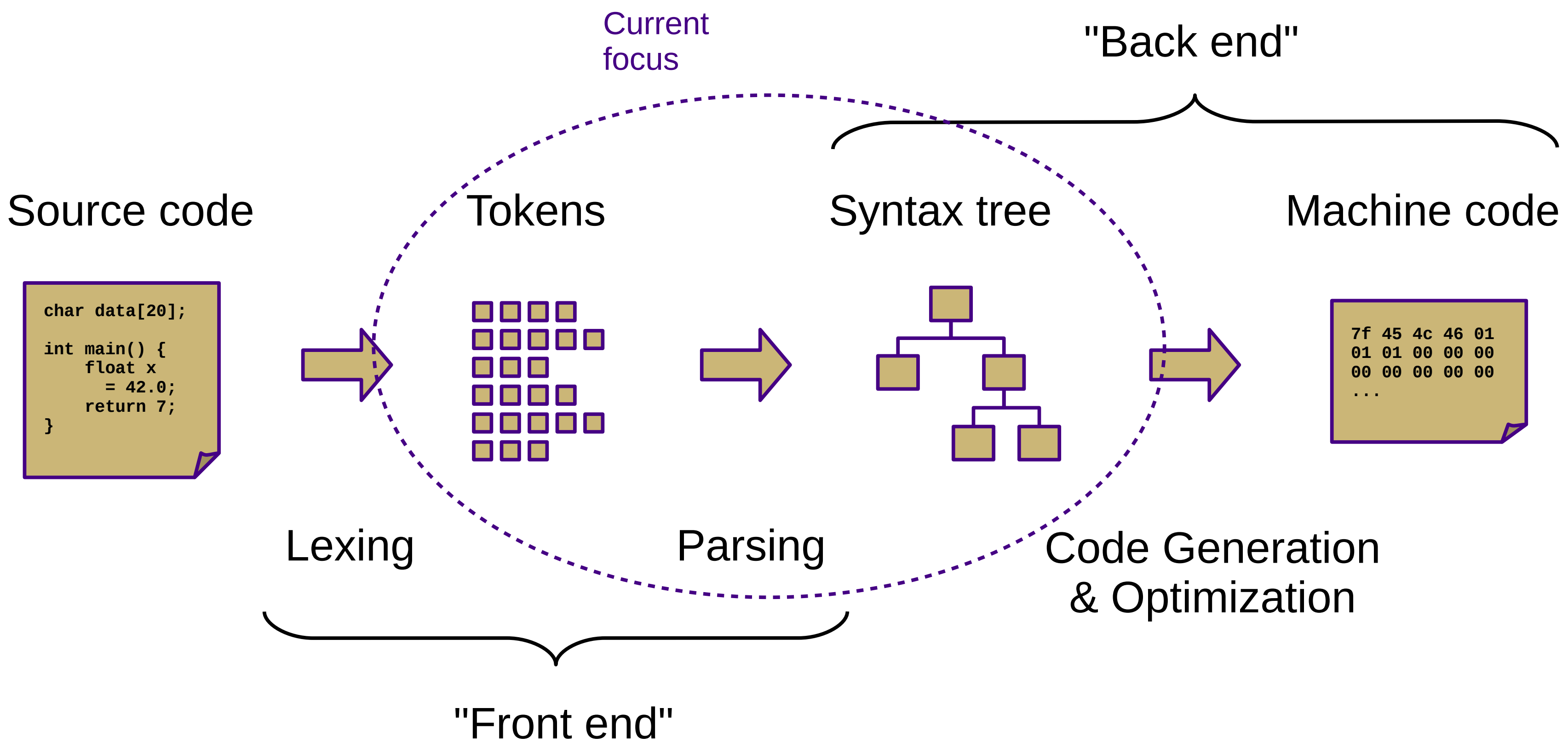
CS 432
Fall 2025

Mike Lam, Professor

Top-Down (LL) Parsing

Compilation



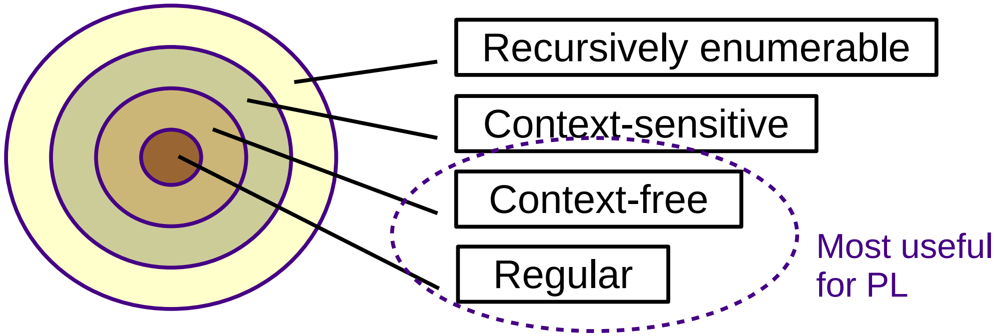
Review

- Recognize regular languages with finite automata
  - Described by regular expressions
  - Rule-based transitions, no memory required
- Recognize context-free languages with pushdown automata
  - Described by context-free grammars
  - Rule-based transitions, MEMORY REQUIRED
