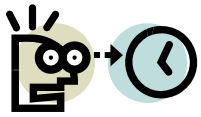
Introduction to Shift/Reduce Parsing

Procrastination is Good

Tuesday, November 24, 2009
Reading: Appel 3.3



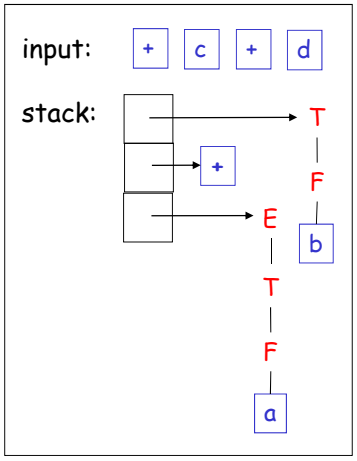
(Lyn before a CS235 lecture!)

CS235 Languages and Automata

Department of Computer Science
Wellesley College

Postponing decisions with a stack

- LL(k) parsing techniques must predict which production to use based on the next k tokens.
- Today, we study so-called LR(k) parsers, which postpone the decision until it has seen input tokens of the entire right-hand side of the production in question.
- Postponement is achieved by pushing tokens and partially-built parse trees on a stack. This allows "seeing" more input before making decisions, and building left-recursive trees in a bottom-up fashion.



Rose Trees: A Motivational Example

Productions are numbered for easy reference

Appel uses $ instead of EOF

0	S′	→	S $
1	S	→	(L)
2	S	→	x
3	L	→	S
4	L	→	L , S

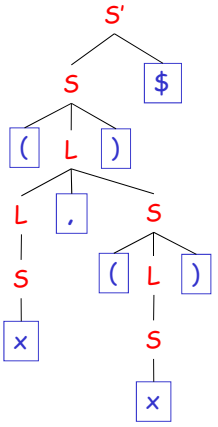
$G_{RoseTree}$
(Appel's Grammar 3.20)

Examples:

x$ (x)$ (x,x)$ (x,x,x)$

(x,(x))$ (((x,x),(x)),x,((x,x,x,x)))$

Parse tree for (x, (x))$



$G_{RoseTree}$ is Not LL(k) (i.e., not Predictive)

$G_{RoseTree}$

0	S′	→	S $
1	S	→	(L)
2	S	→	x
3	L	→	S
4	L	→	L , S

LL(1) Parsing Table

	x	(
S′	S′ → S $	S′ → S $
S	S → x	S → (L)
L	L → S L → L , S	L → S L → L , S

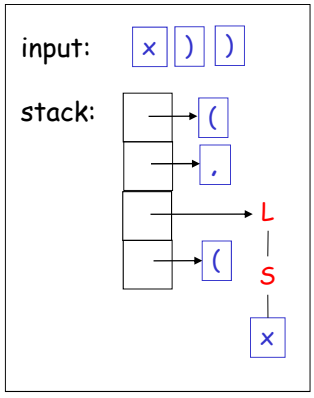
- $G_{RoseTree}$ is not LL(1)
- Not LL(k) for any k!
- Isn't there a better way?

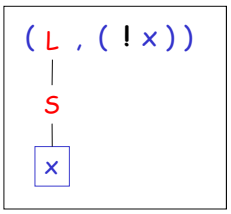
A Better Way: Shift/Reduce Parsing (LR(k))

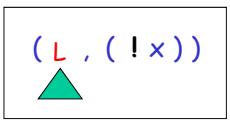
- The state of the parser (a **configuration**) has two components:
  1. The still-to-be-processed **input** tokens
  2. A **stack** of tokens and parse trees for the already processed tokens
- On each step of the parsing process, one of two actions occurs:
  1. The first input token is **shifted** to the top of the stack.
  2. The top k stack elements are **reduced** to a variable according to a production in the grammar.
- Parsing succeeds if there is a configuration where the stack contains only the "real" start symbol of the grammar (S, not S') after all input tokens except $ have been processed.
- Shift/reduce (LR(k)) parsing is more powerful than predictive (LL(k)) parsing because the decision of what to do next can take into account the stack elements as well as the next few input tokens

A Sample Configuration

Here's a sample intermediate configuration from parsing (x,(x))$ along with some abbreviations:







(To be more consistent with Appel's notatation, we should use a period rather than an exclamation point to separate stack and input, but the period is too hard to see.)

