LL(1) Parsing, pt. 1

CS 440: Programming Languages and Translators, Spring 2020

A. Parsing Using a Context-Free Grammar and Push-Down Automaton

- Regular expressions, finite-state automata, and regular grammars work with the same set of languages.
- Similarly, context-free grammars and "push-down automata" both relate to context-free languages:
  - A context-free language is one generated by a context-free grammar. Context-free languages are also the ones recognizable using push-down automata.
  - A **Push-Down Automaton (PDA)** is a finite state machine augmented with a stack.
    - The state transition function maps a state, input symbol, and top-of-stack symbol to a new state and a stack operation (pop the stack, push a symbol onto the stack, or do nothing).
  - As with finite-state machines, there are deterministic and non-deterministic PDAs, but the non-deterministic PDAs can do things deterministic PDAs can't.
    - The set-of-states transformation used to map nondeterministic finite automata to equivalent deterministic finite automata doesn't work with PDAs.
    - (We can keep track of all the possible states we might be in, but we can't form the set of all possible stack configurations we might be in.)
  - For parsing, we much prefer the **deterministic PDAs (DPDAs)** because they don't require backtracking to execute the way a nondeterministic PDA can. A **deterministic CFL (DCFL)** is a language that can be parsed using a DPDA, and we use DCFLs to describe programming languages.
  - The simplest kind of DPDA has just one automaton state: Given the next input symbol and the top of the stack, gobble the input symbol and perform a stack operation.
  - It turns out that a language is LL(1) if it can be parsed by a DPDA with just one state.

B. LL(1) Parsing, the Predict table, and First sets

- We've been parsing LL(1) languages using recursive descent. The state of a recursive descent parser can be described using the current input symbol and a stack (which tracks the sequence of rule applications that got us where we're at). We have a nonterminal A that we're trying to expand and we look at the next input symbol to figure out which of the rules for A to apply ($A \to \alpha_1$? $A \to \alpha_2$? etc?)
  - As an example if we have rules $A \to \mathtt{x} \ldots$ and $A \to \mathtt{y} \ldots$ and the next input symbol is `x`, then we go with the $A \to \mathtt{x} \ldots$ rule.
  - If no rules apply the next input symbol is `x` and there's no way for A to yield something that starts with x (i.e., $A \not\Rightarrow^* \mathtt{x} \ldots$) then we have a parse error. If > 1 rule applies, the language isn't LL(1).
- For a stack-based approach to LL(1) parsing, the "which rule do we apply?" decisions are stored in a **prediction table**: $\mathit{Predict}(A, \mathtt{x})$ = the set of rules that apply if the current nonterminal to expand is A and the next input symbol is `x`. If $\mathit{Predict}(A, \mathtt{x}) = \varnothing$, then we have a parse error. If $\mathit{Predict}(A, \mathtt{x})$ contains > 1 rule, then the language is not LL(1). (This is something we can check for when we build the prediction table.)

- The key to building the *Predict* table is the idea that for rule $A \to \alpha$, if from α we can continue to something that begins with terminal symbol `x`, then it makes sense to try expanding A using $A \to \alpha$ if the next input symbol is `x`.
- The set $First(\alpha)$ holds the answers to the question of what terminal symbols we can find if we start with α and look at all its possible derived yields.
- **Definition**: $First(\alpha) = \{\mathtt{x} \in T \mid \alpha \to^* \mathtt{x}\ \beta \text{ for some } \beta \in (V \mid T)^*)\}$. I.e., what sentential forms `x`… can we get to if we start from α?
  - For all β, $\{\mathtt{x}\} = First(\mathtt{x}\ \beta)$. I.e., if α begins with a terminal symbol `x`, then certainly $\mathtt{x} \in First(\alpha)$.
  - For all rules $A \to \alpha$, $First(A\ \beta) \supseteq First(\alpha\ \beta)$. I.e., if we want to know the *First* set for A, we certainly need the *First* sets for all rhs's of rules for A.
  - There's a complicating factor: If $A \to^* \mathtt{x}\ldots$, then for all β, $A\ \beta \to^* \mathtt{x}\ldots\ \beta$, so we don't need to know about $First(\beta)$, right? But if $A \to^* \varepsilon$, then $A\ \beta \to^* \varepsilon\ \beta$, so $First(A\ \beta) \supseteq First(\beta)$, so we do care about $First(\beta)$. So,
    - If $A \not\to^* \varepsilon$ then for all β, $First(A\ \beta) = First(A)$.
    - If $A \to^* \varepsilon$ then for all β, $First(A\ \beta) \supseteq First(\beta)$.
- **Definition 1**: For a nonterminal A and terminal `x`,
  - $Predict(A, \mathtt{x}) = \{\text{rule } A \to \alpha \mid \text{(For some } \beta), S \to^* \ldots A\ \beta \text{ and } \mathtt{x} \in First(A\ \beta)\}$
- If $A \to^* \varepsilon$ then in theory, we need to know $First(\beta)$ for all possible β (of which there are an infinite number). But actually, we only need to worry about the $A\ \beta$ we can get to from the start symbol. The situation is actually even better than that, but let's first look at the basic stack-based parsing algorithm and then go back to the details of calculating the *Predict* table using $First(A)$ and a helper function $Follow(A)$.

