- *Quiz 1 today, will activate when it's time*
  - *On context-free grammars*
- *In Submitty, please have your laptops ready*
- *25 min limit FROM when you open the quiz*
- *Won't be accessible after class*
- *Work together, use notes and search*
- *Generative AI is NOT allowed*

1

Programming Language Syntax: Scanning and Parsing

Read: Scott, Chapter 2.2 and 2.3.1

2

Lecture Outline

- *Catchup: Disambiguating expression grammars*
- *Overview of scanning*
- *Overview of top-down and bottom-up parsing*

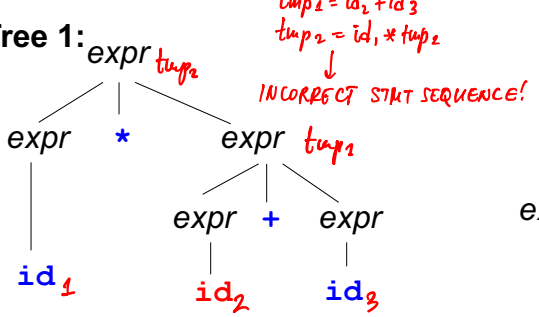
- *Top-down parsing*
  - *Recursive descent*
  - *Predictive parsing*
  - *LL(1) grammars and tables*

Ambiguity

expr → **id** | **(** *expr* **)** | *expr* **+** *expr* | *expr* ***** *expr*

- How many parse trees for **id * id + id** ?

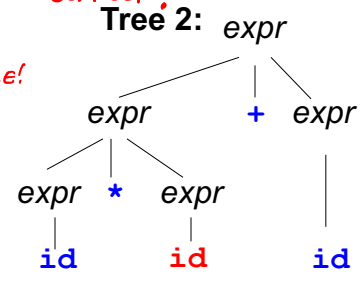
Tree 1:



$tmp_1 = id_2 + id_3$
$tmp_2 = id_1 * tmp_1$
INCORRECT STMT SEQUENCE!

CORRECT!

Tree 2:



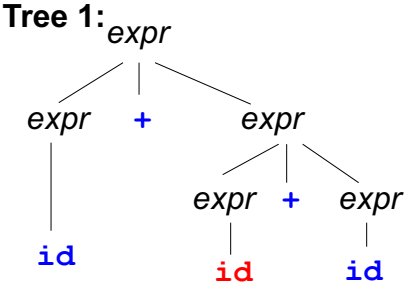
- Which one is "correct"? Tree 2.

Ambiguity

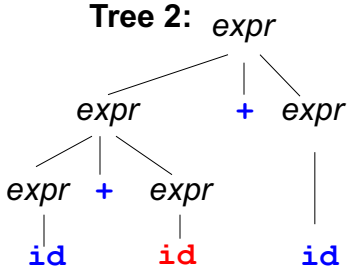
expr → **id** | **(** *expr* **)** | *expr* **+** *expr* | *expr* ***** *expr*

- How many parse trees for **id + id + id** ?

Tree 1:



Tree 2:



- Which one is "correct"?

5

5

Expression Grammars

- Generate expressions
  - Arithmetic expressions
  - Regular expressions
  - Other
- Strings are made up of operands, operators, and parentheses. E.g.,

expr → **id** | **(** *expr* **)** | *expr* **+** *expr* | *expr* ***** *expr*

6

Handling Ambiguity

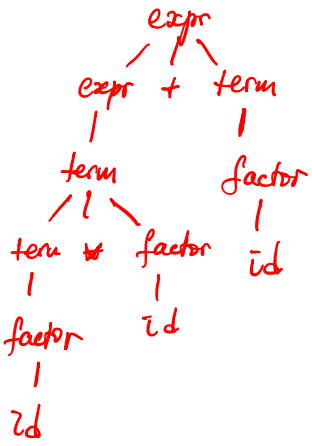
Our ambiguous grammar:
expr → **id** | **(** *expr* **)** | *expr* **+** *expr* | *expr* ***** *expr*

