Programming Language Syntax: Scanning and Parsing

Read: Scott, Chapter 2.2 and 2.3.1

Lecture Outline

- Overview of scanning
- Overview of top-down and bottom-up parsing

- Top-down parsing
  - Recursive descent
  - LL(1) parsing tables

Scanning

- Scanner groups characters into tokens
- Scanner simplifies the job of the parser

position = initial + rate * 60;

↓

Scanner

↓

`id = id + id * num ;`

↓

Parser

↓

- Scanner is essentially a Finite Automaton
  - Regular expressions specify the syntax of tokens
  - Scanner recognizes the tokens in the program

Question

- Why most programming languages disallow nested multi-line comments?

  - Comments are usually handled by the scanner, which essentially is a DFA. Handling multiline comments would require recognizing $(/*)^n(*/)^n$ which is beyond the power of a DFA.

Calculator Language

- Tokens

$\textit{times} \rightarrow \mathbf{*}$

$\textit{plus} \rightarrow \mathbf{+}$

$\textit{id} \rightarrow \textit{letter} \; (\; \textit{letter} \mid \textit{digit} \;)^{*}$

except for **read** and **write** which are keywords (keywords are tokens as well)

Ad-hoc (By hand) Scanner

skip any initial white space (space, tab, newline)

if current_char in { `+`, `*` }

  return corresponding single-character token (*plus* or *times*)

if current_char is a letter

  read any additional letters and digits

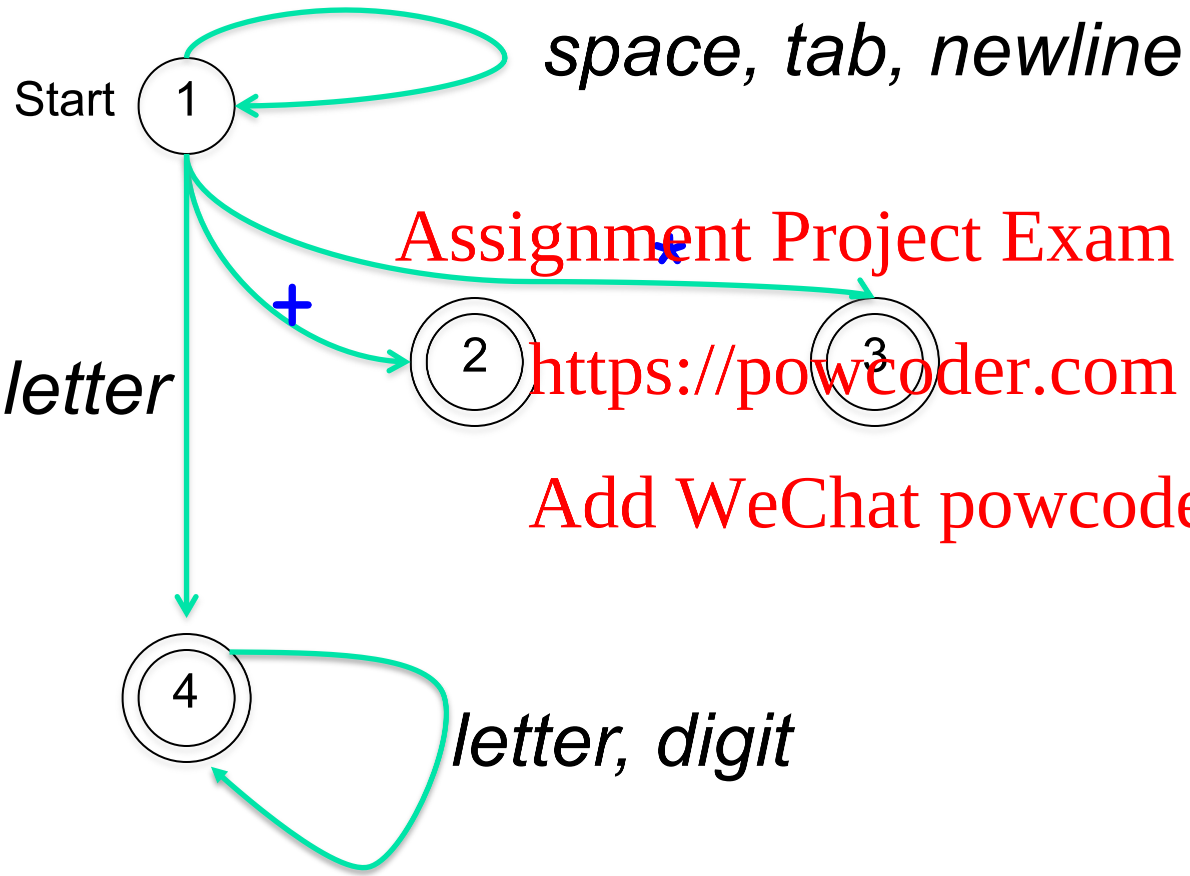
  check to see if the resulting string is **read** or **write**

