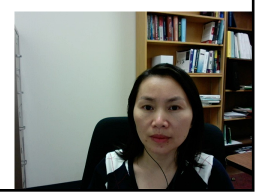
CSCI-C311 Programming Languages

LL(1) Parsers

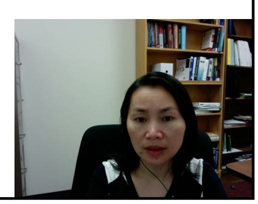
Dr. Hang Dinh



1

Outline and Reading

- After this lecture, you will learn
  - LL(1) parsing algorithm
  - Recursive descent parsers
  - Table-driven top down parsing
- Reading
  - Scott 4e –Section 2.3.1
  - Scott 4e –Section 2.3.3



2

LL Parsing Overview

- Recall: The **LL** class of linear-time parsing algorithms
  - Scans input left-to-right
  - Discovers *left-most derivation*
  - Constructs parse tree in *top-down* fashion
  - Uses *predictive* algorithms
- Class **LL(1)**
  - Subclass of **LL**
  - Uses just one token of look-ahead

3

LL(1) Grammar Example

- **LL(1)** grammar for a simple "calculator" language (part 1):

$$
\begin{aligned}
&\textit{program} \longrightarrow \textit{stmt\_list}\ \texttt{\$\$} \\
&\textit{stmt\_list} \longrightarrow \textit{stmt}\ \textit{stmt\_list} \mid \epsilon \\
&\textit{stmt} \longrightarrow \texttt{id := } \textit{expr} \mid \texttt{read id} \mid \texttt{write } \textit{expr} \\
&\textit{expr} \longrightarrow \textit{term}\ \textit{term\_tail} \\
&\textit{term\_tail} \longrightarrow \textit{add\_op}\ \textit{term}\ \textit{term\_tail} \mid \epsilon
\end{aligned}
$$

The end marker token $$ is produced by the scanner at the end of the input.

4

LL(1) Grammar Example

- **LL(1)** grammar for a simple "calculator" language (part 2):

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

5

LL(1) Parsing Example

- Example of input string: The "sum-and-average" program

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

- How do we parse a string with the "calculator" grammar?
  - by building the parse tree incrementally
  - start at the top of the tree and predict needed rules based on the current left-most nonterminal in the tree and the current input token.

6

Two Approaches to LL(1) Parsing

- The **recursive descent** approach
  - build a *recursive descent parser* whose subroutines correspond, one-to-one, to the non-terminals of the grammar
  - Recursive descent parsers are typically constructed by hand, but can be constructed automatically by the ANTLR parser generator.
- The **table-driven** approach
  - Build an *LL parse table* which is then read by a driver program
  - Table-driven parsers are almost always constructed automatically by a parser generator.

7

Recursive Descent Parser for the Calculator Grammar: Pseudocode Part 1

```
procedure match(expected)
    if input_token = expected then consume_input_token()
    else parse_error

–– this is the start routine:
procedure program()
    case input_token of
        id, read, write, $$ :
            stmt_list()
            match($$)
        otherwise parse_error
```

$$\boxed{program \longrightarrow stmt\_list \;\; \$\$}$$

8

Recursive Descent Parser for the Calculator Grammar: : Pseudocode Part 2

```
procedure stmt_list()
    case input_token of
        id, read, write : stmt(); stmt_list()
        $$ : skip        –– epsilon production
        otherwise parse_error

procedure stmt()
    case input_token of
        id : match(id); match(:=); expr()
        read : match(read); match(id)
        write : match(write); expr()
        otherwise parse_error
```

stmt_list $\longrightarrow$ *stmt stmt_list* | ϵ

stmt $\longrightarrow$ `id := ` *expr* | `read id` | `write` *expr*

9

Recursive Descent Parser for the Calculator Grammar: Pseudocode Part 3

```
procedure expr()
    case input_token of
        id, number, ( : term(); term_tail()
        otherwise parse_error

procedure term_tail()
    case input_token of
        +, - : add_op(); term(); term_tail()
        ), id, read, write, $$ :
            skip        –– epsilon production
        otherwise parse_error

procedure term()
    case input_token of
        id, number, ( : factor(); factor_tail()
        otherwise parse_error
```

expr $\longrightarrow$ *term term_tail*

term_tail $\longrightarrow$ *add_op term term_tail* | ϵ

term $\longrightarrow$ *factor factor_tail*

factor_tail $\longrightarrow$ *mult_op factor factor_tail* | ϵ

factor $\longrightarrow$ `(` *expr* `)` | `id` | `number`

add_op $\longrightarrow$ `+` | `-`

10

Recursive Descent Parser for the Calculator Grammar: Pseudocode Part 4

```
procedure factor_tail()
    case input_token of
        *, / : mult_op(); factor(); factor_tail()
        +, -, ), id, read, write, $$ :
            skip        -- epsilon production
        otherwise parse_error

procedure factor()
    case input_token of
        id : match(id)
        number : match(number)
        ( : match((); expr(); match())
        otherwise parse_error
```