- Can we disambiguate the grammar?
- I.e., rewrite into a new grammar s.t. it is a) unambiguous, and b) generates same language:
  expr → *expr* **+** *term* | *term* LEFT RECURSION
- *term* → *term* ***** *factor* | *factor*
- *factor* → **id** | **(** *expr* **)**

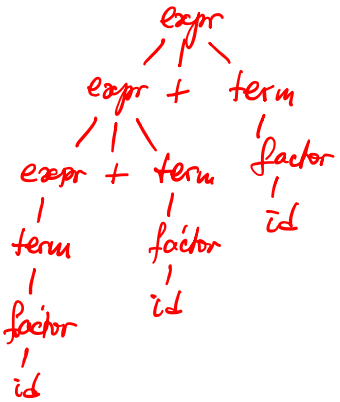
Example?

expr → expr + term | term
term → term * factor | factor
factor → id | (expr)

id * id + id :



id + id + id :



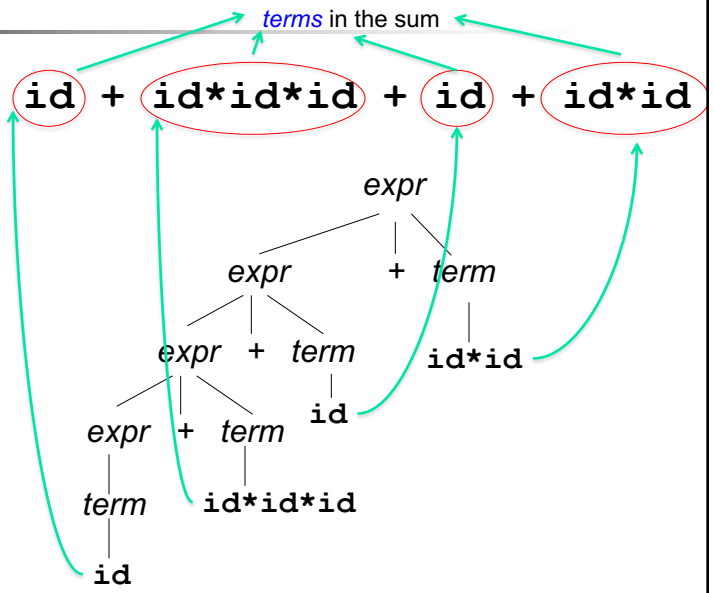
Rewriting Expression Grammars: Intuition

expr → **id** | **(** *expr* **)** | *expr* **+** *expr* | *expr* ***** *expr*

- A new nonterminal, *term*
- *expr* ***** *expr* becomes *term*. Pushes ***** down the tree, forcing higher precedence of ***** over **+**
- *expr* **+** *expr* becomes *expr* **+** *term*. Pushes leftmost **+** down the tree, forcing operand to associate with **+** on its left
  - *expr* → *expr* **+** *expr* becomes *expr* → *expr* **+** *term* | *term*

Rewriting Expression Grammars: Intuition

E.g., look at **id + id*id*id + id + id*id**



Rewriting Expression Grammars: Intuition

- To parse terms, we add a new nonterminal, *factor,* and productions:
  - *term* → *term* `*` *factor* | *factor*
  - *factor* → `id` / `(` *expr* `)`

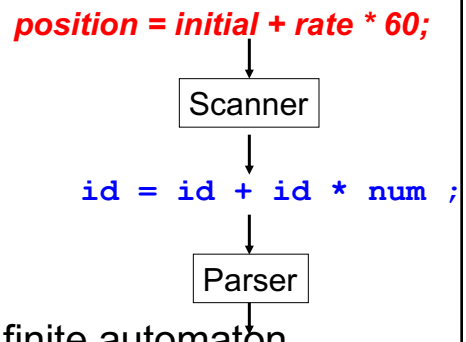
Lecture Outline

- *Catchup: Disambiguating expression grammars*
- *Overview of scanning*
- *Overview of top-down and bottom-up parsing*

- *Top-down parsing*
  - *Recursive descent*
  - *Predictive parsing*
  - *LL(1) grammars and tables*