  if so, then return the corresponding token

  else return `id`

else announce an ERROR

The Scanner as a DFA



Building a Scanner

- Scanners are (usually) automatically generated from regular expressions:

  Step 1: From a Regular Expression to an NFA

  Step 2: From an NFA to a DFA

  Step 3: Minimizing the DFA

- **lex/flex** utilities generate scanner code
- Scanner code explicitly captures the states and transitions of the DFA

Table-Driven Scanning

```
…
cur_state := 1
loop
    read cur_char
    case scan_tab[cur_char, cur_state].action of
        move:
            …
            cur_state = scan_tab[cur_char, cur_state].new_state
        recognize: // emits the token
            tok = token_tab[current_state]
            unread cur_char --- push back char
            exit loop
        error:
```

Table-Driven Scanning

	space,tab,newline	*	+	*digit*	*letter*	other	
1	5	2	3	-	4	-	
2	-	-	-	-	-	-	*times*
3	-	-	-	-	-	-	*plus*
4	-	-	-	4	4	-	*id*
5	5	-	-	-	-	-	*space*

Sketch of table: scan_tab and token_tab. See Scott for details.

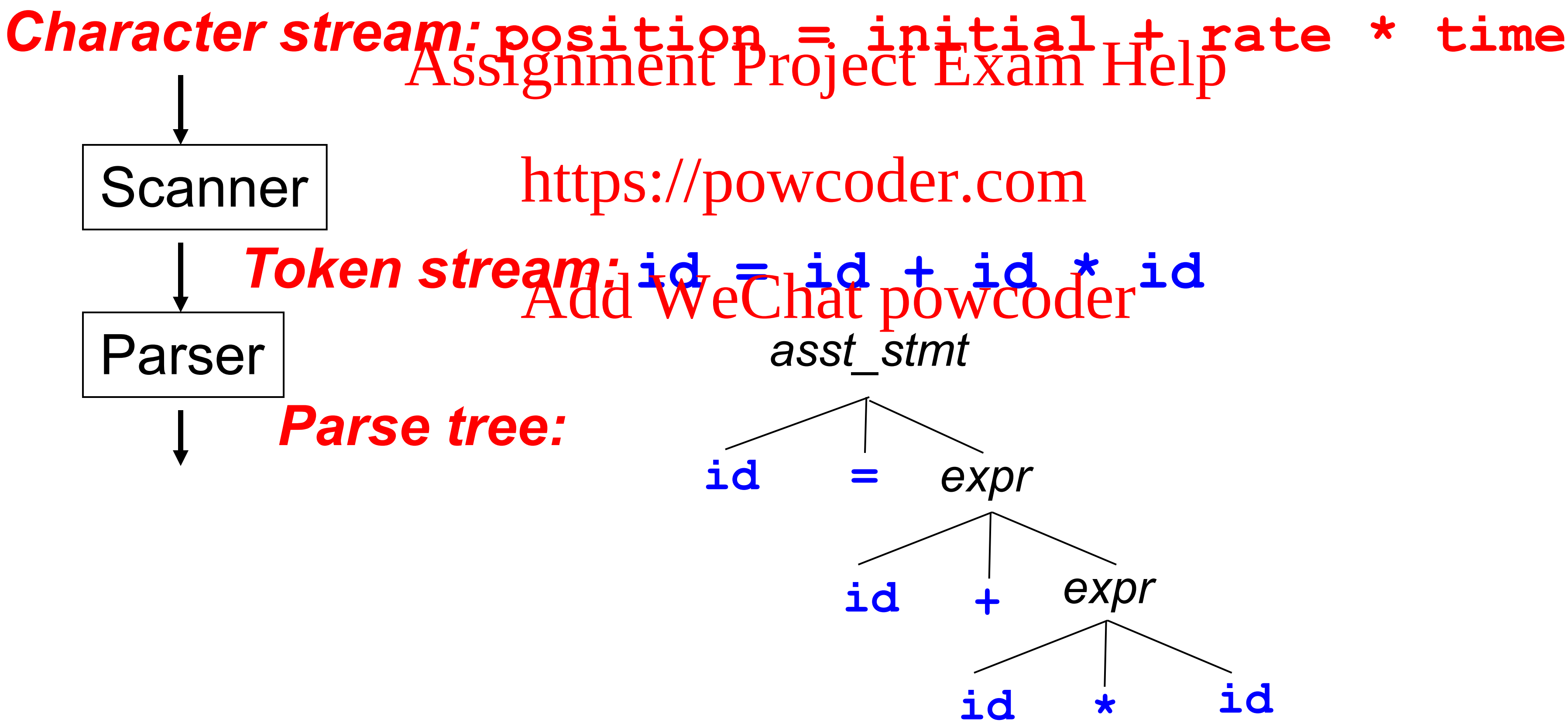
Today's Lecture Outline

- Overview of scanning
- Overview of top-down and bottom-up parsing

- Top-down parsing
  - Recursive descent
  - LL(1) parsing tables

A Simple Calculator Language

