HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 5: LL Parsing

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Announcements

- Proj 1 out
  - Due Thursday Sept 20 (2 days away)
- Proj 2 out
  - Due Thursday Oct 4 (16 days away)

Academic Integrity

- Kevin Stephen, Harvard College Honor Council

Today

- LL Parsing
  - Nullable, First, Follow sets
  - Constructing an LL parsing table

LL(k) Parsing

- Our parser combinators backtrack
  - `alt p1 p2 = fun cs -> (p1 cs) @ (p2 c2)` runs `p1` on `cs`, then backs up and runs `p2` on same input!
  - Inefficient! Tries all possible parses
- Could we somehow know which production to use?
- Basic idea: look at the next k symbols to predict whether we want `p1` or `p2`
- How do we predict which production to use?

FIRST Sets

- Given string γ of terminal and non-terminal symbols FIRST(γ) is set of all terminal symbols that can start a string derived from γ

$E \rightarrow T\ E'$	$T \rightarrow F\ T'$	$F \rightarrow$ `id`
$E' \rightarrow$ `+` $T\ E'$	$T' \rightarrow$ `*` $F\ T'$	$F \rightarrow$ `num`
$E' \rightarrow$ `-` $T\ E'$	$T' \rightarrow$ `/` $F\ T'$	$F \rightarrow$ `(` E `)`
$E' \rightarrow$	$T' \rightarrow$	

- E.g., FIRST($F\ T'$) = { **`id`**, **`num`**, **`(`** }
- We can use FIRST sets to determine which production to use!
  - Given nonterminal X, and all its productions

    $X \rightarrow \gamma_1$, $X \rightarrow \gamma_2$, … ,$X \rightarrow \gamma_n$,

    if FIRST(γ_1), …, FIRST(γ_n) all mutually disjoint,

    then next character tells us which production to use

Computing FIRST Sets

- See Appel for algorithm. Intuition here...
- Consider FIRST($X\ Y\ Z$)
- How do compute it? Do we just need to know FIRST(X)?
- What if X can derive the empty string?
- Then FIRST(Y) $\subseteq$ FIRST($X\ Y\ Z$)
- What if Y can also derive the empty string?
- Then FIRST(Z) $\subseteq$ FIRST($X\ Y\ Z$)

Computing FIRST, FOLLOW and Nullable

- To compute FIRST sets, we need to compute whether nonterminals can produce empty string
- FIRST(γ) = all terminal symbols that can start a string derived from γ
- Nullable(*X*) = true iff X can derive the empty string
- We will also compute:
  FOLLOW(*X*) = all terminals that can immediately follow *X*
  - i.e., $t \in$ FOLLOW(*X*) if there is a derivation containing *Xt*
- Algorithm iterates computing these until fix point reached
- **Note:** knowing nullable(*X*) and FIRST(*X*) for all non-terminals *X* allows us to compute nullable(γ) and FIRST(γ) for arbitrary strings of symbols γ

Example

$S \rightarrow E$ `eof`

$E \rightarrow T\ E'$

$E' \rightarrow$ `+` $T\ E'$

$E' \rightarrow$ `-` $T\ E'$

$E' \rightarrow$

$T \rightarrow F\ T'$

$T' \rightarrow$ `*` $F\ T'$

$T' \rightarrow$ `/` $F\ T'$

$T' \rightarrow$

$F \rightarrow$ `id`

$F \rightarrow$ `num`

$F \rightarrow$ `(` E `)`

	nullable	FIRST	FOLLOW
S	$\bot$		
E	$\bot$		
E'	$\top$		
T	$\bot$		
T'	$\top$		
F	$\bot$		

X is nullable if there is a production $X \rightarrow \gamma$ where γ is empty, or γ is all nullable nonterminals

T' and E' are nullable!

And, we've finished nullable. Why?

Example

$S \rightarrow E$ eof

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

$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T'$
$T' \rightarrow /\ F\ T'$
$T' \rightarrow$

$F \rightarrow$ id
$F \rightarrow$ num
$F \rightarrow$ (E)

	nullable	FIRST	FOLLOW
S	$\bot$		
E	$\bot$		
E'	$\top$	+ -	
T	$\bot$		
T'	$\top$	* /	
F	$\bot$	id num (	

Given production $X \rightarrow t\gamma$, $t \in$ FIRST(X)

Example

$S \rightarrow E$ `eof`

$E \rightarrow T\,E'$

$E' \rightarrow + \,T\,E'$

$E' \rightarrow - \,T\,E'$

$E' \rightarrow$

$T \rightarrow F\,T'$

$T' \rightarrow * \,F\,T'$

$T' \rightarrow / \,F\,T'$

$T' \rightarrow$

$F \rightarrow$ `id`

$F \rightarrow$ `num`

$F \rightarrow$ `(` E `)`

	nullable	FIRST	FOLLOW
S	⊥	id num (	
E	⊥	id num (	
E'	⊤	+ -	
T	⊥	id num (	
T'	⊤	* /	
F	⊥	id num (	

Given production $X \rightarrow \gamma Y \sigma$,

if nullable(γ) then FIRST(Y) $\subseteq$ FIRST(X)

Repeat until no more changes...

