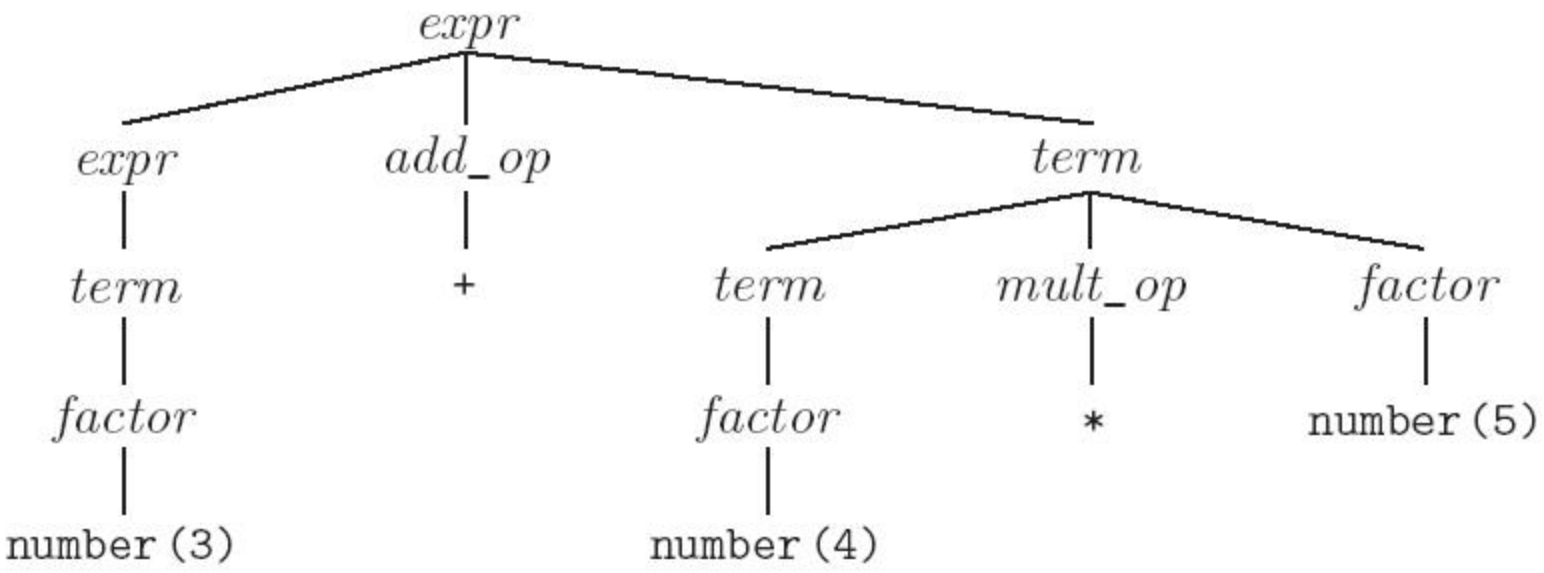
  op → + | - | * | /

  - Each of the rules in a CFG is known as a *production*.
  - The symbols on the left-hand sides of the productions are *nonterminals*.
- A CFG consists of:
  - A set of terminals T,
  - A set of non-terminals N,
  - A start symbol S (a non-terminal), and
  - A set of productions.

Derivations and ParseTrees

- A context-free grammar shows us how to generate a syntactically valid string of terminals.
- Parse tree for expression grammar (with precedence) for

3 + 4 * 5



Context free grammars

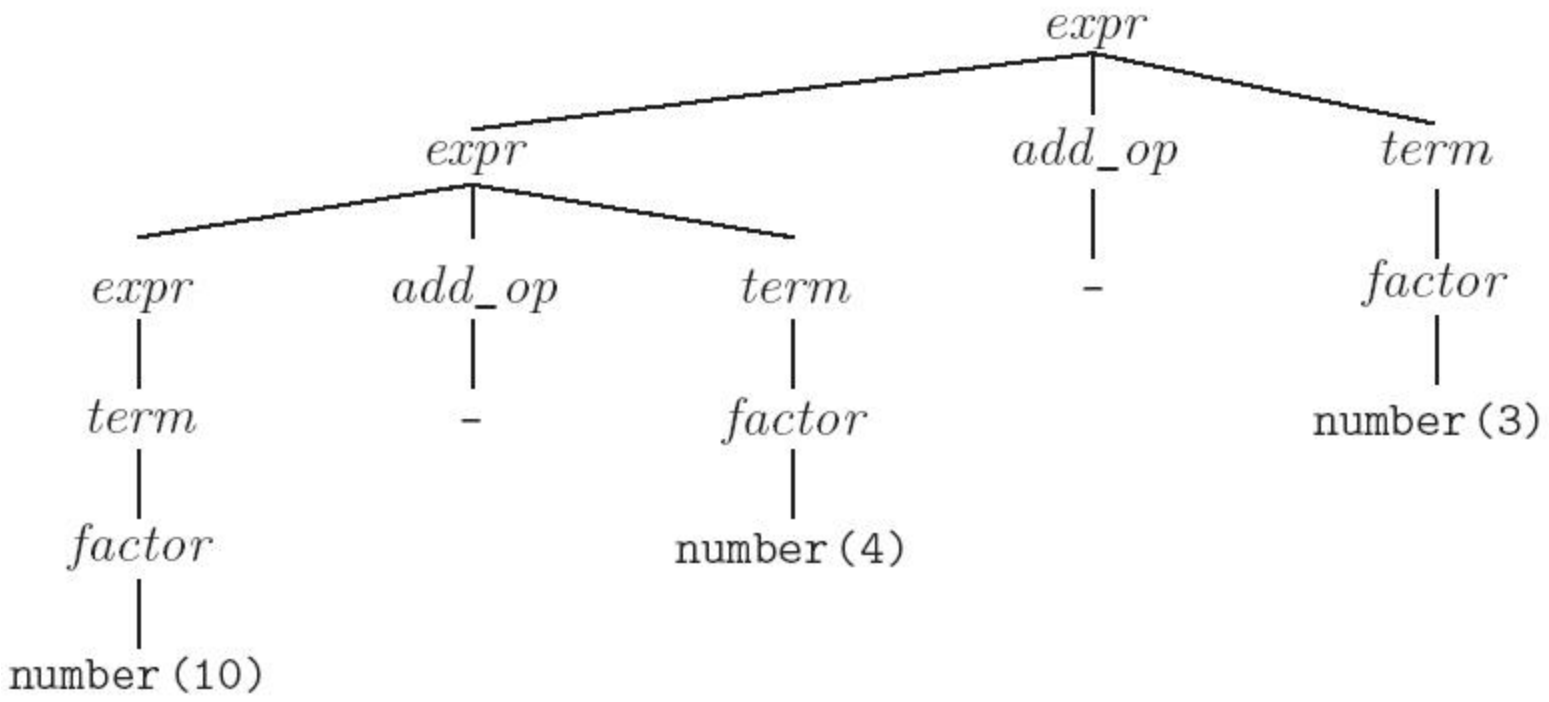
- The previous grammar was ambiguous (can generate multiple parse trees): one corresponds to 3+(4*5) and one corresponds to (3+4)*5
- A better version of our expression grammar should include precedence and associativity:

