Parsing #2

Leonidas Fegaras
Bottom-up Parsing

- Rightmost derivations; use of rules from right to left
- Uses a stack to push symbols
  - the concatenation of the stack symbols with the rest of the input forms a valid bottom-up derivation

```
E - num
```

```
x-2*y$
```

Derivation: E-num*id$

- Two operations:
  - reduction: if a postfix of the stack matches the RHS of a rule (called a handle), replace the handle with the LHS nonterminal of the rule
    eg, reduce the stack \[ x \ast E + E \] by the rule \[ E ::= E + E \]
    new stack: \[ x \ast E \]
  - shifting: if no handle is found, push the current input token on the stack and read the next symbol

- Also known as shift-reduce parsing
### Example

**Input:** x-2*y$

<table>
<thead>
<tr>
<th>Stack</th>
<th>rest-of-the-input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) id - num * id $</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>2) id - num * id $</td>
<td>reduce by rule 8</td>
<td></td>
</tr>
<tr>
<td>3) F - num * id $</td>
<td>reduce by rule 6</td>
<td></td>
</tr>
<tr>
<td>4) T - num * id $</td>
<td>reduce by rule 3</td>
<td></td>
</tr>
<tr>
<td>5) E - num * id $</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>6) E - num * id $</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>7) E - num * id $</td>
<td>reduce by rule 7</td>
<td></td>
</tr>
<tr>
<td>8) E - F * id $</td>
<td>reduce by rule 6</td>
<td></td>
</tr>
<tr>
<td>9) E - T * id $</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>10) E - T * id $</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>11) E - T * id $</td>
<td>reduce by rule 8</td>
<td></td>
</tr>
<tr>
<td>12) E - T * F $</td>
<td>reduce by rule 4</td>
<td></td>
</tr>
<tr>
<td>13) E - T $</td>
<td>reduce by rule 2</td>
<td></td>
</tr>
<tr>
<td>14) E $</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>15) S</td>
<td>accept (reduce by 0)</td>
<td></td>
</tr>
</tbody>
</table>

0) S ::= E $
1) E ::= E + T
2) E ::= E - T
3) E ::= T
4) T ::= T * F
5) T ::= T / F
6) T ::= F
7) F ::= num
8) F ::= id

0) S ::= E $
1) E ::= E + T
2) E ::= E - T
3) E ::= T
4) T ::= T * F
5) T ::= T / F
6) T ::= F
7) F ::= num
8) F ::= id

0) S ::= E $
1) E ::= E + T
2) E ::= E - T
3) E ::= T
4) T ::= T * F
5) T ::= T / F
6) T ::= F
7) F ::= num
8) F ::= id
Machinery

• Need to decide when to shift or reduce, and if reduce, by which rule
  – use a DFA to recognize handles
• Example
  0) $S ::= R \, S$
  1) $R ::= R \, b$
  2) $R ::= a$
  – state 2: accept (reduce by 0)
  – state 3: reduce by 2
  – state 4: reduce by 1
• The DFA is represented by
  an ACTION and a GOTO table
• The stack contains state numbers now

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>s3</td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>r2</td>
</tr>
<tr>
<td>4</td>
<td>r1</td>
</tr>
</tbody>
</table>
push(0);
read_next_token();
for(;;)
{
    s = top();  /* current state is taken from top of stack */
    if (ACTION[s,current_token] == 'si')  /* shift and go to state i */
    {
        push(i);
        read_next_token();
    }
    else if (ACTION[s,current_token] == 'ri')  /* reduce by rule i: X ::= A1...An */
    {
        perform pop() n times;
        s = top();  /* restore state before reduction from top of stack */
        push(GOTO[s,X]);  /* state after reduction */
    }
    else if (ACTION[s,current_token] == 'a')
        success!!
    else error();
### Example: parsing `abb$`