asst_stmt → **id** **=** *expr* // *asst_stmt is the start symbol*
expr → *expr* **+** *expr* | *expr* ***** *expr* | **id**



(Parse tree simplified to fit on slide.)

A Simple Calculator Language

asst_stmt $\rightarrow$ **id =** *expr* // *asst_stmt is the start symbol*

expr $\rightarrow$ *expr* **+** *expr* | *expr* ***** *expr* | **id**

Character stream: **position + initial = rate * time**

Scanner

Token stream: **id + …**

Parser

Parse tree:

Token stream is ill-formed according to our grammar, parse tree construction fails, therefore Syntax error!

Most compiler errors occur in the parser.

Parsing

- For any CFG, one can build a parser that runs in $O(n^3)$
  - Well-known algorithms

- But $O(n^3)$ time is unacceptable for a parser in a compiler!

Parsing

- Objective: build a parse tree for an input string of tokens from a single scan of input
  - Only special subclasses of context-free grammars (LL and LR) can do this

- Two approaches
  - Top-down: builds parse tree from the root to the leaves
  - Bottom-up: builds parse tree from the leaves to the top
  - Both are easily automated

Grammar for Comma-separated Lists

list $\rightarrow$ **id** *list_tail*   // *list is the start symbol*

list_tail $\rightarrow$ **,** **id** *list_tail* | **;**

Generates comma-separated lists of **id**'s.