An Example: Parsing (x,(x))$

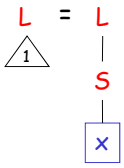
!(x,(x))$

⇒(!x,(x))$ *shift* (

⇒(x!,(x))$ *shift* x

⇒(S!,(x))$ *reduce* S → x

⇒(L!,(x))$ *reduce* L → S, where



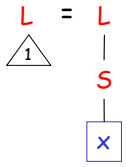
⇒(L,!(x))$ *shift* ,

⇒(L,(!x))$ *shift* (

⇒(L,(x!))$ *shift* x

Parsing (x,(x))$ (Continued)

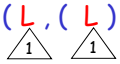
(L,(x!))$



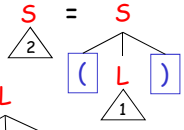
⇒(L,(S!))$ *reduce* S → x

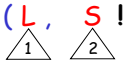
⇒(L,(L!))$ *reduce* L → S

⇒(L,(L)!)$ *shift*)

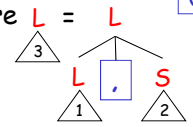


⇒(L, S!)$ *reduce* S → (L), where





⇒(L!)$ *reduce* L → L,S, where

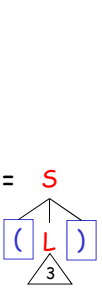


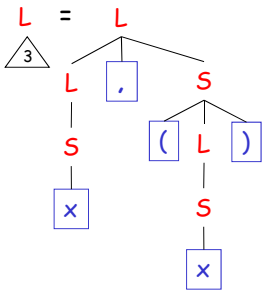
Parsing (x,(x))$ (Continued Again)

(L !) $
3

⇒ (L) ! $ *shift*)
3

⇒ S ! $ *reduce* S → (L) , where S = S
4 4





By convention, this is an accepting state
(we never reduce S' → S $)

A Rightmost Derivation of (x, (x))$

S' ⇒ S $	*by* S' → S $
⇒ (L) $	*by* S → (L)
⇒ (L , S) $	*by* L → L , S
⇒ (L , (L)) $	*by* S → (L)
⇒ (L , (S)) $	*by* L → S
⇒ (L , (x)) $	*by* S → x
⇒ (S , (x)) $	*by* L → S
⇒ (x , (x)) $	*by* S → x

Observe that our shift/reduce parsing example for (x, (x))$ performs the productions of the rightmost derivation precisely in reverse order.

If we had constructed a derivation rather than a parse tree, we would have constructed precisely the rightmost derivation.

LR(k) Parsing

A context-free grammar is LR(k) iff it can be parsed by a shift/reduce parser using *k* tokens of lookahead from the input.

In **LR**(k)

- The **L** means that the tokens are processed **L**eft-to-right
- The **R** means that the result of parsing is a parse tree constructed via a **R**ightmost derivation.

As in LL(k) parsing, LR(k) parsing is guided by a parsing table, as we'll see soon.

We'll see that $G_{RoseTree}$ is LR(0) : the parser doesn't actually need to look at any input tokens in order to determine whether to shift or reduce. It makes this decision based on the stack alone.

But there are LR grammars that require nonzero lookahead.