<table>
<thead>
<tr>
<th>Stack</th>
<th>rest-of-input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>abb$</code></td>
<td><code>s3</code></td>
</tr>
<tr>
<td>0 3</td>
<td><code>bb$</code></td>
<td><code>r2</code> (pop, push GOTO[0,R] since R ::= a)</td>
</tr>
<tr>
<td>0 1</td>
<td><code>bb$</code></td>
<td><code>s4</code></td>
</tr>
<tr>
<td>0 1 4</td>
<td><code>b$</code></td>
<td><code>r1</code> (pop twice, push GOTO[0,R] since R ::= R b)</td>
</tr>
<tr>
<td>0 1</td>
<td><code>b$</code></td>
<td><code>s4</code></td>
</tr>
<tr>
<td>0 1 4</td>
<td><code>$</code></td>
<td><code>r1</code> (pop twice, push GOTO[0,R] since R ::= R b)</td>
</tr>
<tr>
<td>0 1</td>
<td><code>$</code></td>
<td><code>s2</code></td>
</tr>
<tr>
<td>0 1 2</td>
<td></td>
<td><code>accept</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>s3</td>
</tr>
<tr>
<td>b</td>
<td>s4</td>
</tr>
<tr>
<td>$</td>
<td>s2</td>
</tr>
<tr>
<td>S</td>
<td>R</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>r2</td>
</tr>
<tr>
<td>4</td>
<td>r1</td>
</tr>
</tbody>
</table>

### Table:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>s3</td>
</tr>
<tr>
<td>b</td>
<td>s4</td>
</tr>
<tr>
<td>$</td>
<td>s2</td>
</tr>
<tr>
<td>S</td>
<td>R</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>r2</td>
</tr>
<tr>
<td>4</td>
<td>r1</td>
</tr>
</tbody>
</table>
Table Construction

- Problem: given a CFG, construct the finite automaton (DFA) that recognizes handles
- The DFA states are *itemsets* (sets of items)
- An *item* is a rule with a dot at the RHS
  - eg, possible items for the rule $E ::= E + E$:
    
    \begin{align*}
    E & ::= . E + E \\
    E & ::= E . + E \\
    E & ::= E + . E \\
    E & ::= E + E .
    \end{align*}
  - The dot indicates how far we have progressed using this rule to parse the input
    - eg, the item $E ::= E + . E$ indicates that
      1) we are using the rule $E ::= E + E$
      2) we have parsed $E$, we have seen the token $+$, and we are ready to parse another $E$
• The items in a itemset indicate different possibilities that will have to be resolved later by reading more input tokens
  – eg, the itemset:
    \[ T ::= ( E ) \]
    \[ E ::= E \cdot + T \]
  
  corresponds to a DFA state where we don't know whether we are looking at an \( ( E ) \) handle or an \( E + T \) handle
  
  • it will be \( ( E ) \) if the next token is \( ) \)
  • it will be \( E + T \) if the next token is \( + \)

• When the dot is at the end of an item, we have found a handle
  – eg, \[ T ::= ( E ) \]
  
  it corresponds to a reduction by \[ T ::= ( E ) \]

• *reduce/reduce conflict*: an itemset should never have more than one item with a dot at the end
  
  • otherwise we can't choose a handle
Closure of an Itemset

- The closure of an item
  - $X ::= a \cdot t b$ is the singleton set that contains the item $X ::= a \cdot t b$ only
  - $X ::= a \cdot Y b$ is the set consisting of the item itself, plus all rules for $Y$ with the dot at the beginning of the RHS, plus the closures of these items

- eg, the closure of the item $E ::= E + . T$ is the set:
  
  $E ::= E + . T$
  
  $T ::= . T * F$
  
  $T ::= . T / F$
  
  $T ::= . F$
  
  $F ::= . num$
  
  $F ::= . id$

- The closure of an itemset is the union of closures of all items in the itemset
Constructing the DFA

• The initial state of the DFA (state 0) is the closure of the item $S ::= . a$, where $S ::= a$ is the first rule of the grammar.

• For each itemset, if there is an item $X ::= a . s b$ in an itemset, where $s$ is a symbol, we have a transition labeled by $s$ to an itemset that contains $X ::= a s . b$.