Example

$S \rightarrow E$ `eof`

$E \rightarrow T\ E'$

$E' \rightarrow + \ T\ E'$

$E' \rightarrow -\ T\ E'$

$E' \rightarrow$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'$

$T' \rightarrow /\ F\ T'$

$T' \rightarrow$

$F \rightarrow$ `id`

$F \rightarrow$ `num`

$F \rightarrow$ `(` E `)`

	nullable	FIRST	FOLLOW
S	$\bot$	id num (	
E	$\bot$	id num (	eof)
E'	$\top$	+ -	eof)
T	$\bot$	id num (	+ - eof)
T'	$\top$	* /	+ - eof)
F	$\bot$	id num (	* / + - eof)

Given production $X \rightarrow \gamma Z \delta \sigma$

FIRST(δ) $\subseteq$ FOLLOW(Z)

and if δ is nullable then FIRST(σ) $\subseteq$ FOLLOW(Z)

and if $\delta\sigma$ is nullable then FOLLOW(X) $\subseteq$ FOLLOW(Z)

Predictive Parsing Table

- Make **predictive parsing table** with rows nonterminals, columns terminals
  - Table entries are productions
  - When parsing nonterminal *X*, and next token is *t*, entry for *X* and *t* will tell us which production to use

Example

$S \to E$ eof

$E \to T\,E'$

$E' \to +\,T\,E'$

$E' \to -\,T\,E'$

$E' \to$

$T \to F\,T'$

$T' \to *\,F\,T'$

$T' \to /\,F\,T'$

$T' \to$

$F \to$ id

$F \to$ num

$F \to (E)$

	nullable	FIRST	FOLLOW
S	⊥	id num (	
E	⊥	id num (	eof)
E'	⊤	+ -	eof)
T	⊥	id num (	+ - eof)
T'	⊤	* /	+ - eof)
F	⊥	id num (	* / + - eof)

	id	num	+	-	*	/	(	)	eof
S	$S \to E$ eof	$S \to E$ eof					$S \to E$ eof		
E	$E \to T\,E'$	$E \to T\,E'$					$E \to T\,E'$		
E'			$E' \to +\,T\,E'$	$E' \to -\,T\,E'$				$E' \to$	$E' \to$
T	$T \to F\,T'$	$T \to F\,T'$					$T \to F\,T'$		
T'			$T' \to$	$T' \to$	$T' \to *\,F\,T'$	$T' \to /\,F\,T'$		$T' \to$	$T' \to$
F	$F \to$ id	$F \to$ num					$F \to (E)$		

For $X \to \gamma$, add $X \to \gamma$ to row X column t for all $t \in \text{FIRST}(\gamma)$

For $X \to \gamma$, if γ is nullable, add $X \to \gamma$ to row X column t for all $t \in \text{FOLLOW}(X)$

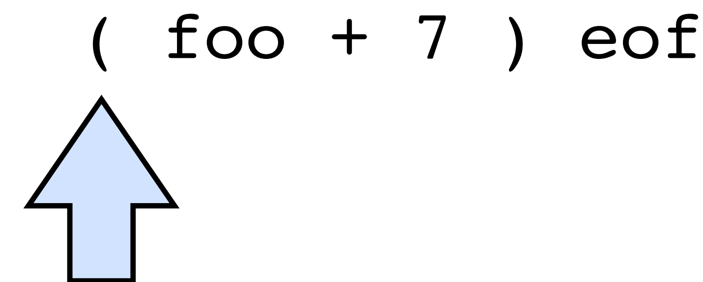
Example

	id	num	+	-	*	/	(	)	eof
S	$S \to E$ eof	$S \to E$ eof					$S \to E$ eof		
E	$E \to T\,E'$	$E \to T\,E'$					$E \to T\,E'$		
E'			$E' \to + \,T\,E'$	$E' \to - \,T\,E'$				$E' \to$	$E' \to$
T	$T \to F\,T'$	$T \to F\,T'$					$T \to F\,T'$		
T'			$T' \to$	$T' \to$	$T' \to * \,F\,T'$	$T' \to / \,F\,T'$		$T' \to$	$T' \to$
F	$F \to$ id	$F \to$ num					$F \to (E)$		

- If each cell contains at most one production, parsing is predictive!
  - Table tells us exactly which production to apply

