Lecture 5: LL(1) parsing

David Hovemeyer

September 15, 2021

601.428/628 Compilers and Interpreters

Today

- LL(1) parsing

Top-down parsing

- A *top-down* parsing algorithm attempts to derive an input string by starting with the start symbol, and applying productions (and consuming terminal symbols from the input string) until no nonterminals remain in the working string
- Using the lexer for lookahead helps the parser choose productions correctly
- Recursive descent is a top-down parsing technique
- LL(1) is a *generalized* top-down parsing technique
- Given a suitable context-free grammar, an LL(1) parser can be *generated*

Parser generators

- A *parser generator* takes a context-free grammar as input, and generates code for a parser
- The context-free grammar can be augmented with *semantic actions* to be carried out as productions are chosen
  - E.g., the semantic actions could build a parse tree or AST
- Advantage to using a parser generator: less work!
  - Also, greater confidence in correctness of parser
- Disadvantage to using a parser generator: less control (especially for generating meaningful error messages)

LL(1)

LL(1) is a simple table-driven top-down parsing algorithm

Requirements:

- Grammar has no left recursion
- For each nonterminal symbol A, FIRST^+ sets of all productions on A are disjoint (explanation soon)

FIRST sets

For a symbol K, FIRST(K) is the set of all terminal symbols which could begin an expansion of K

- Also, FIRST(K) contains ϵ if K can expand to the empty string

Example grammar: $A \rightarrow a$ (A is the start symbol)

$A \rightarrow B$

$A \rightarrow C$

$B \rightarrow b$

$B \rightarrow \epsilon$

$C \rightarrow c\ e$

$C \rightarrow d\ e$

$\text{FIRST}(A) = \{a, b, c, d, \epsilon\}$

Computing FIRST set

The FIRST set of any terminal symbol t is trivially $\{t\}$

For any nonterminal A, FIRST(A) can be computed as follows:

For each production of the form A $\rightarrow s_1\ s_2\ s_3\ \ldots\ s_n$:

- If s_1 is a terminal symbol, s_1 is in FIRST(A)
- If s_1 is a nonterminal, then all symbols in FIRST(s_1) are also in FIRST(A)
- If for some i in $1\,..\,n$, all symbols $s_1\,..\,s_{i-1}$ are nonterminals which can expand to ϵ, then all symbols in FIRST(s_i) are in FIRST(A)
- If each $s_1\,..\,s_n$ are nonterminals which can expand to ϵ, then ϵ is in FIRST(A)

Generalizing FIRST to string of symbols

If β is a string of symbols s_1, s_2, ..., s_n

Then FIRST(β) is the union of FIRST(s_1), FIRST(s_2), ..., FIRST(s_i) where for all j such that $1 \leq j < i$, $\epsilon \in$ FIRST(s_j)

- Except: $\epsilon \in$ FIRST(β) if and only if $\epsilon \in$ FIRST(s_k) for all $1 \leq k \leq n$

FOLLOW sets

For a nonterminal K, FOLLOW(K) is the set of terminal symbols which could follow an expansion of K

- Useful for knowing when it is appropriate to apply an epsilon production

Example grammar:

$A \rightarrow a\ B\ c$ (A is the start symbol)
$A \rightarrow C$
$C \rightarrow d\ B\ G\ f$
$B \rightarrow g$
$G \rightarrow h$
$G \rightarrow \epsilon$

FOLLOW(A) is { *eof* }

FOLLOW(B) is { c, f, h }

Note: *eof* is a special "end of file" token

Computing FOLLOW set

If S is the start symbol, then *eof* is in FOLLOW(S).

If a production A $\rightarrow \alpha$ B β exists, then all symbols in FIRST(β) except ϵ are in FOLLOW(B).

If either

- a production A $\rightarrow \alpha$ B exists, or
- a production A $\rightarrow \alpha$ B β exists, and FIRST(β) contains ϵ

then all symbols in FOLLOW(A) are in FOLLOW(B)

Example FIRST and FOLLOW sets

Let's see the construction of FIRST and FOLLOW sets in action:

Grammar (start symbol is E):

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E'$$
$$E' \rightarrow -\ T\ E'$$
$$E' \rightarrow \epsilon$$

$$T \rightarrow F\ T'$$
$$T' \rightarrow *\ F\ T'$$
$$T' \rightarrow /\ F\ T'$$
$$T' \rightarrow \epsilon$$
$$F \rightarrow i$$
$$F \rightarrow n$$

(Note that we removed the A nonterminal and its productions, since they would cause the grammar to be unsuitable for LL(1) parsing)

Example FIRST and FOLLOW sets

Grammar (start symbol is E):