1. *expr* ⟶ *term* | *expr add_op term*
2. *term* ⟶ *factor* | *term mult_op factor*
3. *factor* ⟶ id | number | - *factor* | (*expr*)
4. *add_op* ⟶ + | -
5. *mult_op* ⟶ * | /

see example parse tree on the next page

Context-Free Grammars

- Parse tree for expression grammar (with left associativity) for 10 - 4 - 3



Scanning

- The scanner and parser for a programming language are responsible for discovering the syntactic structure of a program (the *syntax analysis*).
- The scanner 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

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

assign → :=

plus → +

minus → -

times → *

div → /

lparen → (

rparen →)

id → *letter* (*letter* | *digit*)* except for *read* and *write*

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

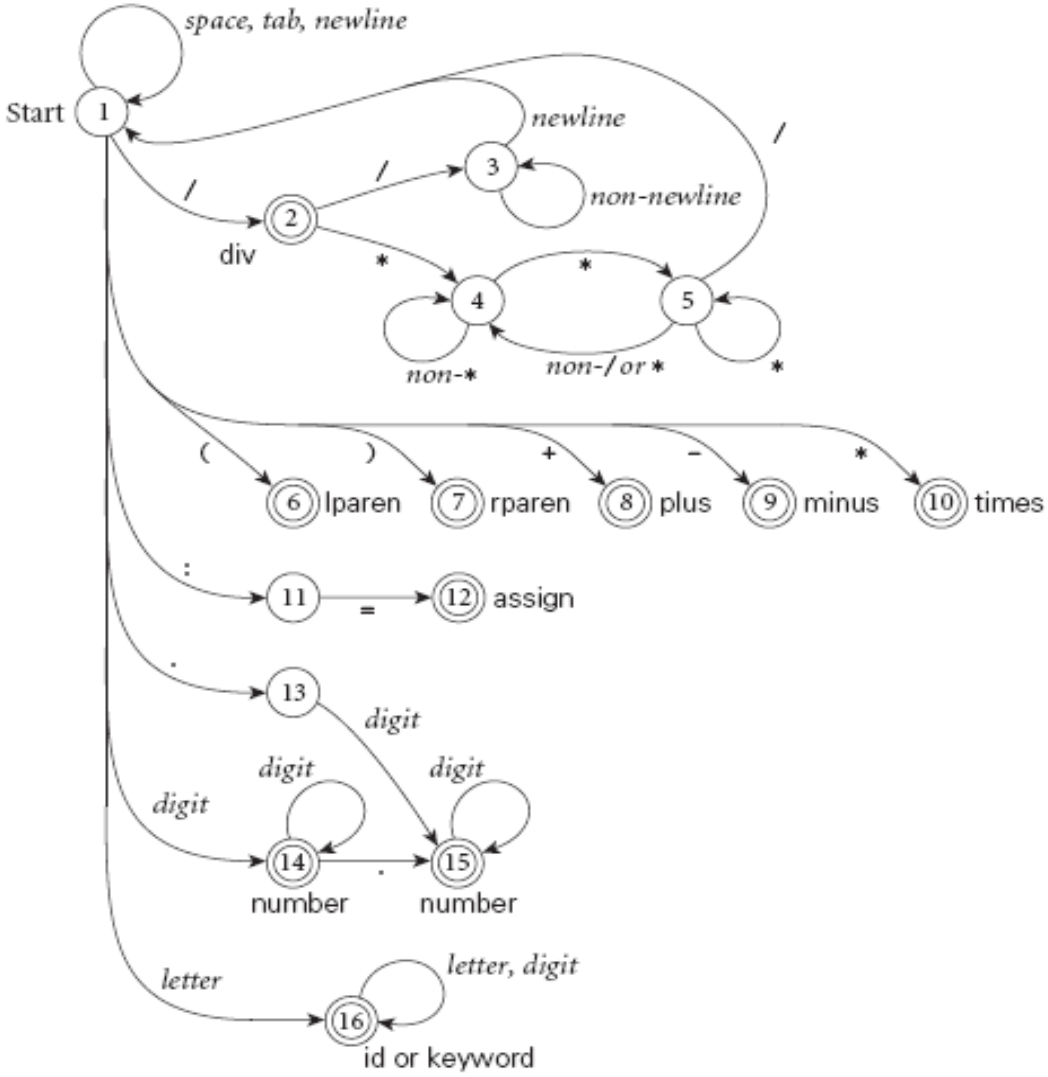
skip any initial white space (spaces, tabs, and newlines)
if cur_char ∈ {'(', ')', '+', '-', '*'}
    return the corresponding single-character token
if cur_char = ':'
    read the next character
    if it is '=' then return *assign* else announce an error
if cur_char = '/'
    peek at the next character
    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