Shift-Reduce Example: Postfix Expressions

S → E $

E → ID(str) | INT(int) | E E + | E E *

Examples:

2 3 * 4 + $

2 3 4 * + $

Infix Expressions Revisited

$G_{IntExpAmbig}$

S → E $

E → INT(int) | E + E | E * E | E ^ E

- ^ is an exponentiation operator
- leave out - , /, and (E) for simplicity
- **precedence**: + < * < ^
- **associativity**:
  - + and * are left associative
  - ^ is right associative
    (Why? If left associative, then
    x ^ y ^ z = (x^y)^z = x^(y*z).
    So x^(y^z) expresses something different)

Shift-Reduce Parsing of Integer Expressions

When shift-reduce parsing integer expressions, we must choose between shifting an input token and reducing a production RHS on the stack. The choice for +|*|^ is determined by precedence/associativity.

	stack	next token	shift or reduce?
1	⊥	INT(i)	
2	… E +|*|^	INT(i)	
3	… INT(i)	+|*|^|$	
3	⊥ E	+|*|^	
4	… E +|*|^ E	+	
5	… E + E	*	
6	… E *|^ E	*	
7	… E +|*|^ E	^	
8	… E +|*|^ E	$	
9	⊥ E	$	

Example

Using the rules in the table, parse the following expression using a shift-reduce parser:

1 + 2 * 3 ^ 4 ^ 5 * 6 + 7 $

Manually Building a Shift-Reduce Parser in SML

Based on the shift-reduce table, we can build a shift-reduce parser for integer expressions in SML (~cs235/download/intexp-parsers/IntexpParserInfix.sml)

```
datatype stkval = Tok of token
                |   Exp of exp

 (* step: (stackval list) * (token list) -> exp *)
 and step( [Exp(e)], [] ) = e (* final parsed expression *)

   (* Reduce binapps on stack when reach end of input *)
   | step( Exp(e2)::Tok(OP(binop))::Exp(e1)::stk,  [] ) =
     step( Exp(BinApp(binop,e1,e2))::stk,          [] ) (*reduce*)

   (* Integer token cases *)
   | step(           [],  INT(i)::toks ) =
     step( [Exp(Int(i))],          toks ) (*shift*)

   | step(             Tok(OP(binop))::Exp(e)::stk,  INT(i)::toks ) =
     step( Exp(Int(i))::Tok(OP(binop))::Exp(e)::stk,          toks ) (*shift*)

   (* Always shift operator onto singleton stack *)
   | step(               [Exp(e)],  OP(binop)::toks ) =
     step( [Tok(OP(binop)),Exp(e)],            toks ) (*shift*)
```

AShift-Reduce Parser in SML (cont.)

```
(* Always reduce binapp on stack when see Add/Sub token *)
   | step( Exp(e2)::Tok(OP(binop))::Exp(e1)::stk,  OP(addop as (Add|Sub))::toks ) =
     step( Exp(BinApp(binop,e1,e2))        ::stk,  OP(addop)             ::toks ) (*reduce*)

   (* Mul/Div token cases *)
   (*   Case 1: Shift Mul/Div if lower precedence binapp is on stack *)
   | step(    Exp(e2)::Tok(OP(addop as (Add|Sub)))::Exp(e1)::stk,
              OP(mulop as (Mul|Div))::toks ) =
     step( Tok(OP(mulop))::Exp(e2)::Tok(OP(addop))::Exp(e1)::stk, toks ) (*shift*)
   (*   Case 2: Reduce binapp if equal or higher precedence
        (If previous pattern didn't apply, then below binop must be Mul/Div/Expt) *)
   | step( Exp(e2)::Tok(OP(binop))::Exp(e1)::stk,  OP(mulop as (Mul|Div))::toks ) =
     step( Exp(BinApp(binop,e1,e2))        ::stk,  OP(mulop)             ::toks )

   (* Always shift an Expt token *)
   | step(               Exp(e2)::Tok(OP(binop))::Exp(e1)::stk,  OP(Expt)::toks ) =
     step( Tok(OP(Expt))::Exp(e2)::Tok(OP(binop))::Exp(e1)::stk,            toks ) (*shift*)

   (* All other configurations are ill-defined *)
   | step( stk, toks ) =
                  raise Fail ("Unexpected configuration:"
                         ^ "\nstack = " ^ (ListUtils.listToString stkvalToString stk)
                         ^ "\ntoken = " ^ (ListUtils.listToString Token.toString toks))

 (* top-level function to convert string to exp *)
 fun stringToExp str = step([], Scanner.stringToTokens str)
```


