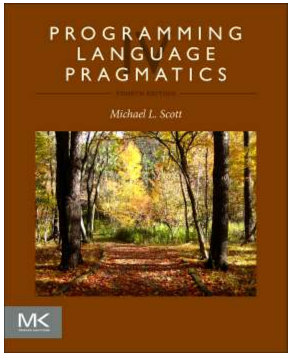
Top-Down Parsing

17-363/17-663: Programming Language Pragmatics



Reading: PLP section 2.3

Prof. Jonathan Aldrich

Parsing

- A context-free grammar (CFG) is a *generator* for a context-free language (CFL)

– A parser is a language *recognizer*

- There are an infinite number of grammars for every context-free language
  - Not all grammars are created equal, however
    - Ambiguity
    - Understandability
    - Performance

Parsing

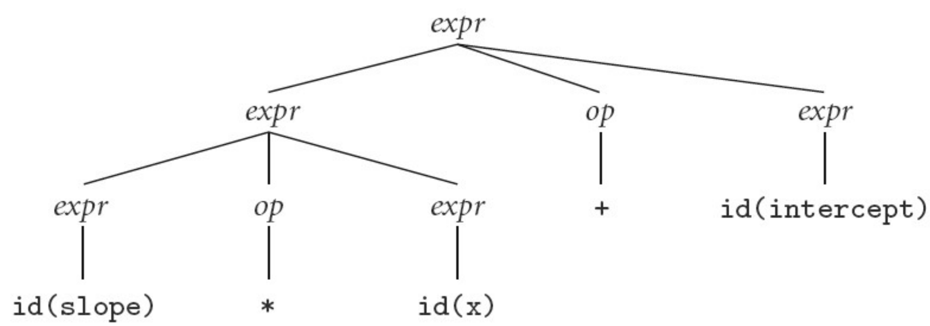
- It turns out that for any CFG we can create a parser that runs in $O(n^3)$ time
  - E.g. the Generalized LR (GLR) parser used to parse expressions in SASyLF
- $O(n^3)$ time is clearly unacceptable for a parser in a compiler - too slow
  - It's OK in SASyLF because we only write small expressions in proofs

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 vs. Rightmost Derivations

$$
\begin{aligned}
expr &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\
&\quad\ \mid expr\ op\ expr \\
op &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$



Leftmost derivation

- Always chooses the left-most nonterminal to replace

$$
\begin{aligned}
expr &\Rightarrow \underline{expr}\ op\ expr \\
&\Rightarrow \underline{expr}\ op\ expr\ op\ expr \\
&\Rightarrow \text{id}\ \underline{op}\ expr\ op\ expr \\
&\Rightarrow \text{id} * \underline{expr}\ op\ expr \\
&\Rightarrow \text{id} * \text{id}\ \underline{op}\ expr \\
&\Rightarrow \text{id} * \text{id} + \underline{expr} \\
&\Rightarrow \text{id} * \text{id} + \text{id}
\end{aligned}
$$

Rightmost derivation

- Always chooses the right-most nonterminal to replace

$$
\begin{aligned}
expr &\Rightarrow expr\ op\ \underline{expr} \\
&\Rightarrow expr\ \underline{op}\ \text{id} \\
&\Rightarrow \underline{expr} + \text{id} \\
&\Rightarrow expr\ op\ \underline{expr} + \text{id} \\
&\Rightarrow expr\ \underline{op}\ \text{id} + \text{id} \\
&\Rightarrow \underline{expr} * \text{id} + \text{id} \\
&\Rightarrow \text{id} * \text{id} + \text{id}
\end{aligned}
$$

- Note: both derivations produce the same tree!

Parsing

- LL parsers are also called 'top-down', or 'predictive' parsers & LR parsers are also called 'bottom-up', or 'shift-reduce' parsers
  - We'll discuss LL parsers today, and LR parsers in the next lecture
- There are several important sub-classes of LR parsers
  - SLR
  - LALR
- We won't be going into detail on the differences between them

Parsing

- You commonly see LL or LR (or whatever) written with a number in parentheses after it
  - This number indicates how many tokens of look-ahead are required in order to parse
  - Almost all real compilers use one token of look-ahead
    - Some tools let you special-case to look further ahead for certain constructs
- The expression grammar (with precedence and associativity) you saw before is LR(1), but not LL(1)
- Every CFL that can be parsed deterministically has an SLR(1) grammar (which is LR(1))

LL Parsing Example

- Let's start with the following statement grammar
  - This is not an LL(1) grammar – we'll see how we need to adapt it

```
program        → stmt_list $$$
stmt_list      → stmt stmt_list
               | ε
stmt           → id := id
               | read id
               | write id
               | id ( id_list )
id_list        → id
               | id_list , id
```