- This is a deterministic finite automaton (DFA)
  - lex, scangen, etc. build these things automatically from a set of regular expressions
  - Specifically, they construct a machine that accepts the language identifier | int const | real const | comment | symbol | …

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 const and never 3, ., and 14159
- Regular expressions "generate" a regular language;
- DFAs "recognize" it

Scanning

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

Scanning

- Writing a pure DFA as a set of nested case statements is a surprisingly useful programming technique
  - though it's often easier to 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

Scanning

- Note that 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")

Perl-style Regexp

- Learning by examples:

  abcd - concatenation

  a(b|c)d - grouping

  a(b|c)*d - can apply # 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?

  \d+\.\d*|\.\d+

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

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 (*syntax-directed translation*).
- Most use a context-free grammar (CFG)
- Terminology:
  - symbols
    - terminals (tokens)
    - non-terminals
  - production
  - derivations (left-most and right-most - canonical)
  - parse trees
  - sentential form

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 1000 tokens.

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'.
- LR stands for 'Left-to-right, Rightmost derivation‘.
- Leftmost derivation - work on the left side of the parse tree.
- Rightmost derivation - work on the right side of the tree.

Parsing

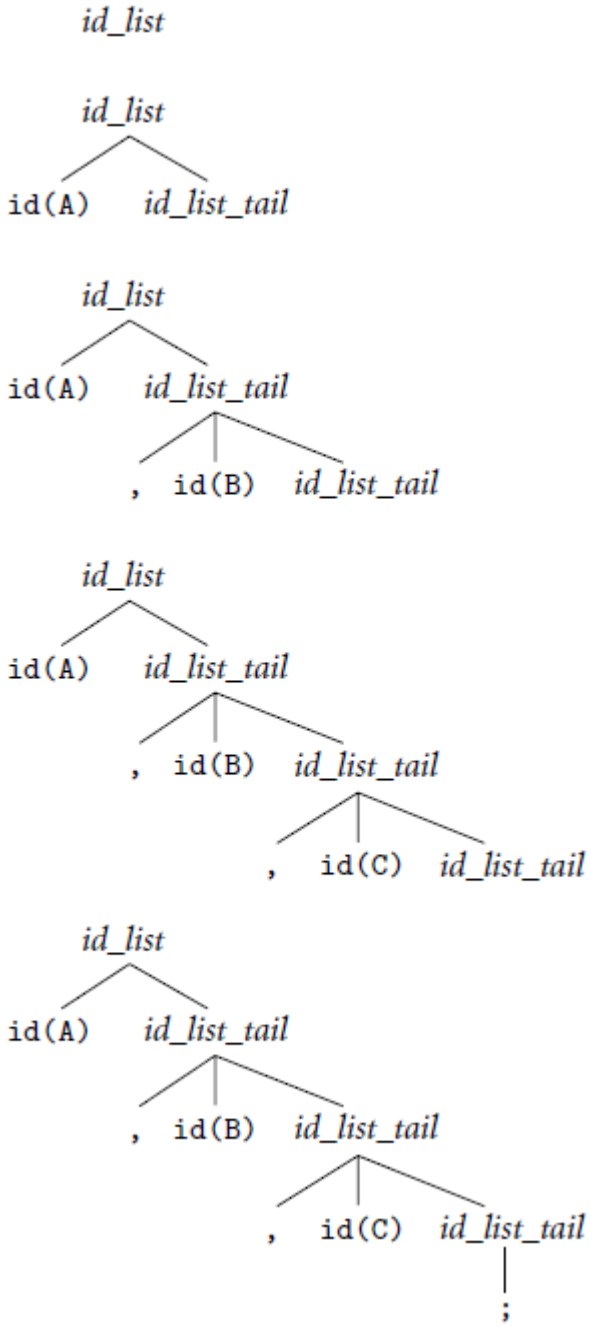
- LL parsers are also called 'top-down', or 'predictive' parsers & LR parsers are also called 'bottom-up', or 'shift-reduce' parsers
- There are several important sub-classes of LR parsers
  - SLR
  - LALR
  - “full LR”

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 → ;

