CS4410 : Spring 2013

A little bit about LR Parsing

Background

- We'll see three ways to write parsers:
  - By hand, typically recursive descent
  - Using parsing combinators
    - In both of these, left-recursion is a problem.
  - Using tools like Lex and Yacc
- Lex: limited to regular expressions
  - Compile regexp to NDFA, then to DFA
- Yacc: limited to LALR(1) grammars
  - Read details in book! This is just intuition.

Various Kinds of Grammars

- General context free grammars.
- Regular expressions (no recursion).
- LL(k): *left-to-right, left-most derivation with k symbol lookahead.*
- LR(k): *left-to-right, right-most derivation with k symbol lookahead.*
  - SLR(k) and LALR(k) are restricted variants that are almost as good, but much more space-efficient.

Intuition Behind Predictive Parsing

- Parsing combinators make it easy to write recursive descent parsers, but do lots of back-tracking:

  alt p1 p2 = fun cs -> (p1 cs) @ (p2 cs)

  - We run p1 on the string cs, then back up and run p2 on the same input.

- It would be much better if we could predict whether to select p1 or p2 and avoid running both of them.
  - Want a procedure first(p), which computes the set of characters that strings matching p can start with:
  - Check if the next input character is in p1 – if not, then we can skip it. Similarly with p2.
  - Ideally, it's only in one so we don't have to backtrack.

Computing First Sets

e → t | t + e
t → INT | (e)

first(e) = first(t)
first(t) = {INT, (}

Seems pretty easy.

But consider…

N → a | M P Q

M → b | ε

P → c*

Q → d N

First(N) = {a} + first(M)?

But consider…

N → a | M P Q
M → b | ε
P → c*
Q → d N

first(N) = {a} + first(M)?

Since M can match the empty string, N might start with a character in first(P). And since P can match the empty string, it must also include first(Q)!

But consider…

N → a | M P Q
M → b | ε
P → c*
Q → d N

first(N) = {a} + first(M) + first(P) + first(Q)
= {a,b,c,d}

Since M can match the empty string, N might start with a character in first(P). And since P can match the empty string, it must also include first(Q)!

In General:

- Given a parsing expression (X1 X2 … Xn)
- To compute it's first set:
  - Add first(X1).
  - Check if X1 is *nullable* (i.e., derives empty string).
  - If so, then add first(X2), and if X2 is nullable, …
- So we need a procedure to check if a non-terminal is nullable.

Not Enough

- In general, our alternatives will not have disjoint first(-) sets.
  - We could try to transform the grammar so that this is the case.
  - We effectively did this for regular expressions by compiling to NDFAs and then to DFAs.
  - Alternatively, we can calculate the *derivative* of the grammar with respect to each element of the first set.
  - Unfortunately, for CFGs, the derivatives (or states) are not finite…
- So a tool like Yacc does a lot more…

LR(1) Parsing a la Yacc

- Use a stack and DFA:
  - The states correspond to parsing *items*
    - Roughly, a set of productions marked with a position.
    - Equivalently, a derivative
    - Lets us effectively remember where we came from.
  - The DFA tells us what to do when we peek at the next character in the input.
    - For LR(k), we look at k-symbols of input.
    - By using the first sets (and other computations, like follow sets), we can predict what production(s) we should work with and avoid back-tracking.
  - Possible actions:
    - *Shift* the input symbol, pushing it onto the stack.
    - *Reduce* symbols on the top of the stack to a non-terminal.

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:

[]

Input:

(3 + 4) + (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:

[]

Shift the '(' onto the stack.

Input:

(3 + 4) + (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:

[]

['(']

Input:

(3 + 4) + (5 + 6)

3 + 4) + (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:

[]

['(']

Can't reduce, so shift the 3 onto the stack.

Input:

(3 + 4) + (5 + 6)

3 + 4) + (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:

[]

['(']

['(',INT]

Input:

(3 + 4) + (5 + 6)

3 + 4) + (5 + 6)

+ 4) + (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Reduce by production 1

Stack:

[]

['(']

['(',INT]

Input:

(3 + 4) + (5 + 6)

3 + 4) + (5 + 6)

+ 4) + (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:

[]

['(']

['(',INT]

['(',e]

Input:

(3 + 4) + (5 + 6)

3 + 4) + (5 + 6)

+ 4) + (5 + 6)

+ 4) + (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:

[]
['(']
['(',INT]
['(',e]

Shift the '+' onto the stack

Input:

(3 + 4) + (5 + 6)
3 + 4) + (5 + 6)
+ 4) + (5 + 6)
+ 4) + (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:

['(',e,'+']

Shift 4 onto stack.

Input:

4) + (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Reduce using rule 1

Stack:

['(',e,'+']

['(',e,'+',INT]

Input:

4) + (5 + 6)

) + (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:

['(',e,'+']
['(',e,'+',INT]
['(',e,'+',e]

Reduce again with rule 3.