E $\rightarrow$ T E'
E' $\rightarrow$ + T E'
E' $\rightarrow$ - T E'
E' $\rightarrow \epsilon$
T $\rightarrow$ F T'
T' $\rightarrow$ * F T'
T' $\rightarrow$ / F T'
T' $\rightarrow \epsilon$
F $\rightarrow$ i
F $\rightarrow$ n

Symbol	FIRST	FOLLOW
+		
-		
*		
/		
i		
n		
E		
E'		
T		
T'		
F		

Example FIRST and FOLLOW sets

Grammar (start symbol is E):

E $\rightarrow$ T E'
E' $\rightarrow$ + T E'
E' $\rightarrow$ - T E'
E' $\rightarrow \epsilon$
T $\rightarrow$ F T'
T' $\rightarrow$ * F T'
T' $\rightarrow$ / F T'
T' $\rightarrow \epsilon$
F $\rightarrow$ i
F $\rightarrow$ n

Symbol	FIRST	FOLLOW
+	{ + }	—
-	{ - }	—
*	{ * }	—
/	{ / }	—
i	{ i }	—
n	{ n }	—
E	{ i, n }	{ *eof* }
E'	{ +, -, ϵ }	{ *eof* }
T	{ i, n }	{ +, -, *eof* }
T'	{ *, /, ϵ }	{ +, -, *eof* }
F	{ i, n }	{ *, /, +, -, *eof* }

FIRST$^+$ sets

For a production A $\rightarrow \beta$, FIRST$^+$(A $\rightarrow \beta$) is

- FIRST(β) if $\epsilon \notin$ FIRST(β)
- FIRST(β) $\cup$ FOLLOW(A) otherwise

Production	FIRST$^+$ set
E $\rightarrow$ T E'	
E' $\rightarrow$ + T E'	
E' $\rightarrow$ - T E'	
E' $\rightarrow \epsilon$	
T $\rightarrow$ F T'	
T' $\rightarrow$ * F T'	
T' $\rightarrow$ / F T'	
T' $\rightarrow \epsilon$	
F $\rightarrow$ i	
F $\rightarrow$ n	

Symbol	FIRST	FOLLOW
+	{ + }	—
-	{ - }	—
*	{ * }	—
/	{ / }	—
i	{ i }	—
n	{ n }	—
E	{ i, n }	{ *eof* }
E'	{ +, -, ϵ }	{ *eof* }
T	{ i, n }	{ +, -, *eof* }
T'	{ *, /, ϵ }	{ +, -, *eof* }
F	{ i, n }	{ *, /, +, -, *eof* }

FIRST$^+$ sets

For a production A $\rightarrow \beta$, FIRST$^+$(A $\rightarrow \beta$) is

- FIRST(β) if $\epsilon \notin$ FIRST(β)
- FIRST(β) $\cup$ FOLLOW(A) otherwise

Production	FIRST$^+$ set
E $\rightarrow$ T E'	{ i, n }
E' $\rightarrow$ + T E'	{ + }
E' $\rightarrow$ - T E'	{ - }
E' $\rightarrow \epsilon$	{ ϵ, *eof* }
T $\rightarrow$ F T'	{ i, n }
T' $\rightarrow$ * F T'	{ * }
T' $\rightarrow$ / F T'	{ / }
T' $\rightarrow \epsilon$	{ ϵ, +, -, *eof* }
F $\rightarrow$ i	{ i }
F $\rightarrow$ n	{ n }

Symbol	FIRST	FOLLOW
+	{ + }	—
-	{ - }	—
*	{ * }	—
/	{ / }	—
i	{ i }	—
n	{ n }	—
E	{ i, n }	{ *eof* }
E'	{ +, -, ϵ }	{ *eof* }
T	{ i, n }	{ +, -, *eof* }
T'	{ *, /, ϵ }	{ +, -, *eof* }
F	{ i, n }	{ *, /, +, -, *eof* }

FIRST$^+$ sets intuitively

The FIRST$^+$ set for a production $A \to \beta$ tells us:

- If we want to expand an occurrence of A,
- What lookahead tokens indicate that $A \to \beta$ is the right production to apply

LL(1) parse tables

An LL(1) parse table indicates, for each nonterminal in a grammar, what production to apply based on what the next input token is

- Rows: nonterminal symbols
- Columns: terminal symbols and *eof*
- Entries: contain either a production number, or "invalid"

Building LL(1) parse table:

Mark all entries as "invalid"
For each production numbered p of the form $A \rightarrow \beta$
    For each terminal $t \in \text{FIRST}^+(A)$
        Put p in row A, column t
    If *eof* $\in \text{FIRST}^+(A)$
        Put p in row A, column *eof*