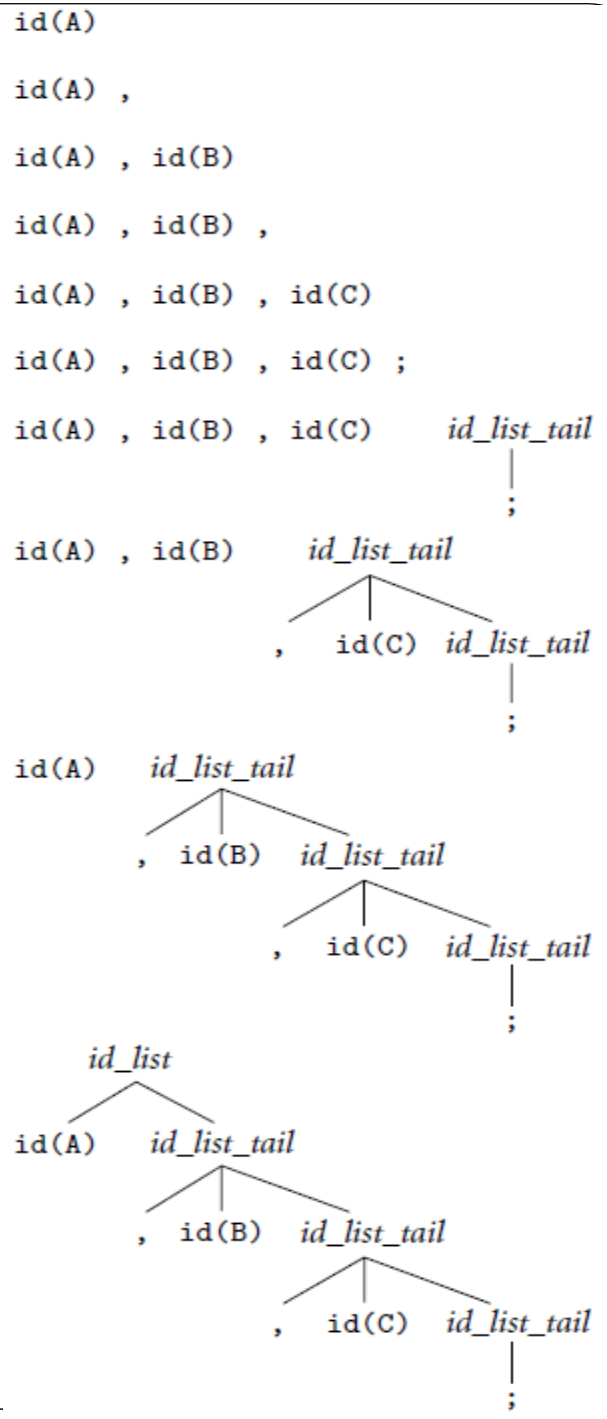
- The top-down construction of a parse tree for the string: "A, B, C;" starts from the root and applies rules and tried to identify nodes.



Bottom-up parsing (LR)

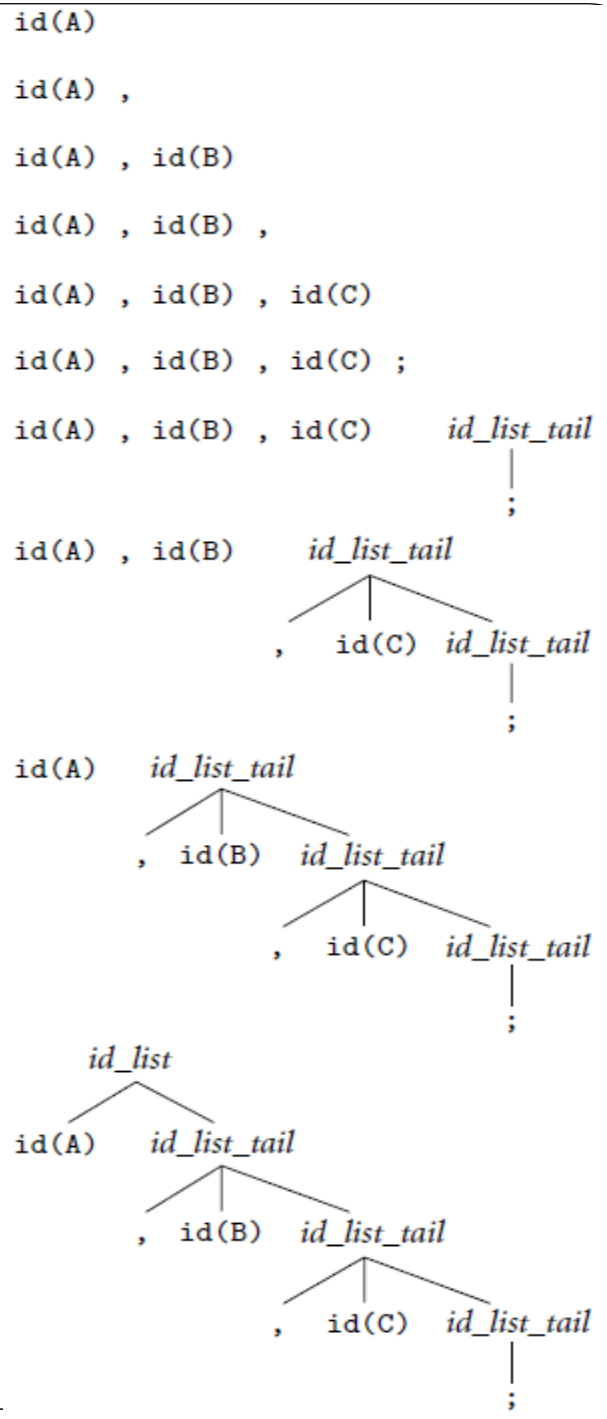
id_list → id *id_list_tail*
id_list_tail → , id *id_list_tail*
id_list_tail → ;

- The bottom-up construction of a parse tree for the 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
- The expression grammar (with precedence and associativity) you saw before is LR(1), but not LL(1)
- Every LL(1) grammar is also LR(1), though right recursion in production tends to require very deep stacks and complicates semantic analysis
- Every CFL that can be parsed deterministically has an SLR(1) grammar (which is LR(1))
- Every deterministic CFL with the prefix property (no valid string is a prefix of another valid string) has an LR(0) grammar.