    - Add a stack!

Segue

KEY OBSERVATION: Allowing the translator to use memory to track parse state information enables a wider range of automated machine translation.

Chomsky Hierarchy of Languages

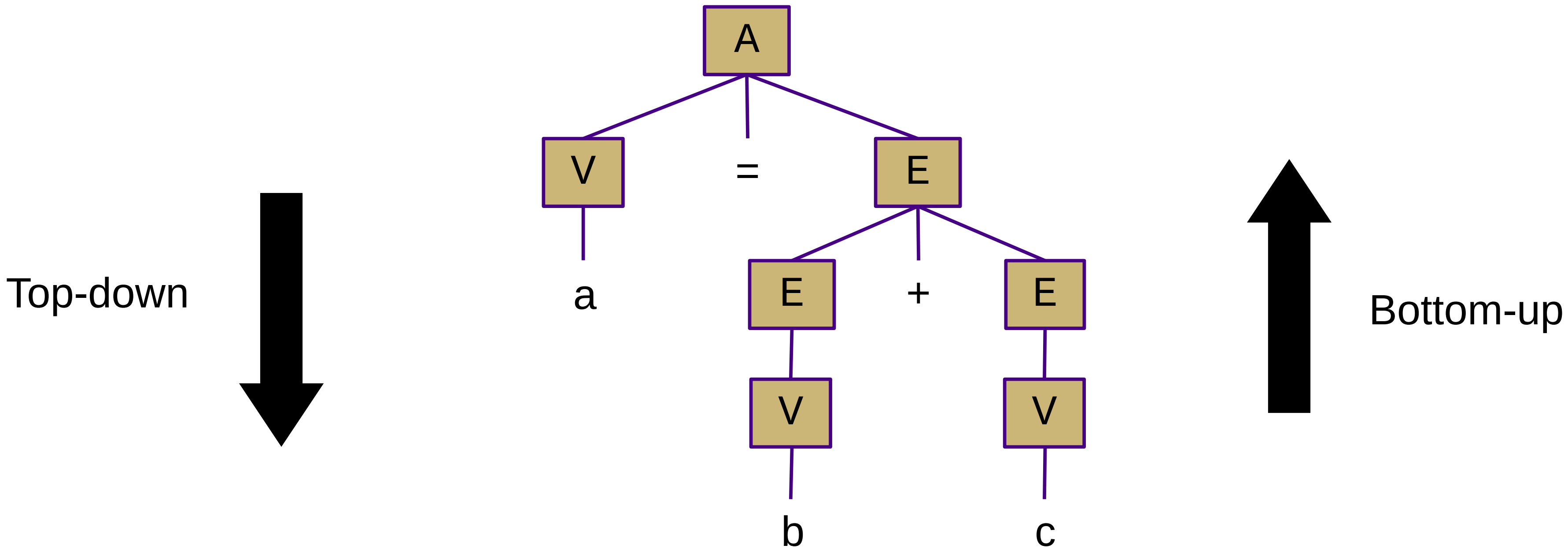


Grammar	Languages	Automaton	Production rules (constraints)
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$ (no restrictions)
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow a$ and $A \rightarrow aB$