Example

	id	num	+	-	*	/	(	)	eof
S	$S \to E$ eof	$S \to E$ eof					$S \to E$ eof		
E	$E \to T\ E'$	$E \to T\ E'$					$E \to T\ E'$		
E'			$E' \to + T\ E'$	$E' \to - T\ E'$				$E' \to$	$E' \to$
T	$T \to F\ T'$	$T \to F\ T'$					$T \to F\ T'$		
T'			$T' \to$	$T' \to$	$T' \to * F\ T'$	$T' \to / F\ T'$		$T' \to$	$T' \to$
F	$F \to$ id	$F \to$ num					$F \to (E)$		



S — Parse S, next token is (, use $S \to E$ eof

E eof — Parse E, next token is (, use $E \to T\ E'$

$T\ E'$ eof — Parse T, next token is (, use $T \to F\ T'$

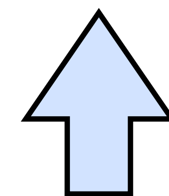
$F\ T'\ E'$ eof — Parse F, next token is (, use $F \to (E)$

$(E)\ T'\ E'$ eof

Example

	id	num	+	-	*	/	(	)	eof
S	$S \to E$ eof	$S \to E$ eof					$S \to E$ eof		
E	$E \to T\ E'$	$E \to T\ E'$					$E \to T\ E'$		
E'			$E' \to + T\ E'$	$E' \to - T\ E'$				$E' \to$	$E' \to$
T	$T \to F\ T'$	$T \to F\ T'$					$T \to F\ T'$		
T'			$T' \to$	$T' \to$	$T' \to * F\ T'$	$T' \to / F\ T'$		$T' \to$	$T' \to$
F	$F \to$ id	$F \to$ num					$F \to (E)$		

`( foo + 7 ) eof`

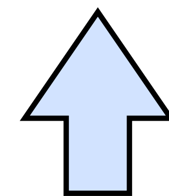


S	Parse S, next token is `(`, use $S \to E$ eof
E eof	Parse E, next token is `(`, use $E \to T\ E'$
$T\ E'$ eof	Parse T, next token is `(`, use $T \to F\ T'$
$F\ T'\ E'$ eof	Parse F, next token is `(`, use $F \to (E)$
$(E)\ T'\ E'$ eof	Parse E, next token is id, use $E \to T\ E'$
$(T\ E')\ T'\ E'$ eof	Parse T, next token is id, use $T \to F\ T'$
$(F\ T'\ E')\ T'\ E'$ eof	Parse F, next token is id, use $F \to$ id

Example

	id	num	+	-	*	/	(	)	eof
S	$S \rightarrow E$ eof	$S \rightarrow E$ eof					$S \rightarrow E$ eof		
E	$E \rightarrow T\,E'$	$E \rightarrow T\,E'$					$E \rightarrow T\,E'$		
E'			$E' \rightarrow +\,T\,E'$	$E' \rightarrow -\,T\,E'$				$E' \rightarrow$	$E' \rightarrow$
T	$T \rightarrow F\,T'$	$T \rightarrow F\,T'$					$T \rightarrow F\,T'$		
T'			$T' \rightarrow$	$T' \rightarrow$	$T' \rightarrow *\,F\,T'$	$T' \rightarrow /\,F\,T'$		$T' \rightarrow$	$T' \rightarrow$
F	$F \rightarrow$ id	$F \rightarrow$ num					$F \rightarrow (E)$		

`( foo + 7 ) eof`



($F\,T'\,E'$) $T'\,E'$ eof — Parse F, next token is id, use $F \rightarrow$ id

(id $T'\,E'$) $T'\,E'$ eof — Parse T', next token is +, use $T' \rightarrow$

(id E') $T'\,E'$ eof — Parse E', next token is +, use $E' \rightarrow +\,T\,E'$

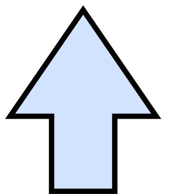
(id + $T\,E'$) $T'\,E'$ eof — Parse T, next token is num, use $T \rightarrow F\,T'$

(id + $F\,T'\,E'$) $T'\,E'$ eof — Parse F, next token is num, use $F \rightarrow$ num

Example

	id	num	+	-	*	/	(	)	eof
S	$S \to E$ eof	$S \to E$ eof					$S \to E$ eof		
E	$E \to T\,E'$	$E \to T\,E'$					$E \to T\,E'$		
E'			$E' \to + \, T\,E'$	$E' \to - \, T\,E'$				$E' \to$	$E' \to$
T	$T \to F\,T'$	$T \to F\,T'$					$T \to F\,T'$		
T'			$T' \to$	$T' \to$	$T' \to * \, F\,T'$	$T' \to / \, F\,T'$		$T' \to$	$T' \to$
F	$F \to$ id	$F \to$ num					$F \to (E)$		