E.g., **id ;**   **id , id , id ;**

Example derivation:

list $\Rightarrow$ **id** *list_tail*

$\Rightarrow$ **id , id** *list_tail*

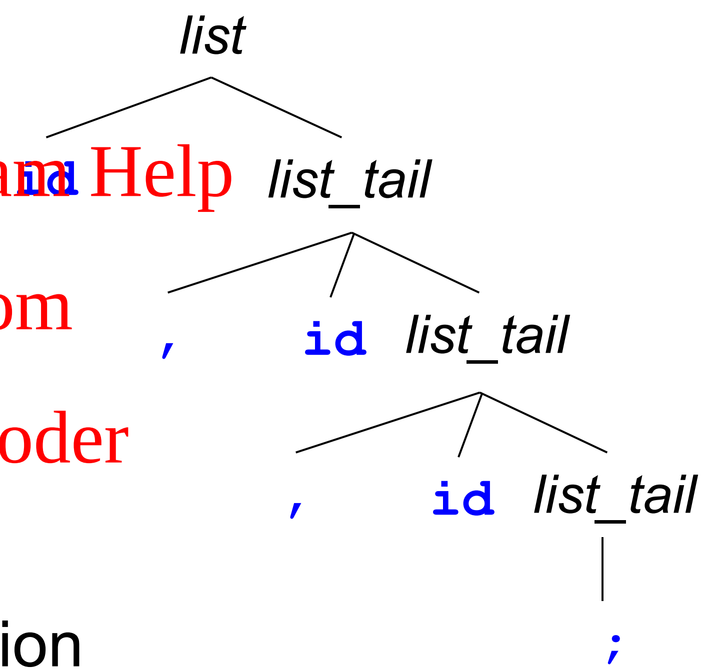
$\Rightarrow$ **id , id ;**

Top-down Parsing

list → **`id`** *list_tail*
list_tail → **`,`** **`id`** *list_tail* | **`;`**

- Terminals are seen in the order of appearance in the token stream

`id , id , id ;`



- The parse tree is constructed
  - From the top to the leaves
  - Corresponds to a left-most derivation
- Look at left-most nonterminal in current sentential form, and lookahead terminal and “predict” which production to apply

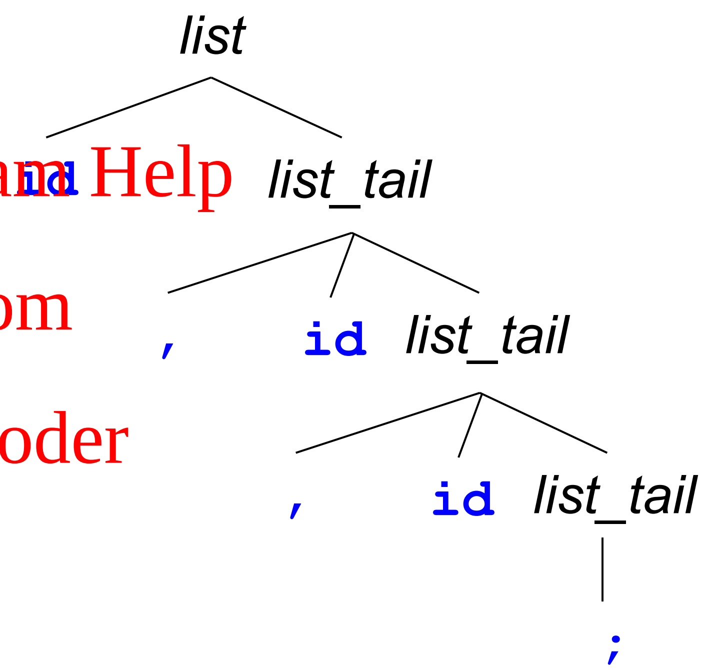
Bottom-up Parsing

list → `id` *list_tail*
list_tail → `,` `id` *list_tail* | `;`

- Terminals are seen in the order of appearance in the token stream

  `id , id , id ;`

- The parse tree is constructed
  - From the leaves to the top
  - A right-most derivation in reverse



Today's Lecture Outline

- Overview of scanning
- Overview of top-down and bottom-up parsing

- Top-down parsing
  - Recursive descent
  - LL(1) parsing tables

Top-down Predictive Parsing

- "Predicts" production to apply based on one or more lookahead token(s)
- Predictive parsers work with LL(k) grammars