Scanning

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



- Scanner is essentially a finite automaton
  - Regular expressions specify the syntax of tokens
  - Scanner recognizes the tokens in the program

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

Calculator Language

- Tokens, specified with regular expressions:

times → *****

plus → **+**

id → *letter* **(** *letter* | *digit* **)***

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

The Scanner as a DFA

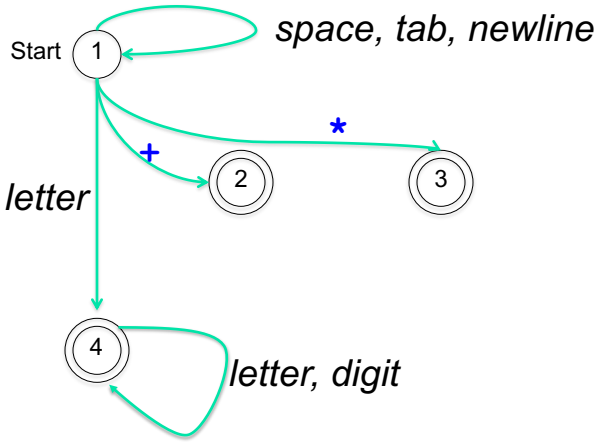


Table-Driven Scanning

```
…
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[cur_state]
            unread cur_char --- push back char
            exit loop
        error:
```


Table-Driven Scanning

scan_tab token_tab

	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.

Lecture Outline

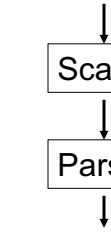
- *Catchup: Ambiguity and expression grammars*
- *Overview of scanning*
- *Overview of top-down and bottom-up parsing*

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

A Simple Calculator Language

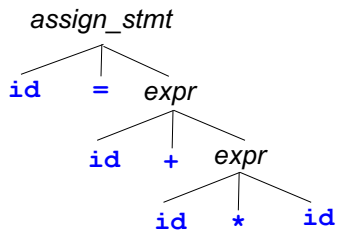
assign_stmt → `id =` *expr* // *assign_stmt is the start symbol*
expr → *expr* `+` *expr* | *expr* `*` *expr* | `id`

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



Token stream: `id = id + id * id`

Parse tree:

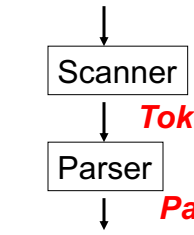


(Parse tree simplified to fit on slide.)

A Simple Calculator Language

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

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



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

Most compiler errors occur in the parser.

Parsing

- Given an arbitrary CFG, one can build a parser that parses a string of length *n* in (essentially) $O(n^3)$ time
  - Well-known algorithms
- But $O(n^3)$ time is unacceptable for a compiler

Parsing

- Objective: build a parse tree for an input string of tokens from a single scan of input
  - 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 → **`id`** *list_tail* // *list is the start symbol*
list_tail → **`,`** **`id`** *list_tail* | **`;`**

Generates comma-separated lists of **`id`**'s.
E.g., **`id ;`** **`id, id, id ;`**

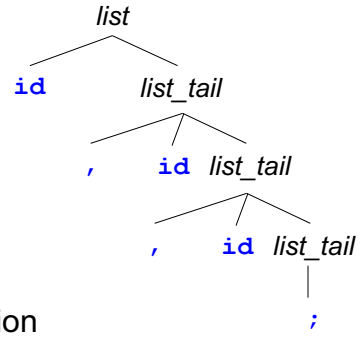
Example derivation:
list ⇒ **`id`** *list_tail*
⇒ **`id , id`** *list_tail*
⇒ **`id , id ;`**

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 leftmost derivation
- Look at leftmost nonterminal in current sentential form, and lookahead terminal and "predict" which production to apply

25

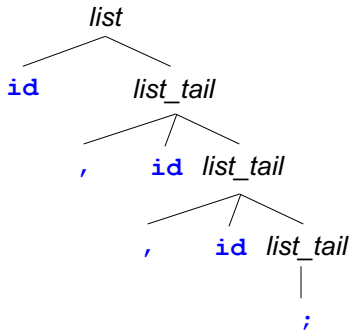
25

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 rightmost derivation in reverse