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

Parsing Approaches

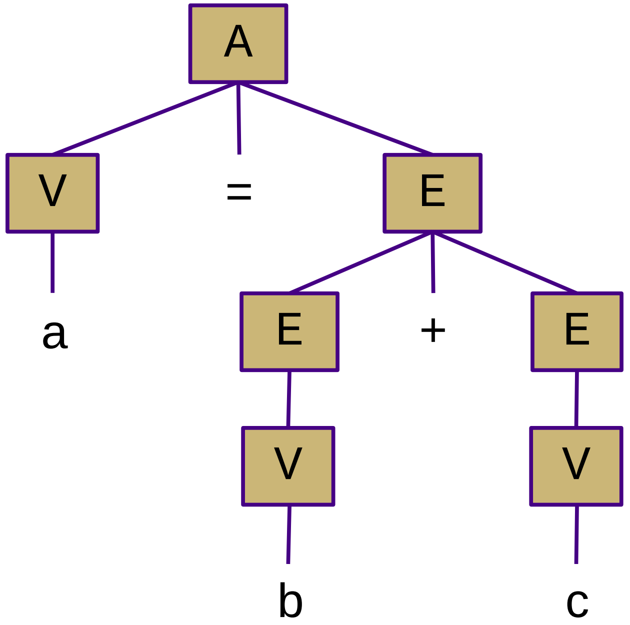
- Top-down: begin with start symbol (root of parse tree), and gradually expand non-terminals
  - Stack contains non-terminals that are still being expanded
- Bottom-up: begin with terminals (leaves of parse tree), and gradually connect using non-terminals
  - Stack contains roots of subtrees that still need to be connected



Top-Down Parsing

```
root = createNode(S)
focus = root
push(null)
token = nextToken()

loop:
    if (focus is non-terminal):
        B = chooseRuleAndExpand(focus)
        for each b in B.reverse():
            focus.addChild(createNode(b))
            push(b)
        focus = pop()

    else if (token == focus):
        token = nextToken()
        focus = pop()

    else if (token == EOF and focus == null):
        return root

    else:
        exit(ERROR)
```

A → V = E
V → a | b | c
E → E + E
  | V



Recursive Descent Parsing

- Idea: use the system stack rather than an explicit stack
  - One parsing function and node for each non-terminal
  - Encode productions with function calls and token checks
  - Use stack/recursion to track current “state” of the parse
  - Easiest kind of parser to write manually

A → ‘if’ C ‘then’ S
  | ‘goto’ L

```
class A {
    enum Type
        { IFTHEN, GOTO }
    Type type
    C cond
    S stmt
    L lbl
}
```

➡

```
parseA(tokens):
    node = new A()
    next = tokens.next()
    if next == ”if”:
        node.type = IFTHEN
        node.cond = parseC()
        matchToken(“then”)
        node.stmt = parseS()
    else if next == “goto”
        node.type = GOTO
        node.lbl = parseL()
    else
        error (“expected ‘if’ or ‘goto’”)
    return node
```

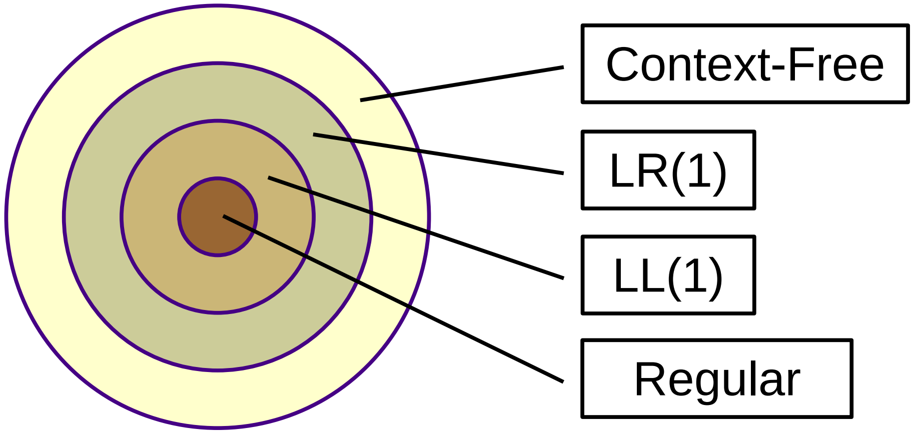
Top-Down Parsing

- Main issue: choosing which rule to use
  - In previous example, we just looked for ‘if’ or ‘goto’
  - With full lookahead, it would be relatively easy
    - This would be very inefficient
  - Can we do it with a single lookahead?
    - That would be much faster
    - Must be careful to avoid backtracking

LL(1) Parsing

- LL(1) grammars and parsers
  - **Left-to-right** scan of the input string
  - **Leftmost** derivation
  - **1 symbol** of lookahead
  - Highly restricted form of context-free grammar
    - No left recursion
    - No backtracking