LL Parsing Example

- Let's parse this program:

```
read A
process(A)
write A
```

```
program      → stmt_list $$$
stmt_list    → stmt stmt_list
             | ε
stmt         → id := id
             | read id
             | write id
             | id ( id_list )
id_list      → id
             | id_list , id
```

- Here's the parse sequence

```
program
stmt_list $$$
stmt stmt_list $$$          // based on lookahead = read
read id stmt_list $$$       // based on lookahead = read
stmt_list $$$               // accept read and id tokens
            // what to do here?
            // id lookahead => assign or call
```

ELSEVIER

LL(1) Parsing Requirements

- Whenever making a choice between two productions of a nonterminal…
- It must be possible to predict which is taken based on 1 lookahead token

LL Parsing

- Problems trying to make a grammar LL(1)
  - common prefixes
    - solved by "left-factoring". Example:

```
stmt          → id := expr
              | id ( arg_list )
```

    - This can be expressed instead:

```
stmt          → id id_stmt_tail
id_stmt_tail  → := expr
              | ( arg_list)
```

    - we can left-factor mechanically

LL Parsing

- Problems trying to make a grammar LL(1)
  - left recursion: another thing that LL parsers can't handle
    - Example of left recursion:

```
id_list → id | id_list , id
```

    - This can be expressed instead:

```
id_list      → id id_list_tail
id_list_tail → , id id_list_tail
             | ε
```

    - we can get rid of all left recursion mechanically in any grammar

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
  - the few that arise in practice, however, can generally be handled with kludges

This Grammar is LL(1)

```
program       → stmt_list $$$
stmt_list     → stmt stmt_list
              | ε
stmt          → id id_stmt_tail
              | read id
              | write id
id_stmt_tail  → := id
              | ( id_list )
id_list       → id id_list_tail
id_list_tail  → , id id_list_tail
              | ε
```

LL Parsing Example

- Let's parse this program:

```
read A
process(A)
write A
```

```
program         → stmt_list $$$
stmt_list       → stmt stmt_list | ε
stmt             → id id_stmt_tail
                | read id
                | write id
id_stmt_tail    → := id
                | ( id_list )
id_list         → id id_list_tail
id_list_tail    → , id id_list_tail | ε
```

- Here's the parse sequence

```
program
read id stmt_list $$$ // several steps here
stmt_list $$$          // accept read and id tokens
stmt stmtlist $$$      // based on id lookahead
id id_stmt_tail stmtlist $$$  // based on id lookahead
id_stmt_tail stmtlist $$$     // accept id token
( id ) stmtlist       $$$     // based on ( lookahead
stmtlist $$$           // accept (, id, and ) tokens
write id stmtlist $$$ // two steps, based on id lookahead
stmtlist $$$           // accept write and id tokens
$$$                    // based on $$$ lookahead
```

ELSEVIER

Exercise: LL Grammar Conversion

- Convert the following grammar to LL(1) form

```
program        → expr $$$
expr           → term | expr + term
term           → id | id ( expr )
```

- What are the advantages/disadvantages of your LL(1) grammar compared to the original one (which was LR(1))?

LL Parsing

```
program      → expr $$$
expr         → term expr_tail
expr_tail    → + term expr_tail
             | ε
term         → id term_tail
term_tail    → ( expr )
             | ε
```

- 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 RHS together!
  - however, the simplicity of the parsing algorithm often makes up for this weakness

Top-Down Parsing Implementations

- There are two approaches to LL top-down parsing
  - Recursive Descent – typically handwritten
  - Parse table – typically generated

Recursive descent parsers

```
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

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

procedure stmt()
```

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

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

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

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

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

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

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

- To keep track of the left-most non-terminal, you push the as-yet-unseen portions of productions onto a stack
  - for details see Figure 2.21 in book
    - similar to what we wrote above
- The key thing to keep in mind is that the stack contains all the stuff you expect to see between now and the end of the program
  - what you *predict* you will see

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

LL Parsing

- Algorithm First/Follow/Predict:
  - $\text{FIRST}(\alpha) == \{a : \alpha \rightarrow^{*} a\ \beta\}$
    $\cup$ (if $\alpha \Rightarrow^{*} \varepsilon$ then $\{\varepsilon\}$ else NULL)
  - $\text{FOLLOW}(A) == \{a : S \rightarrow^{+} \alpha\ A\ a\ \beta\}$
    $\cup$ (if $S \rightarrow^{*} \alpha\ A$ then $\{\varepsilon\}$ else NULL)
  - PREDICT $(A \rightarrow X_1 \ldots X_m) ==$
    (FIRST $(X_1 \ldots X_m) - \{\varepsilon\}$)
    $\cup$ (if $X_1, \ldots, X_m \rightarrow^{*} \varepsilon$ then FOLLOW (A)
    else NULL)
- Details following…