C. Stack-Based LL(1) Parsing using Predict

- First, so that we can distinguish between the first use of the start symbol and any subsequent use, we'll **modify the grammar** a bit:
  - 1: Add a fresh (unused) terminal symbol `$` used to signal end-of-input.
  - 2: Add a new start symbol S' and a rule $S' \to S$ `$` where S is the nominal start symbol.

Basic algorithm:

Concatenate `$` to end of input string and initialize the stack: push S' onto it.
while top of stack ≠ `$` {
  Let `x` be next character of remaining input.
  if `x` = `$`, parse error (input ended early)
  Pop top of stack
  if it's a terminal `y`
    if `x` = `y`, remove `x` from the input and continue loop **else** parse error
  else (the top of the stack was a nonterminal A)
    if $Predict(A, \mathtt{x})$ says error (it's ∅) then parse error (unexpected terminal `x`)
    else check $Predict(A, \mathtt{x})$ for an entry $A \to \alpha$ and push α onto the stack
end

if (loop has ended, top of stack is `$` and) next the input character is `$`

We parsed successfully!

else parse error (leftover input)

D. Example of Using Prediction table for LL(1) Parse

- See Example 1.

Example 1: Grammar of Balanced Parens plus variables `x` and `y`

Terminals = `( ) x y $`

Rules

Rule Nbr	Rule
0	$S' \to S$ `$`
1	$S \to P\ S$
2	$S \to$ `x`
3	$P \to$ \(S \)
4	$P \to$ `y`

Predict(A, s) = Rule number (blank entry = error)

Nonterminal	(	)	x	y
S'	0: $S' \to S$ `$`		2: $S \to$ `x`	1: $S \to P\ S$
S	1: $S \to P\ S$		2: $S \to$ `x`	1: $S \to P\ S$
P	3: $P \to$ \(S \)			4: $P \to$ `y`

Leftmost Derivation of `( y x ) ( x ) x $`

S' `$`

→ S `$`

→ $P\ S$ `$`

→ (S) S `$`

→ ($P\ S$) S `$`

→ (y S) S `$`

→ (y x) $P\ S$ `$`

→ (y x) (S) S `$`

→ (y x) (x) S `$`

→ (y x) (x) x `$`

Trace of Parse of `( y x ) ( x ) x $`

Stack (Top to Left)	Rule nbr or = (terminals)	Input
S'	0	(y x) (x) x $
S $	1	(y x) (x) x $
P S $	3	(y x) (x) x $
(S) S $	=	(y x) (x) x $
S) S $	1	y x) (x) x $
P S) S $	4	y x) (x) x $
y S) S $	=	y x) (x) x $
S) S $	2	x) (x) x $
x) S $	=	x) (x) x $
) S $	=	) (x) x $
S $	1	(x) x $
P S $	3	(x) x $
(S) S $	=	(x) x $
S) S $	2	x) x $
x) S $	=	x) x $
) S $	=	) x $
S $	2	x $
x $	=	x $
$	=	$

NT	(	)	x	y
S'	0: $S' \to S$ $		2: $S \to$ x	1: $S \to P\ S$
S	1: $S \to P\ S$		2: $S \to$ x	1: $S \to P\ S$
P	3: $P \to$ \(S \)			4: $P \to$ y

(End of example 1)

Activity Questions for Lecture 13

1. Calculate the *First*(…) sets and the *Predict* table for a language without ε rules, specifically,

   $S \to A$ `$`

   $A \to$ `a` B

   $B \to C\ D$ `e`

   $C \to$ `c` $C \mid D$

   $D \to$ `d` D

2. Use the *Predict* table from question 1 and calculate a trace of the parse of `a d a d c e e $` using the parsing algorithm.