$factor\_tail \longrightarrow mult\_op\ factor\ factor\_tail \mid \epsilon$

$factor \longrightarrow$ ($expr$) | **id** | **number**

$add\_op \longrightarrow$ + | -

$mult\_op \longrightarrow$ * | /

11

Recursive Descent Parser for the Calculator Grammar: Pseudocode Part 5

```
procedure add_op()
    case input_token of
        + : match(+)
        - : match(-)
        otherwise parse_error

procedure mult_op()
    case input_token of
        * : match(*)
        / : match(/)
        otherwise parse_error
```

$add\_op \longrightarrow$ + | -

$mult\_op \longrightarrow$ * | /

12

Recursive Descent Parser for the Calculator Grammar: Generate Parse Tree

- Without additional code, the given pseudocode merely verifies that the input program is syntactically correct
  - i.e., when the `parse_error` subroutine is never executed
- To save the parse tree itself,
  - allocate and link together records to represent the children of a node immediately before executing the nonterminal subroutines and `match`.
  - need to pass each nonterminal routine an argument that points to the record that is expanded, i.e., whose children are to be discovered.
  - Procedure `match` needs to save information about certain token in the leaves of the tree.

13

Recursive Descent Parser: General Technique

- The trickiest part of writing a recursive descent parser is to figuring out which tokens should label the arms of the `case` statements.
  - Each arm represents one production: one possible expansion of the non-terminal corresponding to the subroutine.
  - The tokens that label the a given arm are those that *predict* the production.
- A token X may predict a production for either of two reasons:
  - The right-hand side of the production, when recursively expanded, may yield a string beginning with X
  - The right-hand side may yield nothing (empty string) and X may begin the yield of what come *next*.

14

Table-Driven LL(1) Parsing

- Idea is to maintain a **parse stack** as follows:
  - Initialize the stack with the start symbol of the input grammar

Loop

  - Pop the top symbol *T* off from stack
  - If the top symbol *T* is a nonterminal,
    - Predict a production $T \rightarrow w_1\ w_2 \ldots .\ w_k$ (each w_i is a terminal or nonterminal)
    - Push w_k, w_{k-1},…, w_2, w_1 in that order into stack
  - If the top symbol *T* is a terminal,
    - If *T* is the end marker, then stop
    - Otherwise, match *T* with current token; if not match, produce a syntax error.
- The driver for a table-driven **LL(1)** parser implements algorithm above.

15

Parse Stack Example

- Parse stack for sum-and-average program in the calculator grammar

Parse stack	Input stream	Comment
program	`read A read B ...`	initial stack contents
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`

See the entire parse stack in Figure 2.21 in Scott 4e

16

1. *program* ⟶ *stmt_list* $$
2. *stmt_list* ⟶ *stmt* *stmt_list*
3. *stmt_list* ⟶ ϵ

LL(1) Parse Table

- To predict a production, the table-driven LL(1) parser needs to look up the **LL(1) parse table** for the given grammar.

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	–

production to predict

A dash indicates an error

17

Predict Sets

- To write a recursive descent parser or construct an LL(1) parse table, we need to compute the **predict set** for each production.
- The predict set of a production $A \rightarrow \alpha$
  - is denoted $\text{PREDICT}(A \rightarrow \alpha)$
  - is the set of all tokens that label the arms of the `case` statement corresponding to production $A \rightarrow \alpha$ in subroutine A in recursive descent parser.
  - Example:

PREDICT(*stmt_list* → *stmt* *stmt_list*) = {id, read, write}

PREDICT(*stmt_list* → ε) = {$$}

```
procedure stmt_list()
    case input_token of
        id, read, write : stmt(); stmt_list()
        $$ : skip          -- epsilon production
        otherwise parse_error
```