LL Parsing

- Consider 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

LL Parsing

term → factor fact_tailt

fact_tail → mult_op fact fact_tail
| ε

factor → (expr)
| id
| number

add_op → +
| -

mult_op → *
| /

LL Parsing

- Like the bottom-up grammar, this one 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
- How do we parse a string with this grammar?
  - by building the parse tree incrementally

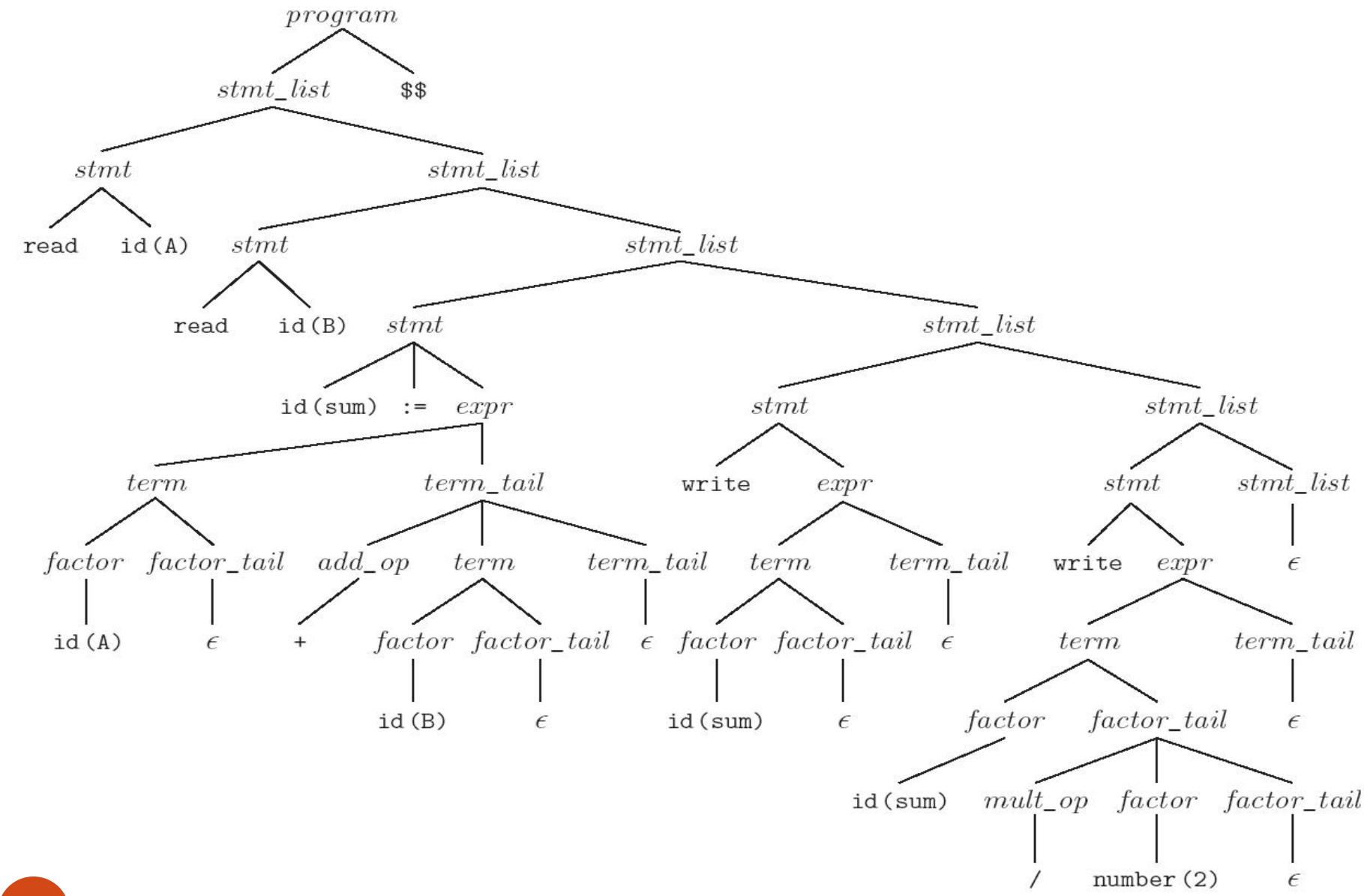
LL Parsing

- Example (the average program):

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

- We 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

Parse tree for the average program



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
  - (3) announce a syntax error

LL Parsing

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

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

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

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
| 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

- Note that eliminating left recursion and common prefixes does NOT make a grammar LL
  - there are infinitely many non-LL LANGUAGES, and the mechanical transformations work on them just fine

LL Parsing

- 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 grammar fragment is ambiguous (Pascal):

stmt → if cond then_clause else_clause | other_stuff

then_clause → then stmt

else_clause → else stmt

| ε

String: “if C1 then if C2 then S1 else S2”

The else can be paired with either then!!!

Desired effect: pair the else with the nearest then.

LL Parsing

- The less natural grammar fragment can be parsed bottom-up but not top-down:

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 thens and elses.
  - An *unbalanced stmt* has more thens.