  - First L stands for "left-to-right" scan of input

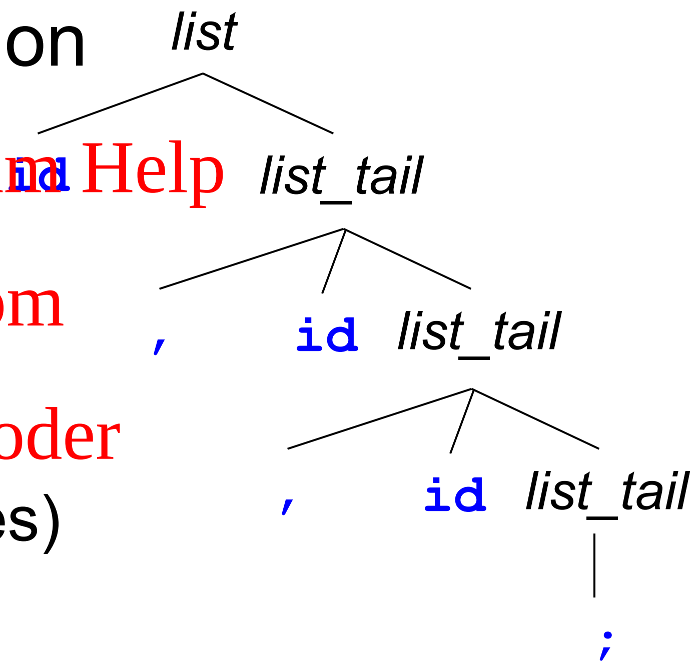
  - Second L stands for "left-most" derivation

    - Parse corresponds to left-most derivation
  - k stands for "need k tokens of lookahead to predict"
- We are interested in LL(1)

Question

list → **id** *list_tail*
list_tail → **,** **id** *list_tail* | **;**

- Can we always predict (i.e., for <u>any</u> input) what production to applies, based on one token of lookahead?



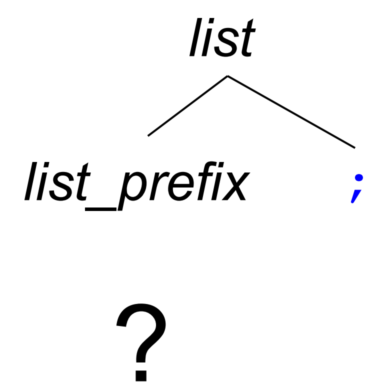
id , id , id ;

  - Yes, there is at most one choice (i.e., at most one production applies)
  - This grammar is an LL(1) grammar

Question

list → *list_prefix* **;**
list_prefix → *list_prefix* **, id** | **id**

- A new grammar
- What language does it generate?
  - Same, comma-separated lists
- Can we predict based on **one** token of lookahead?




```
id , id , id ;
↑
```

Top-down Predictive Parsing

- Back to predictive parsing

- "Predicts" production to apply based on one or more lookahead token(s)
  - Parser always gets it right!
  - There is no need to backtrack, undo expansion, then try a different production

- Predictive parsers work with LL(k) grammars

Top-down Predictive Parsing

- Expression grammar:
  - Not LL(1)

$$
\begin{aligned}
expr &\rightarrow expr \; \mathbf{+} \; expr \\
&\;\;|\; expr \; \mathbf{*} \; expr \\
&\;\;|\; \mathbf{id}
\end{aligned}
$$

- Unambiguous version:
  - Still not LL(1). Why?

$$
\begin{aligned}
expr &\rightarrow expr \; \mathbf{+} \; term \;|\; term \\
term &\rightarrow term \; \mathbf{*} \; \mathbf{id} \;|\; \mathbf{id}
\end{aligned}
$$

