Parsing Algorithms

CS 4447/CS 9545 -- Stephen Watt
University of Western Ontario

The Big Picture

- Develop parsers based on grammars
- Figure out properties of the grammars
- Make tables that drive parsing engines

- Two essential ideas:
  Derivations and *FIRST/FOLLOW sets*

Outline

- Grammars, parse trees and derivations.
- Recursive descent parsing
- Operator precedence parsing
- Predictive parsing
  - *FIRST* and *FOLLOW*
  - LL(1) parsing tables. LL(k) parsing.
- Left-most and right-most derivations
- Shift-reduce parsing
  - LR parsing automaton. LR(k) parsing.
  - LALR(k) parsing.

Example Grammar G1

- We have seen grammars already.
  Here is an example.
  [from *Modern Compiler Implementation in Java*, by Andrew W. Appel]

1. $S \rightarrow S$ “;” S
2. $S \rightarrow$ id “:=” E
3. $S \rightarrow$ “print” “(” L “)”
4. $E \rightarrow$ id
5. $E \rightarrow$ num
6. $E \rightarrow E$ “+” E
7. $E \rightarrow$ “(“ S “,” E “)”
8. $L \rightarrow E$
9. $L \rightarrow L$ “,” E

Parse Trees

- A *parse tree*
  for a given grammar and input
  is a tree where
  each node corresponds to a grammar rule, and
  the leaves correspond to the input.

Example Parse Tree

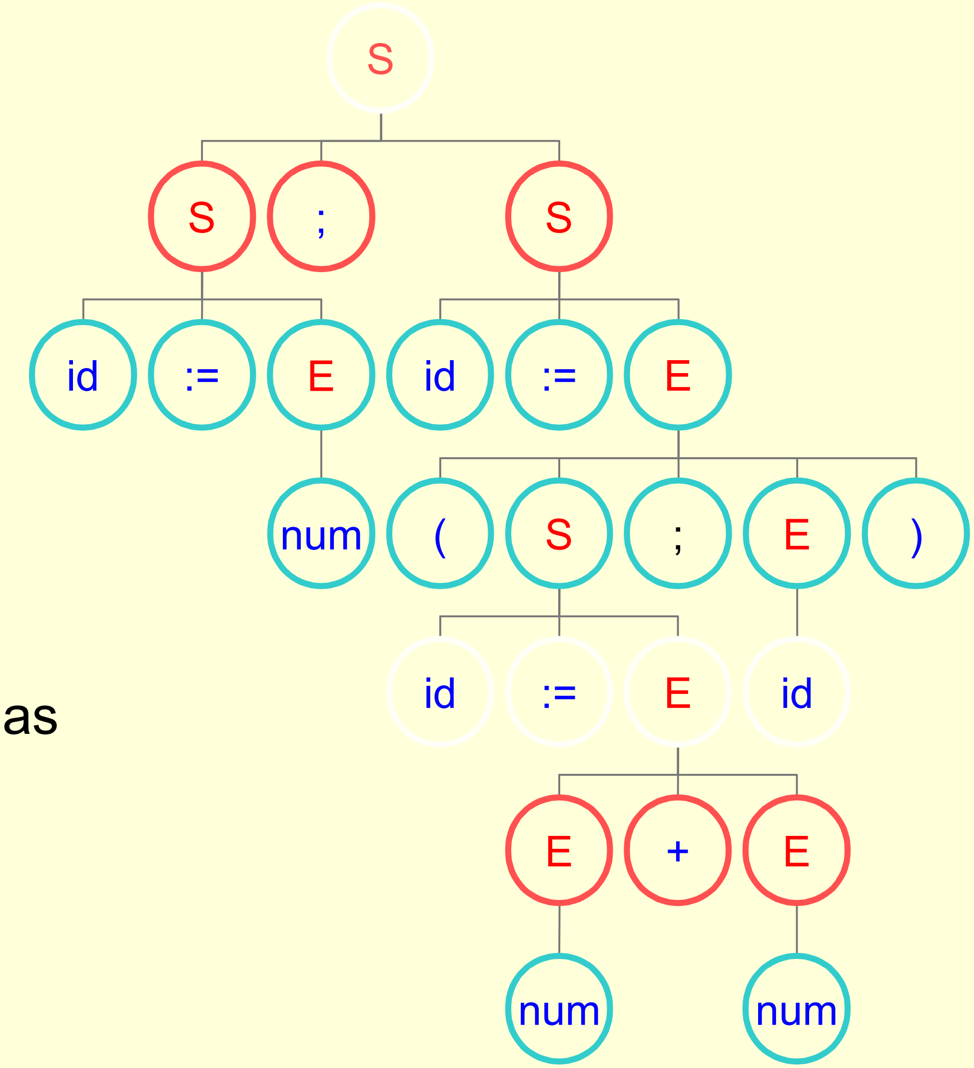
- Consider the following input:

```
a := 7;
b := c + (d := 5 + 6, d)
```

- This gives the *token* sequence:

  id := num ; id := id +
  (id := num + num , id)

- Using example grammar G1, this has a parse tree shown on the right.