A Parsing Table for G_{RoseTree}

G_{RoseTree}

0	S′	→	S $
1	S	→	(L)
2	S	→	x
3	L	→	S
4	L	→	L , S

top of stack states	(	)	x	,	$	S	L
	tokens					variables	
1: ⊥	s3		s2			g4	
2: x	r2	r2	r2	r2	r2		
3: (	s3		s2			g7	g5
4: ⊥ S					a		
5: (L		s6		s8			
6: (L)	r1	r1	r1	r1	r1		
7: (S	r3	r3	r3	r3	r3		
8: (L ,	s3		s2			g9	
9: (L , S	r4	r4	r4	r4	r4		

sk shifts configuration … (*i*: … γ) ! t T to … (*k*: γ t) ! T

rn reduces configuration … (*i*: … γ) (*j*: δ) ! T to … (*k*: … γ V) ! T, where the *n*th production is V → δ and in state *i*, V **goes to** *k* via g*k*

a **accepts** the configuration ⊥ S ! $, where S is the "real" start symbol

More About Parsing Tables

$G_{RoseTree}$

0 S' → S $
1 S → (L)
2 S → x
3 L → S
4 L → L , S

top of stack states	tokens (	)	x	,	$	variables S	L
1: ⊥	s3		s2			g4	
2: x	r2	r2	r2	r2	r2		
3: (	s3		s2			g7	g5
4: ⊥ S					a		
5: (L		s6		s8			
6: (L)	r1	r1	r1	r1	r1		
7: (S	r3	r3	r3	r3	r3		
8: (L ,	s3		s2			g9	
9: (L , S	r4	r4	r4	r4	r4		

Stack states can be determined by an FA reading stack elements bottom up.

This table is LR(0) because the decision about whether to shift or reduce is based only on stack state.

Many of the reduce entries in this table are bogus (no valid configuration)

Table-Guided LR Parsing

	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

$_1$! (x , (x)) $
⇒ $_1$ ($_3$! x , (x)) $
⇒ $_1$ ($_3$ x $_2$! , (x)) $
⇒ $_1$ ($_3$ S $_7$! , (x)) $ reduce by rule 2: S → x
⇒ $_1$ ($_3$ L $_5$! , (x)) $ reduce by rule 3: L → S
⇒ $_1$ ($_3$ L $_5$, $_8$! (x)) $ shift ,
⇒ $_1$ ($_3$ L $_5$, $_8$ ($_3$! x)) $ shift (
⇒ $_1$ ($_3$ L $_5$, $_8$ ($_3$ x $_2$!)) $ shift x
⇒ $_1$ ($_3$ L $_5$, $_8$ ($_3$ S $_7$!)) $ reduce by rule 2: S → x
⇒ $_1$ ($_3$ L $_5$, $_8$ ($_3$ L $_5$!)) $ reduce by rule 3: L → S
⇒ $_1$ ($_3$ L $_5$, $_8$ ($_3$ L $_5$) $_6$!) $ shift)
⇒ $_1$ ($_3$ L $_5$, $_8$ S $_9$!) $ reduce by rule 1: S → (L)
⇒ $_1$ ($_3$ L $_5$!) $ reduce by rule 4: L → L , S
⇒ $_1$ ($_3$ L $_5$) ! $ shift)
⇒ $_1$ S $_4$! $ reduce by rule 1: S → (L)
⇒ **accept!**

A hierarchy of grammar classes

- Sandwiched between LR(0) and LR(1) are two categories SLR and LALR(1) that involve particular kinds of tables (which we don't have time to study this semester; see Appel 3.3 for details.
- LALR(1) has become the standard for programming languages and for automatic parser generators.
- There is a parser construction tool, called YACC, that can automate the construction of LALR(1) parsers.