Context-Free Hierarchy



LL(1) Grammars

- We can convert many grammars to be LL(1)
  - Must remove left recursion
  - Must remove common prefixes (i.e., left factoring)
  - Easy (relatively) to hand-write a parser
    - **Practical** solution to real-world translation problems

$$A \rightarrow A\ \alpha \;|\; \beta$$

Grammar with left recursion

$$A \rightarrow \alpha\ \beta_1 \;|\; \alpha\ \beta_2$$

Grammar with common prefixes

Left Factoring

- Common prefix: $A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2 \ldots$
  - Leads to ambiguous rule choice in a top-down parser
    - One lookahead (α) is not enough to pick a rule; backtracking is required
  - To fix, left factor the choices into a new non-terminal
  - **Practical note (P2)**: A and A' can be a single function in your code
    - Parse and save data about α in temporary variables until you have enough information to choose

$$
\begin{aligned}
A &\rightarrow \alpha\ \beta_1 \\
&\mid\ \alpha\ \beta_2 \\
&\quad \ldots
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
A &\rightarrow \alpha\ A' \\
A' &\rightarrow \beta_1 \\
&\mid\ \beta_2 \\
&\quad \ldots
\end{aligned}
$$

Eliminating Left Recursion

- Left recursion: $A \rightarrow A\ \alpha \mid \beta$
  - Often a result of left associativity (e.g., expression grammar)
  - Leads to infinite looping/recursion in a top-down parser
  - To fix, unroll the recursion into a new non-terminal
  - **Practical note (P2)**: A and A’ can be a single function in your code
    - Parse one β, then continue parsing α’s until there are no more
    - Keep adding the previous parse tree as a left subnode of the new parse tree

$$
\begin{aligned}
A &\rightarrow A\ \alpha \\
  &\mid\ \beta
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
A &\rightarrow \beta\ A' \\
A' &\rightarrow \alpha\ A' \\
   &\mid\ \varepsilon
\end{aligned}
$$

Examples

- Eliminating left recursion:

```
E → E + T
  | E - T
  | T
```

➡

```
E  → T E'

E' → + T E′
   | - T E′
   | ε
```

- Left factoring:

```
C → if E B else B fi
  | if E B fi
```

➡

```
C  → if E B C′

C' → else B fi
   | fi
```

LL(1) Parsing

- LL(1) parsers can also be auto-generated
  - Similar to auto-generated lexers
  - Tables created by a *parser generator* using FIRST and FOLLOW helper sets
  - These sets are also useful when building hand-written recursive descent parsers
    - And (as we'll see next week), when building SLR parsers

LL(1) Parsing

- FIRST(α)
  - Set of terminals (or ε) that can appear at the start of a sentence derived from α (a terminal or non-terminal)
- FOLLOW(A) set
  - Set of terminals (or $) that can occur immediately after non-terminal A in a sentential form
- $FIRST^{+}(A \rightarrow \beta)$
  - If ε is not in FIRST(β)
    - $FIRST^{+}(A) = FIRST(\beta)$
  - Otherwise
    - $FIRST^{+}(A) = FIRST(\beta) \cup FOLLOW(A)$

Useful for choosing which rule to apply when expanding a non-terminal

Calculating FIRST(α)

- Rule 1: α is a terminal **a**
  - FIRST(a) = { **a** }

- Rule 2: α is a non-terminal X
  - Examine all productions $X \rightarrow Y_1\ Y_2\ \ldots\ Y_k$
    - add $FIRST(Y_1)$ if not $Y_1 \rightarrow^* \varepsilon$
    - add $FIRST(Y_i)$ if $Y_1 \ldots Y_j \rightarrow^* \varepsilon$, where j = i-1 (i.e., skip disappearing symbols)
  - FIRST(X) is union of all of the above

- Rule 3: α is a non-terminal X and $X \rightarrow \varepsilon$
  - FIRST(X) includes ε

Calculating FOLLOW(B)

- Rule 1: FOLLOW(S) includes **EOF** / **$**
  - Where S is the start symbol

- Rule 2: for every production $A \rightarrow \alpha\ B\ \beta$
  - FOLLOW(B) includes everything in FIRST(β) except ε

- Rule 3: if $A \rightarrow \alpha\ B$ or ($A \rightarrow \alpha\ B\ \beta$ and FIRST(β) contains ε)
  - FOLLOW(B) includes everything in FOLLOW(A)

Example

- FIRST and FOLLOW sets:

```
A → x A x
  | y B y
B → C m
  | C
C → t
```

FIRST(x) = { x }
FIRST(y) = { y }

FIRST(A) = { x , y }
FIRST(B) = { t }
FIRST(C) = { t }

$FIRST^+$(A → x A x) = { x }
$FIRST^+$(A → y B y) = { y }
(disjoint: this is ok)

$FIRST^+$(B → C m) = { t }
$FIRST^+$(B → C) = { t }
(not disjoint: requires backtracking!)