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)

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;…
```

LL Parsing

- The algorithm to build predict sets is tedious (for a "real" sized grammar), but relatively simple
- It consists of three stages:

  (1) compute FIRST sets for symbols

  (2) compute FOLLOW sets for non-terminals
  (this requires computing FIRST sets for some strings)

  (3) compute predict sets or table for all productions

LL Parsing

- It is conventional in general discussions of grammars to use
  - lower case letters near the beginning of the alphabet for terminals
  - lower case letters near the end of the alphabet for strings of terminals
  - upper case letters near the beginning of the alphabet for non-terminals
  - upper case letters near the end of the alphabet for arbitrary symbols
  - greek letters for arbitrary strings of symbols

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
  - the stack contains a record of what has been seen SO FAR (NOT what is expected)

LR Parsing

- A scanner is a DFA
  - it can be specified with a state diagram
- An LL or LR parser is a PDA
  - Early's & CYK algorithms do NOT use PDAs
  - 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

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.

Classic Parsing Tools

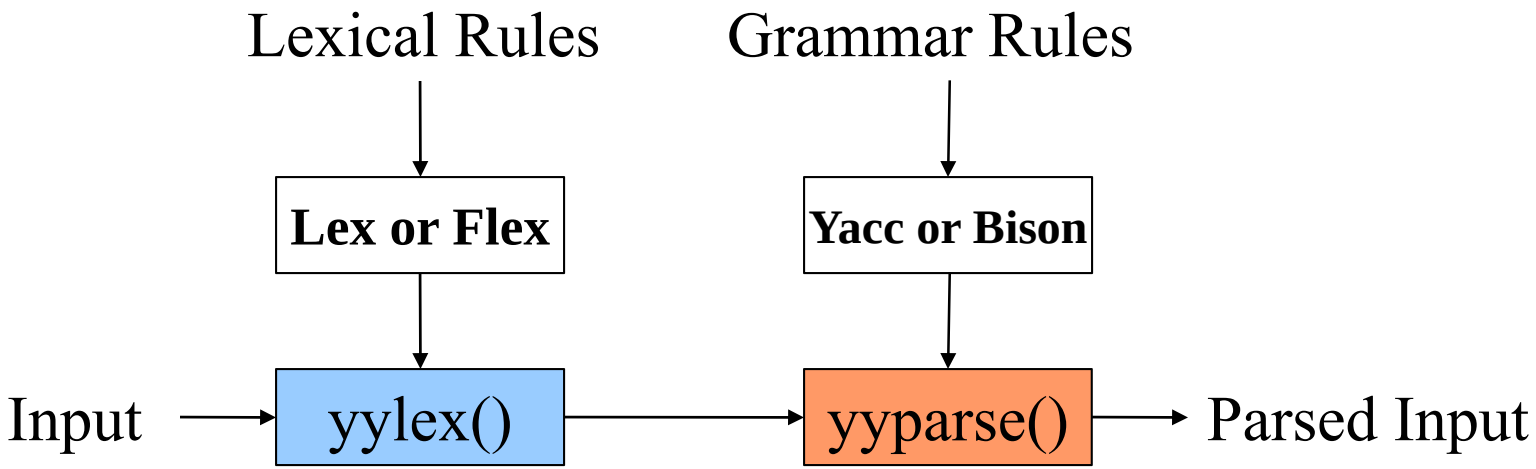
lex - original UNIX lexics generator (Lesk, 1975)

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

yacc - "*yet another compiler compiler*" UNIX parser (Johnson, 1975)

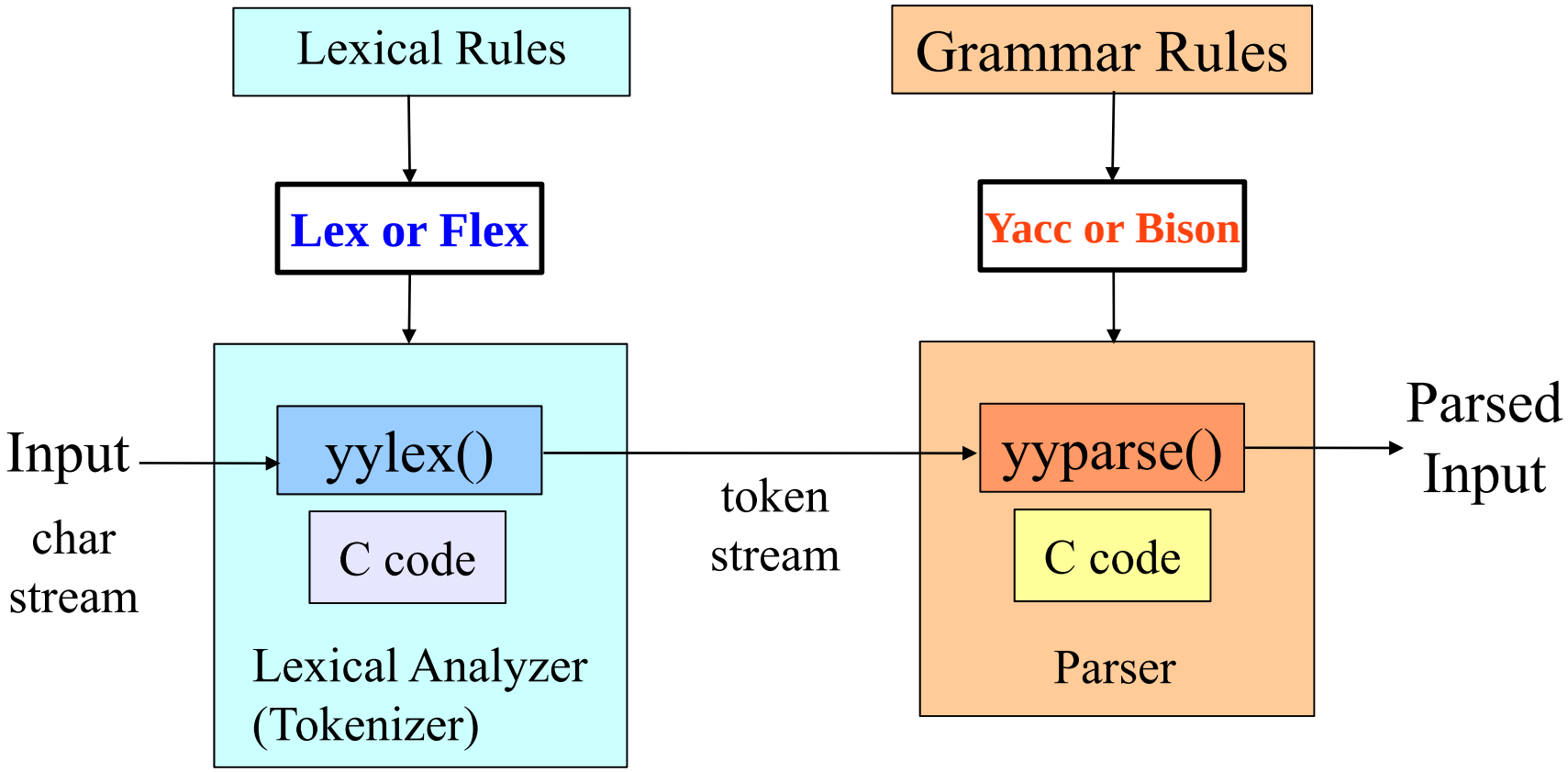
- generate a C program for a parser from BNF rules

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

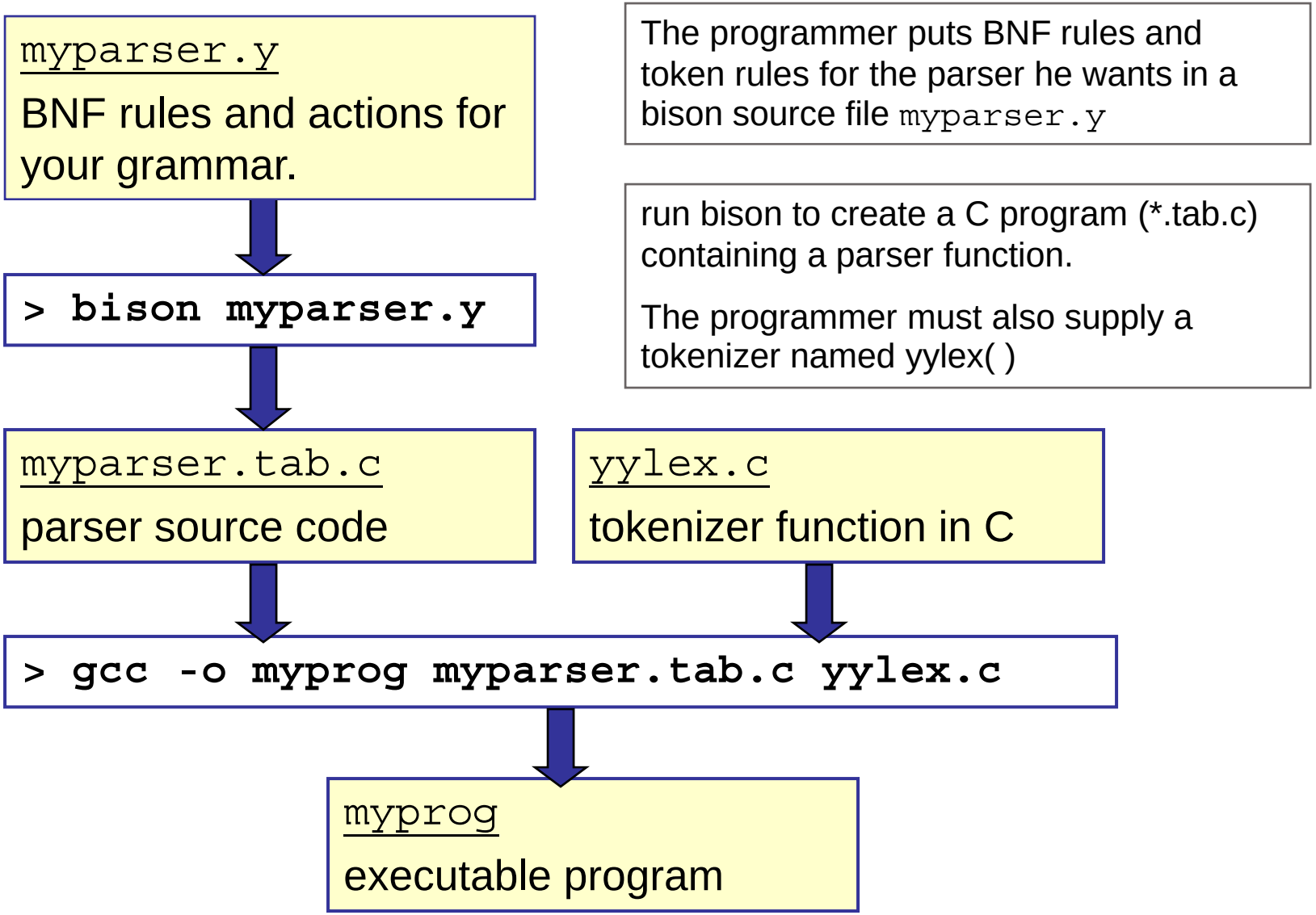


Classic Parsing Tools

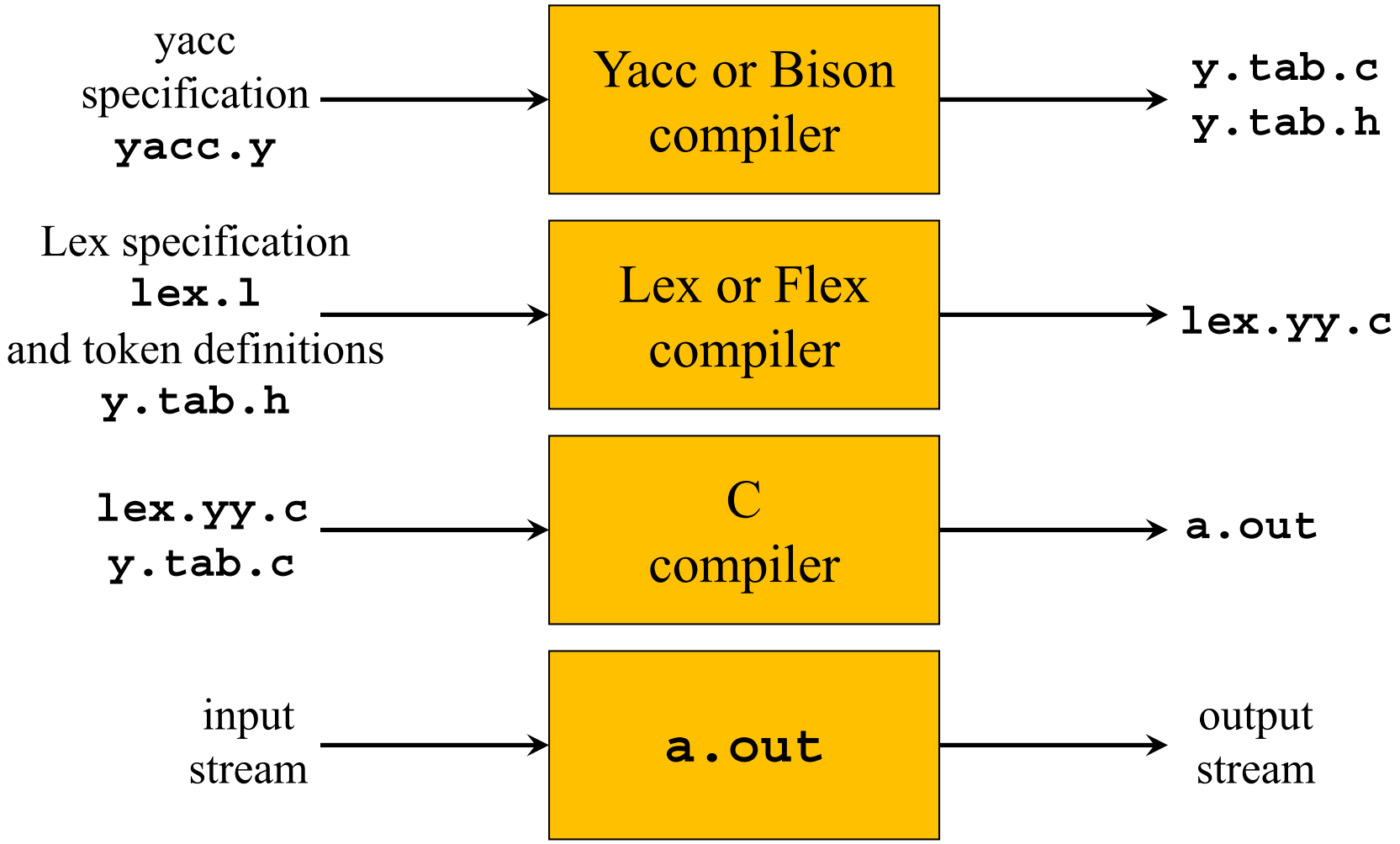
- Lex and Yacc generate C code for your analyzer & parser.



Bison Overview



Combining Lex/Flex with Yacc/Bison



Lex Specification Example