- LL(1) version:

$$
\begin{aligned}
expr &\rightarrow term \; term\_tail \\
term\_tail &\rightarrow \mathbf{+} \; term \; term\_tail \;|\; \varepsilon \\
term &\rightarrow \mathbf{id} \; factor\_tail \\
factor\_tail &\rightarrow \mathbf{*} \; \mathbf{id} \; factor\_tail \;|\; \varepsilon
\end{aligned}
$$

Exercise

expr → *term* *term_tail*
term_tail → **+** *term* *term_tail* | **ε**
term → **id** *factor_tail*
factor_tail → ***** **id** *factor_tail* | **ε**

- Draw parse tree for expression

id + id * id + id

Recursive Descent

- Each <u>nonterminal</u> has a procedure
- The right-hand-sides (rhs) for the nonterminal form the body of its procedure

- lookahead()
  - Peeks at current token in input stream
- match(t)
  - if lookahead() == t then consume current token, else PARSE_ERROR

Recursive Descent

start → *expr* **$$**

expr → *term term_tail*

term → **id** *factor_tail*

term_tail → **+** *term term_tail* | **ε**

factor_tail → ***** **id** *factor_tail* | **ε**

```
start()
    case lookahead() of
        id:   expr(); match($$)          ($$ - end-of-input marker)
        otherwise PARSE_ERROR

expr()
    case lookahead() of
        id:   term(); term_tail()
        otherwise PARSE_ERROR

term_tail()
    case lookahead() of
        +:   match( '+' ); term(); term_tail()
        $$:  skip
        otherwise: PARSE_ERROR
```

Predicting production *term_tail* → **+** *term term_tail*

Predicting epsilon production *term_tail* → **ε**

Recursive Descent

start → *expr* **$$**
expr → *term term_tail*
term → **id** *factor_tail*
term_tail → **+** *term term_tail* | **ε**
factor_tail → ***** **id** *factor_tail* | **ε**

term()
```
    case lookahead() of
        id: match('id'); factor_tail()
        otherwise: PARSE_ERROR
```


factor_tail()
```
    case lookahead() of
        *: match('*'); match('id'); factor_tail();
        +,$$: skip
        otherwise PARSE_ERROR
```

Predicting production *factor_tail* → ***id** *factor_tail*

Predicting production *factor_tail* → **ε**

LL(1) Parsing Table

- But how does the parser "predict"?
  - E.g., how does the parser know to expand a *factor_tail* by *factor_tail* → **ε** on **+** and **$$**?
- It uses the LL(1) parsing table
  - One dimension is nonterminal to expand
  - Other dimension is lookahead token
    - We are interested in one token of lookahead
  - Entry "nonterminal on token" contains the production to apply or contains nothing

LL(1) Parsing Table

- One dimension is nonterminal to expand
- Other dimension is lookahead token

	a
A	α

- E.g., entry "nonterminal A on terminal **a**" contains production $A \rightarrow \alpha$

Meaning: when parser is at nonterminal A and lookahead token is **a**, then parser expands A by production $A \rightarrow \alpha$

LL(1) Parsing Table

start → *expr* **$$**
expr → *term term_tail*
term_tail → **+** *term term_tail* | **ε**
term → **id** *factor_tail*
factor_tail → ***** **id** *factor_tail* | **ε**

	id	**+**	*****	**$$**
start	*expr* **$$**	-	-	-
expr	*term term_tail*	-	-	-
term_tail	-	**+** *term term_tail*	-	**ε**
term	**id** *factor_tail*	-	-	-
factor_tail	-	**ε**	***** **id** *factor_tail*	**ε**

Question

- Fill in the LL(1) parsing table for the comma-separated list grammar

start → *list* **$$**
list → **id** *list_tail*
list_tail → **,** **id** *list_tail* | **;**

	id	**,**	**;**	**$$**
start	*list* **$$**	–	–	–
list	**id** *list_tail*	–	–	–
list_tail	–	**,** **id** *list_tail*	**;**	–

The End