Programming Paradigms
Lecture 4: Syntax (Part 3)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2019/2020

Wake-up Exercise

What does the following Perl code print?

```
$b = 0;
sub foo {
  return $b;
}
sub bar {
  local $b = 1;
  return foo();
}
print bar();
```

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

What does the following Perl code print?

```
$b = 0;
sub foo {
  return $b;
}
sub bar {
  local $b = 1;
  return foo();
}
print bar();
```

Result: 1

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

What does the following Perl code print?

```
$b = 0;
sub foo {
  return $b;
}
sub bar {
  local $b = 1;
  return foo();
}
print bar();
```

Global variable that would be used with static scoping

Result: 1

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

What does the following Perl code print?

```
$b = 0;
sub foo {
  return $b;
}
sub bar {
  local $b = 1;
  return foo();
}
print bar();
```

- But: Perl has dynamic scoping
- Uses last encountered definition of `$b`

Result: 1

https://ilias3.uni-stuttgart.de/vote/0ZT9

Overview

- **Specifying syntax**
  - Regular expressions
  - Context-free grammars
- **Scanning**
- **Parsing**
  - Top-down parsing
  - Bottom-up parsing

How to Automate This?

- **To generate an LL(k) parser, need to predict which rule to apply**
- **Compute PREDICT sets for all productions, based on two helpers**
  - FIRST(N): What terminals come first when expanding non-terminal N?
  - FOLLOW(N): What terminals follow after non-terminal N?

PREDICT Sets

PREDICT set for a rule: Which terminals to look for in LL(1) parser

- If next input token is in PREDICT of rule, apply the rule
- Computing the PREDICT set for rule A $\rightarrow \alpha$:
  - If ϵ in FIRST(A):

    PREDICT(A $\rightarrow \alpha$) = (FIRST(α) - $\{ \epsilon \}$) $\cup$ FOLLOW(A)
  - Otherwise:

    PREDICT(A $\rightarrow \alpha$) = FIRST(α)

Example

Grammar:

S $\rightarrow$ `a` B

S $\rightarrow$ `b` C

B $\rightarrow$ `b` `b` C

C $\rightarrow$ `c` `c`

	FIRST	FOLLOW
S	a, b	EOF
B	b	EOF
C	c	EOF

Example

Grammar:

S → a B
S → b C
B → b b C
C → c c

PREDICT:

{ a }
{ b }
{ b }
{ c }

	FIRST	FOLLOW
S	a, b	EOF
B	b	EOF
C	c	EOF

Example

Grammar:

Grammar	PREDICT:
S → a B	{ a }
S → b C	{ b }
B → b b C	{ b }
C → c c	{ c }

	FIRST	FOLLOW
S	a, b	EOF
B	b	EOF
C	c	EOF

```
S() {
  if (inputToken == a)
    match(a); B();
  else if (inputToken == b)
    match(b); C();
  else error();
}
B() {
  if (inputToken == b)
    match(b); match(b); C();
  else error()
}
C() {
  if (inputToken == c)
    match(c); match(c);
  else error()
}
```

Example

Grammar:

Grammar	PREDICT:
$S \to$ a B	$\boxed{\{\ a\ \}}$
$S \to$ b C	{ b }
$B \to$ b b C	{ b }
$C \to$ c c	{ c }

	FIRST	FOLLOW
S	a, b	EOF
B	b	EOF
C	c	EOF

```
S() {
  if (inputToken == a)
    match(a); B();
  else if (inputToken == b)
    match(b); C();
  else error();
}
B() {
  if (inputToken == b)
    match(b); match(b); C();
  else error()
}
C() {
  if (inputToken == c)
    match(c); match(c);
  else error()
}
```

Example

Grammar:

		PREDICT:
S	$\rightarrow$ a B	{ a }
S	$\rightarrow$ b C	$\boxed{\{ b \}}$
B	$\rightarrow$ b b C	{ b }
C	$\rightarrow$ c c	{ c }

	FIRST	FOLLOW
S	a, b	EOF
B	b	EOF
C	c	EOF

```
S() {
   if (inputToken == a)
     match(a); B();
   else if (inputToken == b)
     match(b); C();
   else error();
}
B() {
   if (inputToken == b)
     match(b); match(b); C();
   else error()
}
C() {
   if (inputToken == c)
     match(c); match(c);
   else error()
}
```