Derivations

- “Parsing” figures out a parse tree for an given input
- A “derivation” gives a rationale for a parse.

- Begin with the grammar’s start symbol and repeatedly replace non-terminals until only terminals remain.

Example Derivation

This derivation justifies the parse tree we showed.

S

S ; S

S ; id := E

id := E ; id := E

id := num ; id := E

id := num ; id := E + E

id := num ; id := E + (S, E)

id := num ; id := id + (S , E)

id := num ; id := id + (id := E , E)

id := num ; id := id + (id := E + E, E)

id := num ; id := id + (id := E + E, id)

id := num ; id := id + (id := num + E, id)

id := num ; id := id + (id := num + num, id)

Derivations and Parse Trees

- A node in a parse tree corresponds to the use of a rule in a derivation.

- A grammar is *ambiguous* if it can derive some sentence with two *different* parse trees.

  E.g. a + b * d can be derived two ways using the rules

  $E \to id$ $E \to E$ "+" E $E \to E$ "*" E

- Even for an unambiguous grammar, there is a choice of which non-terminal to replace in forming a derivation.

  Two choices are
  - Replace the leftmost non-terminal
  - Replace the rightmost non-terminal

Recursive Descent Parsing

- Example for recursive descent parsing:

1. S → E		
2. E → T “+” E	3. E → T	
4. T → F “*” T	5. T → F	
6. F → P “^” F	7. F → P	
8. P → id	9. P → num	10. P → “(” E “)”

- Introduce one function for each non-terminal.

Recursive Descent Parsing (cont’d)

```
PT* S() { return E(); }

PT* E() { PT *pt = T();
          if (peek(“+”)) { consume(“+”); pt = mkPT(pt,E()); }
          return pt; }

PT* T() { PT *pt = F();
          if (peek(“*”)) { consume(“*”); pt = mkPT(pt,F()); }
          return pt; }

PT* F() { PT *pt = P();
          if (peek(“^”)) { consume(“^”); pt = mkPT(pt,P()); }
          return pt; }

PT* P() { PT *pt;
          if (peekDigit())  return new PT(Num());
          if (peekLetter()) return new PT(Id());
          consume(“(”); pt = E(); consume(“)”);
          return pt; }
```

Recursive Descent Parsing -- Problems

- A slightly different grammar (G2) gives problems, though:

  1. S → E
  2. E → E "+" T   3. E → T
  4. T → T "*" F   5. T → F
  6. F → P "^" F   7. F → P
  8. P → id   9. P → num   10. P → "(" E ")"

- This causes problems, e.g.:
  - Do not know whether to use rule 2 or rule 3 parsing an E.
  - Rule 2 gives an infinite recursion.

- We want to be able to *predict* which rule (which recursive function) to use, based on looking at the current input token.

Operator Precedence Parsing

- Each operator has left- and right- precedence. E.g.

  100+101 200×201 301^300

- Group sub-expressions by binding highest numbers first.

  A+B × C × D ^ E ^ F

  A 100 +101 B 200×201 C 200×201 D 301^300 E 301^300 F

  A 100 +101 B 200×201 C 200×201 D 301^300 (E 301^300 F)

  A 100 +101 B 200×201 C 200×201 (D 301^300 (E 301^300 F))

  A 100 +101 (B 200×201 C) 200×201 (D 301^300 (E 301^300 F))

  A 100 +101 ((B 200×201 C) 200×201 (D 301^300 (E 301^300 F)))

  A+((B × C) × (D ^ (E ^ F)))

- Works fine for infix expressions but not well for general CFL.

Predictive Parsing – FIRST sets

- We introduce the notion of "FIRST" sets that will be useful in predictive parsing.
- If **α** is a string of terminals and non-terminals, then FIRST(**α**) is the set of all terminals that may be the first symbol in a string derived from **α.**
- Eg1: For example grammar G1,

  FIRST(S) = { id, "print" }

- Eg2: For example grammar G2,

  FIRST(T "*" F) = { id, num, "(" }

Predictive Parsing -- good vs bad grammars

- If two productions for the same LHS have RHS with intersecting FIRST sets, then the grammar cannot be parsed using predictive parsing.

  E.g. with $E \rightarrow E$ “+” T and $E \rightarrow T$
  FIRST(E “+” T) = FIRST(T) = { id, num, “(”}

- To use predictive parsing, we need to formulate a different grammar for the same language.

- One technique is to eliminate left recursion:

  E.g. replace $E \rightarrow E$ “+” T and $E \rightarrow T$
  with $E \rightarrow T\ E'$ $E' \rightarrow$ “+” $T\ E'$ $E' \rightarrow \varepsilon$

The “nullable” property

- We say a non-terminal is “nullable” if it can derive the empty string.
- In the previous example E’ is nullable.

FOLLOW sets

- The “FOLLOW” set for a non-terminal X is the set of terminals that can immediately follow X.
- The terminal t is in FOLLOW(X) if there is a derivation containing Xt.
- This can occur if there is a derivation containing X Y Z t, if Y and Z are nullable.