```
%option noyywrap
%{
#include "y.tab.h"

extern double yylval;
%}
number  [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ ]             { /* skip blanks */ }
{number}        { sscanf(yytext, "%lf", &yylval);
                  return NUMBER;
                }
\n|.            { return yytext[0]; }
```

Generated by Yacc, contains **`#define NUMBER …`**

Defined in **`y.tab.c`**

Bison Specification Example

- A grammar written in BNF.
  - Bison creates a C program that parses according to the rules

```
term    : term '*' factor { $$ = $1 * $3; }
        | term '/' factor { $$ = $1 / $3; }
        | factor          { $$ = $1; }
        ;
factor  : ID              { $$ = valueof($1); }
        | NUMBER          { $$ = $1; }
        ;
```

```
yacc -d example2.y
lex example2.l
gcc y.tab.c lex.yy.c
./a.out
```

```
bison -d -y example2.y
flex example2.l
gcc y.tab.c lex.yy.c
./a.out
```

TPG example

- 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.

TPG example

- Defining the grammar:
  - Terminal symbols for expressions:

number	[0 - 9]+ or \d+	One or more digits
add	[+-]	a + or a -
mul	[*/]	a * or a /
spaces	\s+	One or more spaces

  - 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 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 ;                  # continue on next slide
```

```
        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
```