Computing the Parse Table

Computing an LL(1) parse table

- Given: PREDICT set of each rule
- Table is a mapping M:

  $N \times T \rightarrow$ Production rule or error

- For all productions A $\rightarrow \alpha$ do
  - For each terminal t in PREDICT(A $\rightarrow \alpha$):

    M[A][t] = A $\rightarrow \alpha$

  - Every undefined table entry is an error

Example: Parse table

Non-term. \ Term.	a	b	c
S	1	2	–
B	–	3	–
C	–	–	4

1) $S \to a B$
2) $S \to b C$
3) $B \to b b C$
4) $C \to c c$

Table-based, Predictive Parsing

```
stack.push(EOF); stack.push(startSymbol);
nextToken = lookAhead();
repeat
  x = stack.pop();
  if x is terminal or EOF
    if x == nextToken
      nextToken = lookAhead()
    else error()
  else // x is non-terminal
    if M[x][nextToken] == x -> y1 y2 .. ym
      stack.push(ym); ...; stack.push(y1);
    else error()
until x is EOF
```

Table-based, Predictive Parsing

Parse stack: Prediction of what will be seen in the future

```
stack.push(EOF); stack.push(startSymbol);
nextToken = lookAhead();
repeat
  x = stack.pop();
  if x is terminal or EOF
    if x == nextToken
      nextToken = lookAhead()
    else error()
  else // x is non-terminal
    if M[x][nextToken] == x -> y1 y2 .. ym
      stack.push(ym); ...; stack.push(y1);
    else error()
until x is EOF
```

Table-based, Predictive Parsing

```
stack.push(EOF); stack.push(startSymbol);
nextToken = lookAhead();
repeat
  x = stack.pop();
  if x is terminal or EOF
    if x == nextToken
      nextToken = lookAhead()
    else error()
  else // x is non-terminal
    if M[x][nextToken] == x -> y1 y2 .. ym
      stack.push(ym); ...; stack.push(y1);
    else error()
until x is EOF
```

Read one token after another, always looking only one token ahead

Table-based, Predictive Parsing

Check if expected terminal is indeed the next token

```
stack.push(EOF); stack.push(startSymbol);
nextToken = lookAhead();
repeat
  x = stack.pop();
  if x is terminal or EOF
    if x == nextToken
      nextToken = lookAhead()
    else error()
  else // x is non-terminal
    if M[x][nextToken] == x -> y1 y2 .. ym
      stack.push(ym); ...; stack.push(y1);
    else error()
until x is EOF
```

Table-based, Predictive Parsing

```
stack.push(EOF); stack.push(startSymbol);
nextToken = lookAhead();
repeat
  x = stack.pop();
  if x is terminal or EOF
    if x == nextToken
      nextToken = lookAhead()
    else error()
  else // x is non-terminal
    if M[x][nextToken] == x -> y1 y2 .. ym
      stack.push(ym); ...; stack.push(y1);
    else error()
until x is EOF
```

Apply a production rule: Push right-hand side onto stack

Table-based, Predictive Parsing

```
stack.push(EOF); stack.push(startSymbol);
nextToken = lookAhead();
repeat
  x = stack.pop();
  if x is terminal or EOF
    if x == nextToken
      nextToken = lookAhead()
    else error()
  else // x is non-terminal
    if M[x][nextToken] == x -> y1 y2 .. ym
      stack.push(ym); ...; stack.push(y1);
    else error()
until x is EOF
```

No entry in table: Raise error

Example: Table-based Parsing

Input: bcc

Stack	Remaining input	Steps
EOF, S	b, c, c, EOF	Pop S Use rule $S \to bC$ Push C, b
EOF, C, b	b, c, c, EOF	Pop b Read next token
EOF, C	c, c, EOF	Pop C Use rule $C \to cc$ Push c, c
EOF, c, c	c, c, EOF	Pop c Read next token.
EOF, c	c, EOF	Pop c Read next token
EOF	EOF	Pop EOF

$\to$ done

Plan for Today

- **Specifying syntax**
  - Regular expressions
  - Context-free grammars
- **Scanning**
- **Parsing**
  - Top-down parsing
  - Bottom-up parsing

Bottom-up Parsing

- **LR(k) parsers**
  - Left-to-right scanning, Right-most derivation, k tokens look-ahead
- **Difficult to do by hand**
- **Mostly based on automatically generated table**