LL Parsing

program ⟶ *stmt_list* `$$`	`$$` ∈ FOLLOW(*stmt_list*)
stmt_list ⟶ *stmt* *stmt_list*	
stmt_list ⟶ ϵ	EPS(*stmt_list*) = true
stmt ⟶ `id` `:=` *expr*	`id` ∈ FIRST(*stmt*)
stmt ⟶ `read` `id`	`read` ∈ FIRST(*stmt*)
stmt ⟶ `write` *expr*	`write` ∈ FIRST(*stmt*)
expr ⟶ *term* *term_tail*	
term_tail ⟶ *add_op* *term* *term_tail*	
term_tail ⟶ ϵ	EPS(*term_tail*) = true
term ⟶ *factor* *factor_tail*	
factor_tail ⟶ *mult_op* *factor* *factor_tail*	
factor_tail ⟶ ϵ	EPS(*factor_tail*) = true
factor ⟶ `(` *expr* `)`	`(` ∈ FIRST(*factor*) and `)` ∈ FOLLOW(*expr*)
factor ⟶ `id`	`id` ∈ FIRST(*factor*)
factor ⟶ `number`	`number` ∈ FIRST(*factor*)
add_op ⟶ `+`	`+` ∈ FIRST(*add_op*)
add_op ⟶ `-`	`-` ∈ FIRST(*add_op*)
mult_op ⟶ `*`	`*` ∈ FIRST(*mult_op*)
mult_op ⟶ `/`	`/` ∈ FIRST(*mult_op*)

Figure 2.22 "Obvious" facts (right) about the LL(1) calculator grammar (left).

LL Parsing

FIRST

- *program* {id, read, write, $$}
- *stmt_list* {id, read, write, ϵ}
- *stmt* {id, read, write}
- *expr* {(, id, number}
- *term_tail* {+, -, ϵ}
- *term* {(, id, number}
- *factor_tail* {*, /, ϵ}
- *factor* {(, id, number}
- *add_op* {+, -}
- *mult_op* {*, /}

Also note that FIRST(a) = {a} $\forall$ tokens a.

FOLLOW

- id {+, -, *, /,), :=, id, read, write, $$}
- number {+, -, *, /,), id, read, write, $$}
- read {id}
- write {(, id, number}
- ({(, id, number}
-) {+, -, *, /,), id, read, write, $$}
- := {(, id, number}
- + {(, id, number}
- - {(, id, number}
- * {(, id, number}
- / {(, id, number}
- $$ {ϵ}
- *program* {ϵ}
- *stmt_list* {$$}
- *stmt* {id, read, write, $$}
- *expr* {), id, read, write, $$}
- *term_tail* {), id, read, write, $$}
- *term* {+, -,), id, read, write, $$}
- *factor_tail* {+, -,), id, read, write, $$}
- *factor* {+, -, *, /,), id, read, write, $$}
- *add_op* {(, id, number}
- *mult_op* {(, id, number}

PREDICT

1. *program* $\longrightarrow$ *stmt_list* $$ {id, read, write, $$}
2. *stmt_list* $\longrightarrow$ *stmt stmt_list* {id, read, write}
3. *stmt_list* $\longrightarrow$ ϵ {$$}
4. *stmt* $\longrightarrow$ id := *expr* {id}
5. *stmt* $\longrightarrow$ read id {read}
6. *stmt* $\longrightarrow$ write *expr* {write}
7. *expr* $\longrightarrow$ *term term_tail* {(, id, number}
8. *term_tail* $\longrightarrow$ *add_op term term_tail* {+, -}
9. *term_tail* $\longrightarrow$ ϵ {), id, read, write, $$}
10. *term* $\longrightarrow$ *factor factor_tail* {(, id, number}
11. *factor_tail* $\longrightarrow$ *mult_op factor factor_tail* {*, /}
12. *factor_tail* $\longrightarrow$ ϵ {+, -,), id, read, write, $$}
13. *factor* $\longrightarrow$ (*expr*) {(}
14. *factor* $\longrightarrow$ id {id}
15. *factor* $\longrightarrow$ number {number}
16. *add_op* $\longrightarrow$ + {+}
17. *add_op* $\longrightarrow$ - {-}
18. *mult_op* $\longrightarrow$ * {*}
19. *mult_op* $\longrightarrow$ / {/}

Figure 2.22: FIRST, FOLLOW, and PREDICT **sets for the calculator language.**

LL Parsing

- If any token belongs to the predict set of more than one production with the same LHS, then the grammar is not LL(1)
- A conflict can arise because
  - the same token can begin more than one RHS
  - it can begin one RHS and can also appear *after* the LHS in some valid program, and one possible RHS is ε