• But
  1) if we have more than one item with a dot before the same symbol $s$, say $X ::= a . s b$ and $Y ::= c . s d$, then the new itemset contains both $X ::= a s . b$ and $Y ::= c s . d$
  2) we need to get the closure of the new itemset
  3) we need to check if this itemset has appeared before so that we don't create it again.
Example #1

0) $S ::= R \ 0$
1) $R ::= R \ b$
2) $R ::= a$

```
0

0  S ::= . R $  
    R ::= . R b 
    R ::= . a 

3   R ::= a . 
```

```
1

1  S ::= R . $  
    R ::= R . b 

4  R ::= R b . 
```

```
2

2  S ::= R $ . 
```

```
```
Example #2

0) $S' ::= S \ $$
1) S ::= B B
2) B ::= a B
3) B ::= c
Example #3

S ::= E $
E ::= ( L )
E ::= ( )
E ::= id
L ::= L , E
L ::= E
LR(0)

- If an itemset has more than one reduction (an item with the dot at the end), it is a reduce/reduce conflict
- If an itemset has at least one shifting (an outgoing transition to another state) and at least one reduction, it is a shift/reduce conflict
- A grammar is LR(0) if it doesn't have any reduce/reduce or shift/reduce conflict

1) \( S ::= E \$ \)
2) \( E ::= E + T \)
3) \( \mid T \)
4) \( T ::= T \ast F \)
5) \( \mid F \)
6) \( F ::= \text{id} \)
7) \( \mid (E) \)

\[
\begin{align*}
S & ::= . E \$ \\
E & ::= . E + T \\
T & ::= . T \\
T & ::= . F \\
F & ::= . \text{id} \\
F & ::= . (E)
\end{align*}
\]
SLR(1)

- There is an easy fix for some of the shift/reduce or reduce/reduce errors
  - requires to look one token ahead (called the lookahead token)
- Steps to resolve the conflict of an itemset:
  1) for each shift item $Y ::= b \cdot c$ you find FIRST(c)
  2) for each reduction item $X ::= a \cdot$ you find FOLLOW(X)
  3) if each FOLLOW(X) do not overlap with any of the other sets, you have resolved the conflict!
- eg, for the itemset with $E ::= T \cdot$ and $T ::= T \cdot \ast \ F$
  - FOLLOW(E) = \{ $, +, ) \}$
  - FIRST(\ast \ F) = \{ \ast \}$
  - no overlapping!
- This is a SLR(1) grammar, which is more powerful than LR(0)
LR(1)

- The SLR(1) trick doesn't always work

\[
S ::= E \$
E ::= L = R \quad | \quad R
L ::= * R \quad | \quad \text{id}
R ::= L
\]

- For a reduction item \( X ::= a \) we need a smaller (more precise) set of lookahead tokens to tell us when to reduce
  - called expected lookahead tokens
  - must be a subset of or equal to FOLLOW(\( X \)) (hopefully subset)
  - they are context-sensitive \( \Rightarrow \) finer control
LR(1) Items

• Now each item is associated with expected lookahead tokens
  – eg, \( L ::= * . R \quad =\$
    
    the tokens = and $ are the expected lookahead tokens
    the are only useful in a reduction item:
      \( L ::= * R . \quad =\$
    
    it indicates that we reduce when the lookahead token from input is = or $

• LR(1) grammar: at each shift/reduce reduce/reduce conflict, the
  expected lookahead tokens of the reductions must not overlap
  with the first tokens after dot (ie, the FIRST(c) in \( Y ::= b . c \))

• Rules for constructing the LR(1) DFA:
  – for a transition from \( A ::= a . s b \) by a symbol s, propagate the expected
    lookaheads
  – when you add the item \( B ::= . c \) to form the closure of \( A ::= a . B b \)
    with expected lookahead tokens t, s, ..., the expected lookahead tokens of
    \( B ::= . c \) are \( \text{FIRST}(bt) \cup \text{FIRST}(bs) \cup ... \)
Example

S ::= E $
E ::= L = R
     | R
L ::= * R
     | id
R ::= L

S ::= . E $
E ::= . L = R $
E ::= . R $
L ::= . * R =$
L ::= . id =$
R ::= . L $
LALR(1)

• If the lookaheads $s_1$ and $s_2$ are different, then the items
  \[ A ::= a \ s_1 \] and \[ A ::= a \ s_2 \] are different
  – this results to a large number of states since the combinations of expected
    lookahead symbols can be very large.