3. Calculate the *First*(…) sets and the *Predict* table for the following language

   $S' \to S$ `$`

   $S \to$ `a` S `b` S

   $S \to$ `c`

4. Use the *Predict* table from question 2 and calculate a trace of the parse of `a a c b a c b c b c $`.

Solutions to Activity Questions for Lecture 13

1. (Calculate First sets and LL prediction table for grammar with no ε rules.)

Here's the reasoning behind the First set contents with each rule and its implications.

$S \to A$ \$	First(S) ⊇ First(A)
$A \to$ a B	a ∈ First(A)
$B \to C\ D$ e	First(B) ⊇ First(C)
$C \to$ c | D	c ∈ First(C); First(C) ⊇ First(D)
$D \to$ d	d ∈ First(C)
$D \to A$	*First*(D) ⊇ *First*(A)

So we get

First(S) = {a}

First(A) = {a}

First(B) = {c, d}

First(C) = {c, d}

First(D) = {a, d}

Prediction Table

Nonterminal	a	c	d	e
S	0: $S \to A$ \$			
A	1: $A \to$ a B			
B		2: $B \to C\ D$ e	2: $B \to C\ D$ e	
C		3: $C \to$ c	4: $C \to D$	5: $C \to D$
D	6: $D \to A$		5: $D \to$ d	

2. (Trace parse)

Trace of Parse of a d a d c e e \$

Stack (Top to Left)	Rule # / = term?	Input
S	0	a d a d c e e \$
A \$	1	a d a d c e e \$
a B \$	=	a d a d c e e \$
B \$	2	d a d c e e \$
$C\ D$ e \$	4	d a d c e e \$
$D\ D$ e \$	5	d a d c e e \$
d D e \$	=	d a d c e e \$
D e \$	6	a d c e e \$
A e \$	1	a d c e e \$
a B e \$	=	a d c e e \$
B e \$	2	d c e e \$
$C\ D$ e e \$	3	d c e e \$
c D e e \$	=	d c e e \$
D e e \$	5	d e e \$
d e e \$	=	d e e \$
e e \$	=	e e \$
e \$	=	e \$
\$	success!	\$

Rule nbr	Rule
0	$S \to A$ \$
1	$A \to$ a B
2	$B \to C\ D$ e
3	$C \to$ c
4	$C \to D$
5	$D \to$ d
6	$D \to A$

3. (First sets and Predict table for LL(1) grammar without ε rules)

Rule 0	$S' \to S$ \$	Implies $First(S) \subseteq First(S')$
Rule 1:	$S \to$ a S b S	a $\in First(S)$
Rule 2:	$S \to$ c	c $\in First(S)$
		So $First(S) = First(S') = \{a, c\}$

Predict table

Nonterminal	a	b	c	\$
S'	0: $S' \to S$ \$		0: $S' \to S$ \$	
S	1: $S \to$ a S b S		2: $S \to$ c	

4. (Calculate trace of a parse)

Trace of Parse of a a c b a c b c b c \$

Stack (Top to Left)	Rule # / = term?	Input
S'	0	a a c b a c b c b c \$
S \$	1	a a c b a c b c b c \$
a S b S \$	=	a a c b a c b c b c \$
S b S \$	1	a c b a c b c b c \$
a S b S b S \$	=	a c b a c b c b c \$
S b S b S \$	2	c b a c b c b c \$
c b S b S \$	=	c b a c b c b c \$
b S b S \$	=	b a c b c b c \$
S b S \$	1	a c b c b c \$
a S b S b S \$	=	a c b c b c \$
S b S b S \$	2	c b c b c \$
c b S b S \$	=	c b c b c \$
b S b S \$	=	b c b c \$
S b S \$	2	c b c \$
c b S \$	=	c b c \$
b S \$	=	b c \$
S \$		c \$
c \$	=	c \$
\$	success!	\$