18

Formalizing Notion of Prediction

- A token X may belong to $\text{PREDICT}(A \rightarrow \alpha)$ for either of two reasons:
  - Nonterminal A, when recursively expanded, may yield a string beginning with X
  - Nonterminal A may yield ε and X may begin the yield of what come *after* A.
- This notion of prediction is formalized by two sets FIRST and FOLLOW
  - For any string α of terminals and non-terminals, $\text{FIRST}(\alpha)$ is the set of all tokens that could be the start of a string obtained by recursively expanding α

$$\text{FIRST}(\alpha) = \{\text{c} \mid \alpha \Longrightarrow^{*} \text{c}\ \beta \text{ for some string } \beta\}$$

  - For any nonterminal A, $\text{FOLLOW}(A)$ is the set of all tokens that could come after A in a string obtained by recursively expanding the start symbol.

$$\text{FOLLOW}(A) = \{\text{c} \mid \text{S} \Longrightarrow^{+} \alpha\ A\ \text{c}\ \beta \text{ for some strings } \alpha, \beta\}$$

S is start symbol

$\Longrightarrow^{*}$ means "derives after zero or more replacements"
$\Longrightarrow^{+}$ means "derives after one or more replacements".

19

Calculating FIRST Sets in Calculator Grammar

- $\text{FIRST}(\alpha) = \{\text{c} \mid \text{c is a token such that } \alpha \Longrightarrow^{*} \text{c}\ \beta \text{ for some string } \beta\}$
- $\text{FIRST}(\text{c}) = \{\text{c}\}$ for any token c
  - Therefore, FIRST($$) = {$$}
- $\text{FIRST}(\mathit{program}) = \text{FIRST}(\mathit{stmt\_list}) \cup \text{FIRST}(\$\$) = \{\text{id}, \text{read}, \text{write}, \$\$\}$
  - Note: *stmt_list* yields ε (i.e., $\mathit{stmt\_list} \Longrightarrow^{*} \varepsilon$)
- $\text{FIRST}(\mathit{stmt\_list}) = \text{FIRST}(\mathit{stmt}\ \mathit{stmt\_list}) = \text{FIRST}(\mathit{stmt})$
  - Note: *stmt* does not yield ε
- $\text{FIRST}(\mathit{stmt}) = \{\text{id}, \text{read}, \text{write}\}$

$$\begin{aligned} &\mathit{program} \longrightarrow \mathit{stmt\_list}\ \$\$ \\ &\mathit{stmt\_list} \longrightarrow \mathit{stmt}\ \mathit{stmt\_list} \mid \epsilon \\ &\mathit{stmt} \longrightarrow \texttt{id := } \mathit{expr} \mid \texttt{read id} \mid \texttt{write } \mathit{expr} \end{aligned}$$

20

Calculating FOLLOW Sets in Calculator Grammar

- $\text{FOLLOW}(A) = \{\text{c} \mid \text{S} \Longrightarrow^{+} \alpha\ A\ \text{c}\ \beta \text{ for some strings } \alpha, \beta\}$
- $\text{FOLLOW}(program) = \{\ \}$ (empty set)
  - When we expand $program$ we'll never see the $program$ symbol again.
- $\text{FOLLOW}(stmt\_list) = \{\$\$\ \}$
  - When we expand $program$, the only token that can come after $stmt\_list$ is $$
- $\text{FOLLOW}(stmt) = \text{FIRST}(stmt\_list) \cup \{\$\$\} = \{\text{id}, \text{read}, \text{write}, \$\$\ \}$
  - $program \Longrightarrow^{+} stmt\ stmt\_list\ \$\$$
  - $\Longrightarrow\ stmt\ \$\$$

$program \longrightarrow stmt\_list\ \ \$\$$

$stmt\_list \longrightarrow stmt\ \ stmt\_list \mid \epsilon$

$stmt \longrightarrow \texttt{id := } expr \mid \texttt{read id} \mid \texttt{write } expr$

21

Calculating PREDICT Sets

- $\text{PREDICT}(A \to \alpha) =$
  - $\text{FIRST}(\alpha)$ if α does not yield ε
  - $\text{FIRST}(\alpha) \cup \text{FOLLOW}(A)$ if α yields ε
- Examples in the calculator grammar
  - $\text{PREDICT}(stmt\_list \to stmt\ stmt\_list) = \text{FIRST}(stmt\ stmt\_list)$
    $= \{\text{id}, \text{read}, \text{write}\}$
  - $\text{PREDICT}(stmt\_list \to \varepsilon) = \text{FIRST}(\varepsilon) \cup \text{FOLLOW}(stmt\_list) = \{\$\$\ \}$
- The grammar is not LL(1) if
  - In the process of calculating PREDICT sets, we find that some tokens belongs to the PREDICT set of more than one production with the same left hand side.

22