Shift-reduce Algorithm

- **Repeat until all tokens read and all symbols reduced to start symbol:**
  - Shift (i.e., read) input tokens
  - Try to reduce a group of symbols into a single non-terminal

Example: Shift-Reduce Parsing

$S \to a\,T\,R\,e$

$T \to T\,b\,c \mid b$

$R \to d$

Steps:

Shift a, shift b

Reduce $T \to b$

Shift b, shift c

Reduce $T \to T\,b\,c$

Shift d

Reduce $R \to d$

Shift e

Reduce $S \to a\,T\,R\,e$

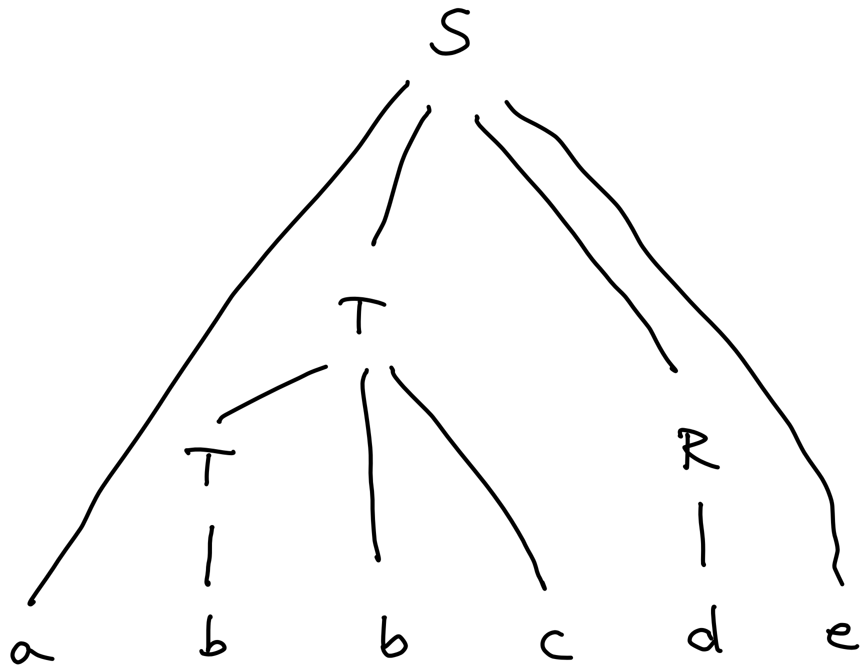

Table-based LR Parsing

- **Two tables**
  - Action table:

    state $\times$ T $\rightarrow$ reduce/shift/accept/error
  - Goto table:

    state $\times$ N $\rightarrow$ state
- **Stack of symbol/state pairs**
  - Record of what has been seen in the past

Example: LR(1) Table

State	a	b	c	d	e	EOF	S	T	R
s0	s1								
s1		s3						2	
s2		s5		s6					4
s3		r3		r3					
s4					s7				
s5			s8						
s6					r4				
s7						acc.			
s8		r2		r2					

Example: LR(1) Table

State	a	b	c	d	e	EOF	S	T	R
s0	s1								
s1		s3						2	
s2		s5		s6					4
s3		r3		r3					
s4					s7				
s5			s8						
s6					r4				
s7						acc.			
s8		r2		r2					

Action table

Example: LR(1) Table

State	a	b	c	d	e	EOF	S	T	R
s0	s1								
s1		s3						2	
s2		s5		s6					4
s3		r3		r3					
s4					s7				
s5			s8						
s6					r4				
s7						acc.			
s8		r2		r2					

Goto table

Example: LR(1) Table

State	a	b	c	d	e	EOF	S	T	R
s0	s1								
s1		s3						2	
s2		s5		s6					4
s3		r3		r3					
s4					s7				
s5			s8						
s6					r4				
s7						acc.			
s8		r2		r2					

s means shift to some state

Example: LR(1) Table

State	a	b	c	d	e	EOF	S	T	R
s0	s1								
s1		s3						2	
s2		s5		s6					4
s3		r3		r3					
s4					s7				
s5			s8						
s6					r4				
s7						acc.			
s8		r2		r2					

r means reduce using some production

Example: LR(1) Table

State	a	b	c	d	e	EOF	S	T	R
s0	s1								
s1		s3						2	
s2		s5		s6					4
s3		r3		r3					
s4					s7				
s5			s8						
s6					r4				
s7						acc.			
s8		r2		r2					