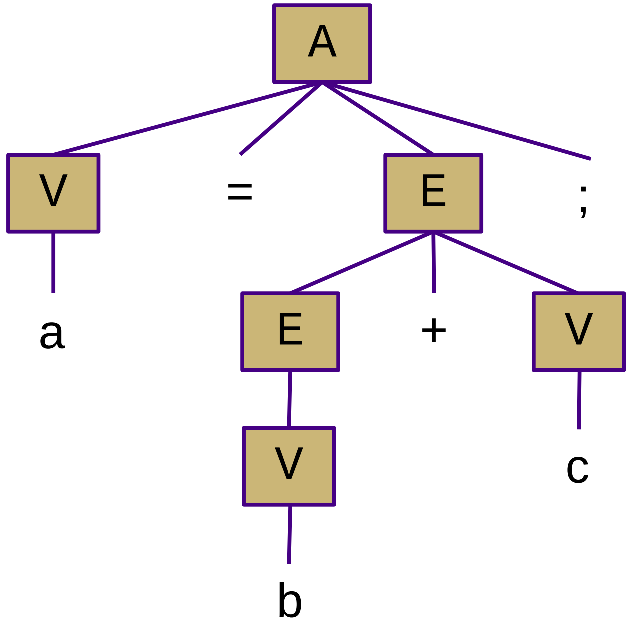
FOLLOW(A) = { x, $ }
FOLLOW(B) = { y }
FOLLOW(C) = { m, y }

Aside: abstract syntax trees

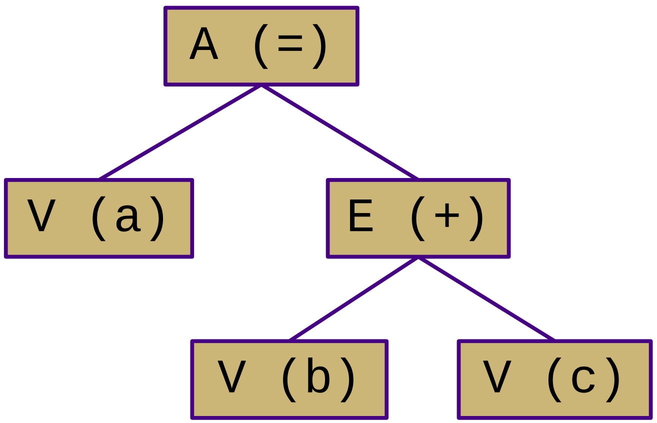
Grammar:

A → V = E ;
E → E + V
  | V
V → a | b | c

Parse tree:



Abstract syntax tree:



In P2, you will build an AST, not a parse tree!

Example expression parser

- Available on stu:
  `/cs/students/cs432/f25/expr_parser.tar.gz`
- Grammar (converted to LL):

```
E  →  T E'
E' →  + T E'
    |  e
T  →  F T'
T' →  * F T'
    |  e
F  →  ( E )
    |  {DEC}
```

Example parsing routine

```
ParseNode* parse_term(TokenQueue* input)
{
    /*
     * T → F T'
     */
    ParseNode* root = parse_factor(input);

    /*
     * T' → * F T'
     *    | ε
     */
    while (is_next_token(input, "*")) {
        ParseNode* new_root = ParseNode_new("*");
        new_root->left = root;
        match_and_remove_token(input, "*");
        new_root->right = parse_factor(input);
        root = new_root;
    }

    return root;
}
```

Aside: Parser combinators

- A parser combinator is a higher-order function for parsing
  - Takes several parsers as inputs, returns new parser as output
  - Allows parser code to be very close to grammar
  - (Relatively) recent development: ‘90s and ‘00s
  - Example: Parsec in Haskell

```
whileStmt :: Parser Stmt
whileStmt =
  do keyword "while"
     cond <- expression
     keyword "do"
     stmt <- statement
     return (While cond stmt)
```

```
assignStmt :: Parser Stmt
assignStmt =
  do var  <- identifier
     operator ":="
     expr <- expression
     return (Assign var expr)
```