`( foo + 7 ) eof`



(id + $F\,T'\,E'$) $T'\,E'$ eof — Parse F, next token is num, use $F \to$ num

(id + num $T'\,E'$) $T'\,E'$ eof — Parse T', next token is), use $T' \to$

(id + num E') $T'\,E'$ eof — Parse E', next token is), use $E' \to$

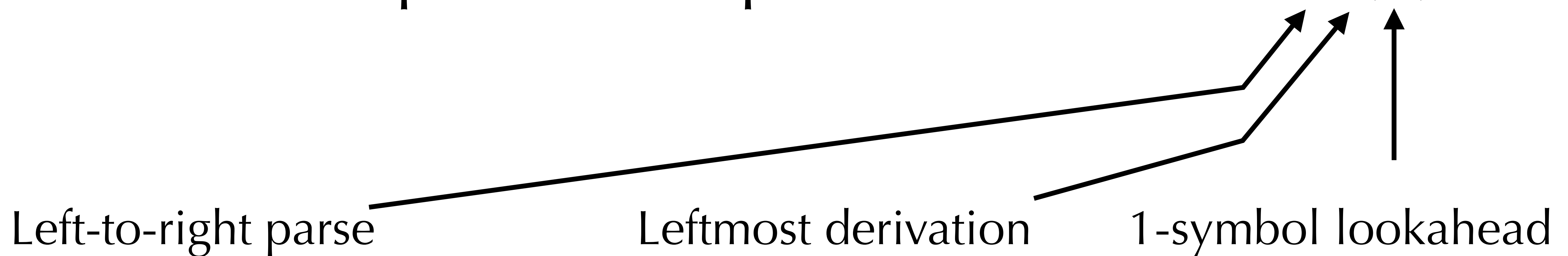
(id + num) $T'\,E'$ eof — Parse T', next token is eof, use $T' \to$

(id + num) E' eof — Parse E', next token is eof, use $E' \to$

(id + num) eof

LL(1), LL(*k*), LL(*)

- Grammars whose predictive parsing table contain at most one production per cell are called LL(1)



i.e., go through token stream from left to right.
(Almost all parsers do this)

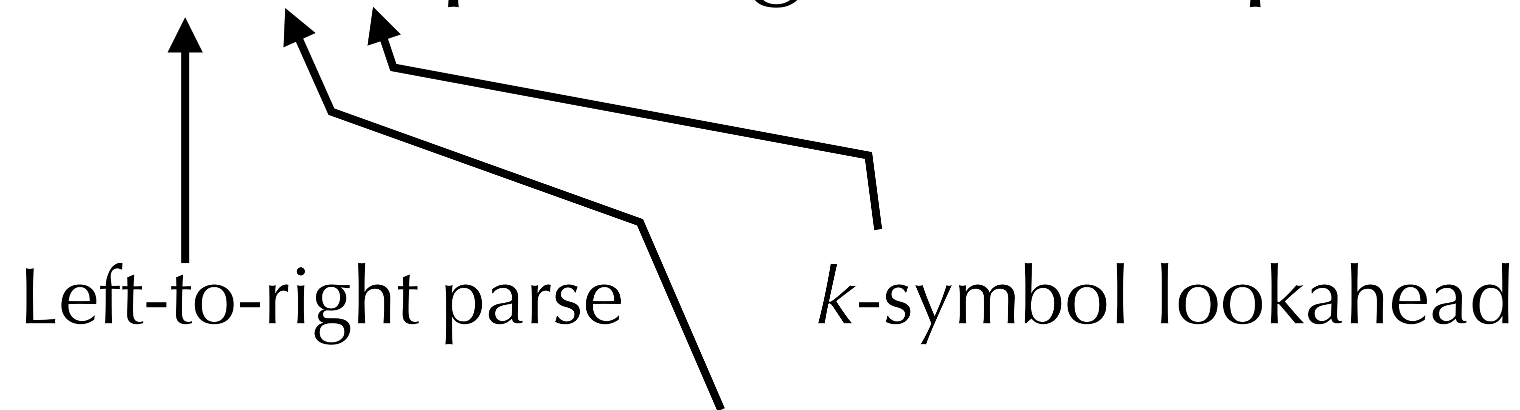
Derivation expands the leftmost non-terminal

LL(1), LL(*k*), LL(*)

- Grammars whose predictive parsing table contain at most one production per cell are called LL(1)
- Can be generalized to LL(2), LL(3), etc.
  - Columns of predictive parsing table have k tokens
  - FIRST(X) generalized to FIRST-k(X)
- An LL(*) grammar can determine next production using finite (but maybe unbounded) lookahead
- An ambiguous grammar is not LL(k) for any k, or even LL(*)
  - Why?

LR(k)

- What if grammar is unambiguous but not LL(k)?
- LR(k) parsing is more powerful technique



Rightmost derivation

Derivation expands the rightmost non-terminal

(Constructs derivation in reverse order!)