Accept input (i.e., done with parsing)

Example: LR(1) Table

State	a	b	c	d	e	EOF	S	T	R
s0	s1								
s1		s3						2	
s2		s5		s6					4
s3		r3		r3					
s4					s7				
s5			s8						
s6					r4				
s7						acc.			
s8		r2		r2					

No entry means error

Table-based LR(1) Parsing

```
stack.push(EOF, 0);
nextToken = lookAhead();
repeat
  s = state on top of stack
  if action[s, nextToken] = shift s′
    stack.push(nextToken, s′);
    nextToken = lookAhead();
  else if action[s, nextToken] = reduce x -> y1 .. ym
    pop m pairs from stack
    s′ = state on top of stack
    push(x, goto[s′, x])
  else if action[s, nextToken] = accept
    accept and return
  else error()
```

Table-based LR(1) Parsing

Stack hold roots of partial trees found so far

```
stack.push(EOF, 0);
nextToken = lookAhead();
repeat
  s = state on top of stack
  if action[s, nextToken] = shift s′
    stack.push(nextToken, s′);
    nextToken = lookAhead();
  else if action[s, nextToken] = reduce x -> y1 .. ym
    pop m pairs from stack
    s′ = state on top of stack
    push(x, goto[s′, x])
  else if action[s, nextToken] = accept
    accept and return
  else error()
```

Table-based LR(1) Parsing

```
stack.push(EOF, 0);
nextToken = lookAhead();
repeat
  s = state on top of stack
  if action[s, nextToken] = shift s′
    stack.push(nextToken, s′);
    nextToken = lookAhead();
  else if action[s, nextToken] = reduce x -> y1 .. ym
    pop m pairs from stack
    s′ = state on top of stack
    push(x, goto[s′, x])
  else if action[s, nextToken] = accept
    accept and return
  else error()
```

Reduce partial trees into a non-terminal by applying a rule

Table-based LR(1) Parsing

```
stack.push(EOF, 0);
nextToken = lookAhead();
repeat
  s = state on top of stack
  if action[s, nextToken] = shift s′
    stack.push(nextToken, s′);
    nextToken = lookAhead();
  else if action[s, nextToken] = reduce x -> y1 .. ym
    pop m pairs from stack
    s′ = state on top of stack
    push(x, goto[s′, x])
  else if action[s, nextToken] = accept
    accept and return
  else error()
```

Read another token

Table-based LR(1) Parsing

```
stack.push(EOF, 0);
nextToken = lookAhead();
repeat
  s = state on top of stack
  if action[s, nextToken] = shift s′
    stack.push(nextToken, s′);
    nextToken = lookAhead();
  else if action[s, nextToken] = reduce x -> y1 .. ym
    pop m pairs from stack
    s′ = state on top of stack
    push(x, goto[s′, x])
  else if action[s, nextToken] = accept
    accept and return
  else error()
```

All subtrees reduced to start symbol

How to Get the Table?

- **Using a "characteristic finite-state machine" computed from the grammar**
- **Details differ for different kinds of LR parsers**
  - SLR (simple LR)
  - LALR (look-ahead LR)
  - Full-LR
- **Beyond the scope of this course**

Quiz: Parsing

Which of these statements is true?

- Recursive descent builds a parse tree from the bottom up.
- The k in LR(k) stands for k tokens look-ahead.
- PREDICT sets are used to compute FIRST and FOLLOW sets.
- The stack of a top-down parser contains the symbols expected in the future.

https://ilias3.uni-stuttgart.de/vote/0ZT9

Quiz: Parsing

Which of these statements is true?

- ~~Recursive descent builds a parse tree from the bottom up~~.
- The k in LR(k) stands for k tokens look-ahead.
- ~~PREDICT sets are used to compute FIRST and FOLLOW sets~~.
- The stack of a top-down parser contains the symbols expected in the future.

https://ilias3.uni-stuttgart.de/vote/0ZT9

Plan for Today

- **Specifying syntax**
  - Regular expressions
  - Context-free grammars
- **Scanning**
- **Parsing**
  - Top-down parsing
  - Bottom-up parsing ✔