26

Lecture Outline

- *Catchup: Disambiguating expression grammars*
- *Overview of scanning*
- *Overview of top-down and bottom-up parsing*

- *Top-down parsing*
  - *Recursive descent*
  - *Predictive parsing*
  - *LL(1) grammars and tables*

Recursive Descent

- Each nonterminal has a procedure
- Procedure (roughly speaking) "matches" the nonterminal in the input string

- lookahead()
  - Peeks at current token in input stream
- match(t) → PARSER EXPECTS TO SEE TERMINAL t
  - if lookahead() == t then consume current token, else ERROR ip = ip+1

Recursive Descent

A → rhs1
A → rhs2
A → rhs3

- Procedure for nonterminal *A*

NONTERMINAL

```
procedure A()
   choose an A-production, A → X1 X2 … Xk
   for ( i = 1 to k )
        if Xi is a nonterminal
          call procedure Xi()
        else match(Xi)
```

SEQUENCE OF TERMINALS AND NONTERMINALS

- Predictive parsers narrow down to one choice
- In general, there is more than one choice and the parser may require backtracking

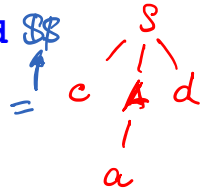
29

29

Backtracking

$S \rightarrow$ **c** A **d**
$A \rightarrow$ **a b** | **a**

- ERROR in **match** is not ultimate PARSE_ERROR:
  - Return to **choose**
  - Try next production
  - Only if no production left, then PARSE_ERROR
  - Need to undo consumed input
- E.g., parse **cad** $



S()
match(c) ✓
A() ✓
Try A→ab
match(a) ✓
match(b) ✗
backtrack
Try A→a
match(a) ✓
match(d) ✓
DONE!

Programming Languages CSCI 4430, A. Milanova

30

30

Homework

$S \rightarrow \mathbf{a}\ S_1\ \mathbf{b}\ S_2 \mid \mathbf{b}\ S_1\ \mathbf{a}\ S_2 \mid \varepsilon$

(1) (2) ~~(3)~~

- Try order (1), (2), (3) first: matches longest S_1. Order (3), (2), (1) matches shortest S_1

abS$ S()

try aSbS
match(a) V
S() V
try aSbS:
match(a) X
backtrack
try bSaS:
match(b) V
S() X fails
backtrack
try ε V
match(b) V
S() : ε V

procedure S()
  save input pointer
  try match(**a**); S(); match(**b**); S()
  if success, return
  reset input pointer
  try match(**b**); S(); match(**a**); S()

if success, return
reset input pointer
try ε: return // always succeeds

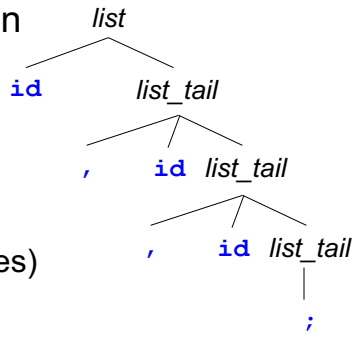
Predictive Parsing

- "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 leftmost derivation
    - Parse corresponds to leftmost 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 any input) what production to apply, based on just one token of lookahead?



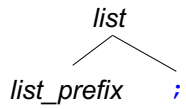
id , id , id ;

- 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 ;

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 and try a different production
- Predictive parsers work with LL(k) grammars

Predictive Parsing

- Expression grammar:
  - Not LL(1)

expr → *expr* **+** *expr*
| *expr* ***** *expr*
| **id**

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

expr → *expr* **+** *term* | *term*
term → *term* ***** **id** | **id**

- LL(1) version:

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

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

Predictive 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*() ← Predicting production *term_tail* → **+** *term* *term_tail*
    $$: skip ← Predicting epsilon production *term_tail* → **ε**
    otherwise: PARSE_ERROR

Predictive 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
  - Table 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 → **id** *factor_tail*
term_tail → **+** *term term_tail* | **ε**
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