LL(1) parse table example

	Production	FIRST$^+$ set
(1)	E $\rightarrow$ T E'	{ i, n }
(2)	E' $\rightarrow$ + T E'	{ + }
(3)	E' $\rightarrow$ - T E'	{ - }
(4)	E' $\rightarrow \epsilon$	{ ϵ, *eof* }
(5)	T $\rightarrow$ F T'	{ i, n }
(6)	T' $\rightarrow$ * F T'	{ * }
(7)	T' $\rightarrow$ / F T'	{ / }
(8)	T' $\rightarrow \epsilon$	{ ϵ, +, -, *eof* }
(9)	F $\rightarrow$ i	{ i }
(10)	F $\rightarrow$ n	{ n }

	+	-	*	/	i	n	*eof*
E							
E'							
T							
T'							
F							

LL(1) parse table example

	Production	FIRST$^+$ set
(1)	E $\rightarrow$ T E'	{ i, n }
(2)	E' $\rightarrow$ + T E'	{ + }
(3)	E' $\rightarrow$ - T E'	{ - }
(4)	E' $\rightarrow \epsilon$	{ ϵ, *eof* }
(5)	T $\rightarrow$ F T'	{ i, n }
(6)	T' $\rightarrow$ * F T'	{ * }
(7)	T' $\rightarrow$ / F T'	{ / }
(8)	T' $\rightarrow \epsilon$	{ ϵ, +, -, *eof* }
(9)	F $\rightarrow$ i	{ i }
(10)	F $\rightarrow$ n	{ n }

	+	-	*	/	i	n	*eof*
E	–	–	–	–	1	1	–
E'	2	3	–	–	–	–	4
T	–	–	–	–	5	5	–
T'	8	8	6	7	–	–	8
F	–	–	–	–	9	10	–

LL(1) parsing

- Data structures are:
  - Sequence of terminal symbols (tokens)
  - Stack of symbols
- Start by pushing *eof* followed by the (nonterminal) start symbol

LL(1) parsing (continued)

- Repeatedly:
  - Inspect top of stack
  - If a terminal symbol, consume same symbol from input string and pop it from the stack (error if next input string symbol doesn't match)
  - If *eof*, make sure we're at end of input, if so, done (otherwise error)
  - If a nonterminal symbol:
    - Based on next input symbol, choose production number from table (error if entry is "invalid")
    - Pop nonterminal from stack
    - For each symbol on right hand side of chosen production from right to left, push it onto the stack

Example parse

Rule	Stack	Input
–	*eof* E	$_{\wedge}$ n - i / n

(1)	E $\rightarrow$ T E'
(2)	E' $\rightarrow$ + T E'
(3)	E' $\rightarrow$ - T E'
(4)	E' $\rightarrow \epsilon$
(5)	T $\rightarrow$ F T'
(6)	T' $\rightarrow$ * F T'
(7)	T' $\rightarrow$ / F T'
(8)	T' $\rightarrow \epsilon$
(9)	F $\rightarrow$ i
(10)	F $\rightarrow$ n

Example parse

Rule	Stack	Input
–	*eof* E	$_\wedge$ n - i / n
1	*eof* E' T	$_\wedge$ n - i / n
5	*eof* E' T' F	$_\wedge$ n - i / n
10	*eof* E' T' n	$_\wedge$ n - i / n
→	*eof* E' T'	n $_\wedge$ - i / n
8	*eof* E'	n $_\wedge$ - i / n
3	*eof* E' T' -	n $_\wedge$ - i / n
→	*eof* E' T'	n - $_\wedge$ i / n
5	*eof* E' T' F	n - $_\wedge$ i / n
9	*eof* E' T' i	n - $_\wedge$ i / n
→	*eof* E' T'	n - i $_\wedge$ / n
7	*eof* E' T' F /	n - i $_\wedge$ / n
→	*eof* E' T' F	n - i / $_\wedge$ n
10	*eof* E' T' n	n - i / $_\wedge$ n
→	*eof* E' T'	n - i / n $_\wedge$
8	*eof* E'	n - i / n $_\wedge$
4	*eof*	n - i / n $_\wedge$

(1)	E → T E'
(2)	E' → + T E'
(3)	E' → - T E'
(4)	E' → ϵ
(5)	T → F T'
(6)	T' → * F T'
(7)	T' → / F T'
(8)	T' → ϵ
(9)	F → i
(10)	F → n

Thoughts about LL(1)

Is LL(1) a "good" parsing algorithm?

- Inability to handle grammars with left recursion is annoying
- 1 token of lookahead isn't sufficient for some grammars
  - For example, in our original grammar, we had productions $\boxed{A \to i = A}$ and $\boxed{A \to E}$
  - FIRST^+ sets of both productions will contain $\boxed{i}$ and $\boxed{n}$, so they will conflict in the LL(1) parse table
  - A recursive descent parser could just look ahead by 2 tokens
  - LL(k) parsing is a generalization of LL(1) to allow greater lookahead