Input:

4) + (5 + 6)
) + (5 + 6)
) + (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:	Input:
['(',e,'+']	4) + (5 + 6)
['(',e,'+',INT]	) + (5 + 6)
['(',e,'+',e]	) + (5 + 6)
['(',e]	) + (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Shift ')' onto the stack

Stack:	Input:
['(',e,'+']	4) + (5 + 6)
['(',e,'+',INT]	) + (5 + 6)
['(',e,'+',e]	) + (5 + 6)
['(',e]	) + (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:	Input:
['(',e,'+',e]	) + (5 + 6)
['(',e]	) + (5 + 6)
['(',e,')']	+ (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:

['(',e,'+',e]
['(',e]
['(',e,')']

Reduce by rule 2.

Input:

) + (5 + 6)
) + (5 + 6)
+ (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:	Input:
['(',e,'+',e]	) + (5 + 6)
['(',e]	) + (5 + 6)
['(',e,')']	+ (5 + 6)
[e]	+ (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:

['(',e,'+',e]
['(',e]
['(',e,')']
[e]

Continuing on…

Input:

) + (5 + 6)
) + (5 + 6)
+ (5 + 6)
+ (5 + 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:	Input:
[e]	+ (5 + 6)
[e,'+']	(5 + 6)
[e,'+','(']	5 + 6)
[e,'+','(',INT]	+ 6)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:	Input:
[e,'+','(',INT]	+ 6)
[e,'+','(',e]	+ 6)
[e,'+','(',e,'+']	6)
[e,'+','(',e,'+',INT]	)

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:	Input:
[e,'+','(',e,'+',INT]	)
[e,'+','(',e,'+',e]	)
[e,'+','(',e]	)
[e,'+','(',e,')']	<none>

Example

1. e → INT
2. e → (e)
3. e → e + e

Stack:

[e,'+','(',e,')']
[e,'+',e]
[e]

Input:

<none>

Right-Most Derivation

If we read the sequence of stacks backwards, we are always expanding the right-most non-terminal. (Hence the "R" in LR parsing.)

Stack:

[e,'+','(',e,'+',e,')']

[e,'+','(',e,')']

[e,'+',e]

[e]

The Tricky Part

- When do we shift and when do we reduce?
  - In general, whether to shift or reduce can be hard to figure out, even when the grammar is unambiguous.
  - When it is ambiguous, we get a conflict:
    - Reduce-Reduce: Yacc can't figure out which of two productions to use to collapse the stack.
    - Shift-Reduce: Yacc can't figure out whether it should shift or whether it should reduce.
    - Look at generated parse.grm.desc file for details.
  - Two fixes:
    - Use associativity & precedence directives
    - Rewrite the grammar

For Our Simple Grammar

State 0:
e : . INT (shift INT 3)
e : . '(' e ')' (shift '(' 2)
e : . e '+' e (or goto state 6)

State 1:
e : e . '+' e (shift '+' 4)

State 2:
e : '(' . e) (shift INT 3)
(shift '(' 2)

State 3:
e : INT . (reduce rule 0)

State 4:
e : e '+' . e (shift INT 3)
(shift '(' 2)

State 5:
e : '(' e . ')' (shift ')' 7)
e : e . '+' e (shift '+' 4)

State 6:
e : e . '+' e (shift '+' 4)
e : e '+' e . (reduce rule 2)

State 7:
e : '(' e ')'. (reduce rule 1)

In this case:

- The fix is to tell Yacc that we want + to be left-associative (in which case it will reduce instead of shifting.)
- Alternatively, rewrite:

e → t

e → t + e

t → INT

t → (e)

Another Example:

s → var := e ;

s → if e then s else s

s → if e then s

Conflict

- if e then (if e then x := 3; else x := 4)
- if e then (if e then x := 3;) else x := 4

By default, Yacc will shift instead of reducing.
(So we get the first parse by default.)
Better to engineer this away…

- Ideally, put in an ending delimiter (end-if)
- Or refactor grammar…

Refactoring

s → os

os → var := e ;

→ if e then os

→ if e then cs else os

cs → var := e ;

→ if e then cs else cs

if e then if e then x:=3; else x:=4;

What You Need to Know

- I'm less concerned that you understand the details in constructing predictive parsing tables.
  - Read the chapters in Appel.
  - They will help you understand what Yacc is doing.
- You should be able to build parsers using Yacc for standard linguistic constructs.
  - That means understanding what conflicts are, and how to avoid them.
  - Hence the Fish front-end.
  - Real grammars are, unfortunately, rather ugly.

One More Consideration

- Error messages:
  - Yacc will generate an error message, but it can easily get confused.
    - e.g., if you forget a closing parenthesis or misspell a keyword.
    - Particularly bad for cascading errors.
  - You can add so-called error productions to the grammar for common mistakes so that you get a more informative error message.
  - Yacc will try to continue when it finds an error by inserting or skipping over tokens (basically a hack.)
  - Probably the *right* thing to do is fall back on the general search paradigm to find possible "close" parses and pick the one that minimizes the syntax errors.
  - Simple grammars (a la Scheme) result in much better error messages and error recovery.