Algorithm for FIRST, FOLLOW, nullable

for each symbol X
  FIRST[X] := { }, FOLLOW[X] := { }, nullable[X] := false

for each terminal symbol t
  FIRST[t] := {t}

repeat
  for each production X → Y1 Y2 … Yk,
    if all Yi are nullable **then**
      nullable[X] := true
    if Y1..Yi-1 are nullable **then**
      FIRST[X] := FIRST[X] U FIRST[Yi]
    if Yi+1..Yk are all nullable **then**
      FOLLOW[Yi] := FOLLOW[Yi] U FOLLOW[X]
    if Yi+1..Yj-1 are all nullable **then**
      FOLLOW[Yi] := FOLLOW[Yi] U FIRST[Yj]

until FIRST, FOLLOW, nullable do not change

Example FIRST, FOLLOW, nullable

Example Grammar G3.

$Z \rightarrow d$　　$Y \rightarrow \varepsilon$　　$X \rightarrow Y$

$Z \rightarrow X\ Y\ Z$　　$Y \rightarrow c$　　$X \rightarrow a$

	nullable	FIRST	FOLLOW
X	false		
Y	false		
Z	false		

	nullable	FIRST	FOLLOW
X	false	a	c d
Y	true	c	d
Z	false	d	

	nullable	FIRST	FOLLOW
X	true	a c	a c d
Y	true	c	a c d
Z	false	a c d	

Predictive Parsing Tables

- Rows: Non-terminals
- Columns: Terminals
- Entries: Productions

$Z \rightarrow d$ $\quad Y \rightarrow \varepsilon$ $\quad X \rightarrow Y$

$Z \rightarrow X\,Y\,Z$ $\quad Y \rightarrow c$ $\quad X \rightarrow a$

Enter production $X \rightarrow \alpha$ in row X, column t for each t in FIRST(α).

If α is nullable, enter the productions in row X, column t
for each t in FOLLOW(X).

	a	c	d
X	$X \rightarrow a$ $X \rightarrow Y$	$X \rightarrow Y$	$X \rightarrow Y$
Y	$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$ $Y \rightarrow c$	$Y \rightarrow \varepsilon$
Z	$Z \rightarrow X\,Y\,Z$	$Z \rightarrow XYZ$	$Z \rightarrow d$ $Z \rightarrow X\,Y\,Z$

	nullable	FIRST	FOLLOW
X	true	a c	a c d
Y	true	c	a c d
Z	false	a c d	

Example of Predictive Parsing

Initial grammar

S → E

E → E "+" T   E → T

T → T "*" F   T → F

F → id

F → num

F → "(" E ")"

Modified grammar

S → E $

E → T E'   E' →   E' → "+" T E'

T → F T'   T' →   T' → "*" F T'

F → id

F → num

F → "(" E ")"

	Nullable	FIRST	FOLLOW
S	False	(id num	
E	False	(id num	) $
E'	True	+	) $
T	False	(id num	) + $
T'	True	*	) + $
F	False	(id num	) * + $

Example of Predictive Parsing (contd)

$S \rightarrow E\ \$$
$E \rightarrow T\ E'$ $E' \rightarrow$ $E' \rightarrow \text{“+”}\ T\ E'$
$T \rightarrow F\ T'$ $T' \rightarrow$ $T' \rightarrow \text{“*”}\ F\ T'$
$F \rightarrow id$
$F \rightarrow num$
$F \rightarrow \text{“(”}\ E\ \text{“)”}$

	Nullable	FIRST	FOLLOW
S	False	(id num	
E	False	(id num	) $
E'	True	+	) $
T	False	(id num	) + $
T'	True	*	) + $
F	False	(id num	) * + $

	+	*	id	num	(	)	$
S			S → E$	S → E$	S → E$		
E			E → TE'	E → TE'	E → TE'		
E'	E' → “+” T E'					E' →	E' →
T			T → FT'	T → FT'	T → FT'		
T'	T' →	T' → “*” F T'				T' →	T' →
F			F → id	F → num	F → “(” E “)”		

LL(k) Grammars

- The predictive parser we built makes use of one look ahead token.
  - We say the grammar is LL(1).
  - LL stands for "Left to right parse, Leftmost derivation"
- If k look ahead tokens are needed, then we say the grammar is LL(k).
  - For k > 1, the columns are the possible sequences of k tokens, and the tables become large.
- There is a better way…

LR Parsing

- LL parsing always uses a grammar rule for the *left-most* non-terminal.
- If we aren't so eager, we can apply grammar rules to other non-terminals
- This allows us to decide about the "hard" non-terminals later.
- We keep a stack of unfinished work.
- Using the *right-most* derivation leads to LR parsing.

LR Parsing

- Parser state consists of a *stack* and *input.*
- First *k* tokens of the unused input is the *"lookahead"*
- Based on what is on the top of the stack and the lookahead, the parser decides whether to
  - Shift = 1. consume the first input token
    2. push it to the top of the stack
  - Reduce = 1. choose a grammar rule X → A B C
    2. pop C, B, A from the stack
    3. push X onto the stack