• We can combine the two states into one by creating an item
  \[ A ::= a \ s_3 \] where $s_3$ is the union of $s_1$ and $s_2$

• LALR(1) is weaker than LR(1) but more powerful than SLR(1)

• LALR(1) and LR(0) have the same number of states

• Easy construction of LALR(1) itemsets: start with LR(0) items
  and propagate lookaheads as in LR(1)
  – don't create a new itemset if the LR(0) items are the same
    • just union together the lookaheads of the corresponding items
  – you may have to propagate lookaheads by looping through the same
    itemsets until you cannot add more

• Most parser generators are LALR(1), including CUP
Example

S ::= E $
E ::= E + E
| E * E
| ( E )
| id
| num

\[
\begin{align*}
S &::= E $ \\
E &::= E + E \\
&\quad | E * E \\
&\quad | ( E ) \\
&\quad | id \\
&\quad | num
\end{align*}
\]
Practical Considerations

• How to avoid reduce/reduce and shift/reduce conflicts:
  – left recursion is good, right recursion is bad
  • left recursion uses less stack than right recursion
  • left recursion produces left associative trees
  • right recursion produces right associative trees

\[ L ::= \text{id} \mid \text{id} \mid \text{id} \]

• Most shift/reduce errors are easy to remove by assigning precedence and associativity to operators

\[ S ::= E \]
\[ E ::= E + E \mid E * E \mid ( E ) \mid \text{id} \mid \text{num} \]

\[ + \text{ and } * \text{ are left-associative} \]
\[ * \text{ has higher precedence than } + \]
Practical Considerations (cont.)

• How precedence and associativity work?
  – the precedence and associativity of a rule comes from the last terminal at
    the RHS of the rule
  eg, the rule  \( E ::= E + E \) has the same precedence and associativity as +
  – you can force the precedence of a rule in CUP:
    • eg.  \( E ::= \text{MINUS} \ E \ \%\text{prec} \ \text{UMINUS} \)
  – in a state with a shift/reduce conflict and you are reading a token \( t \)
    • if the precedence of \( t \) is lower than that of the reduction rule, you reduce
    • if the precedence of \( t \) is equal to that of the reduction rule,
      – if the rule has left associativity, you reduce
      – otherwise you shift
    • otherwise you shift

• Reduce/reduce conflicts are hopeless
  – the parser generator always reduces using the rule listed first
  – fatal error
Error Recovery

- All the empty entries in the ACTION and GOTO tables correspond to syntax errors.
- We can either
  - report it and stop the parsing
  - continue parsing finding more errors (error recovery)
- Error recovery in CUP:
  
  \[
  S ::= \text{L = E ;} \\
  \mid \text{\{} \text{SL} \text{\};} \\
  \mid \text{error ;} \\
  \]

  \[
  \text{SL ::= S ;} \\
  \mid \text{SL S ;}
  \]

- In case of an error, the parser pops out elements from the stack until it finds an error state where it can proceed.
- then it discards tokens from the input until a restart is possible.
The Calculator Parser

terminal LP, RP, COMMA, SEMI, ASSIGN, IF, THEN, ELSE, AND, OR, NOT, QUIT, PLUS, TIMES, MINUS, DIV, EQ, LT, GT, LE, NE, GE, FALSE, TRUE, DEFINE;

terminal String ID;
terminal Integer INT;
terminal Float REALN;
terminal String STRINGT;
non terminal exp, string, name;
non terminal expl, names;
non terminal item, prog;
precedence nonassoc ELSE;
precedence right OR;
precedence right AND;
precedence nonassoc NOT;
precedence left EQ, LT, GT, LE, GE, NE;
precedence left PLUS, MINUS;
precedence left TIMES, DIV;
The Calculator Parser (cont.)

start with prog;
prog ::= item SEMI
    | prog item SEMI
    ;
item ::= exp
    | QUIT
    | ID ASSIGN exp
    | DEFINE ID LP names RP EQ exp
    ;
name ::= ID
    ;
string ::= STRINGT
    ;
expl ::= expl COMMA exp
    | exp
    ;
names ::= names COMMA name
    | name
    ;
The Calculator Parser (cont.)

exp ::= INT
    | REALN
    | TRUE
    | FALSE
    | name
    | string
    | LP exp RP
    | IF exp THEN exp ELSE exp
    | exp EQ exp
    | exp LT exp
    | exp GT exp
    | exp LE exp
    | exp NE exp
    | exp GE exp
    | exp PLUS exp
    | exp MINUS exp
    | exp TIMES exp
    | exp DIV exp
    | exp OR exp
    | exp AND exp
    | NOT exp
    | name LP exp RP