CS 230
Programming Languages

09 / 22 / 2022

Instructor: Michael Eckmann

Today’s Topics

- Questions? / Comments?
- Parsers
  - Top-down, recursive descent parsers
  - Bottom up parser example with predefined table

Recursive descent example

- So called because subroutines/functions (one for each non-terminal) may be recursive and descent because it is a top-down parser.
- EBNF is well suited for these parsers because EBNF reduces the number of non-terminals (as compared to plain BNF) which reduces the number of subroutines/functions.
- A grammar for simple expressions:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | ( <expr> )
```

Recursive descent example

- Please note: the parens in the first two productions are EBNF symbols
- The parens in the <factor> production are actually in the language being described by the grammar.

Recursive descent example

- Assume we have a lexical analyzer named **`lex`**, which puts the next token code in **`nextToken`**
- The coding process when there is only one RHS:
  - For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error
  - For each nonterminal symbol in the RHS, call its associated parsing subroutine/function

Recursive descent example

```
/* Function expr (written in the C language)
   Parses strings in the language generated by the
   rule:   <expr> → <term> {(+ | -) <term>}          */

void expr() {

   /* Parse the first term */

   term();  /* term() is it's own function
               representing the non-terminal <term>*/

/* As long as the next token is + or -, call lex to
   get the next token, and parse the next term */

    while (nextToken == PLUS_OP || nextToken == MINUS_OP)
    {
        lex();
        term();
    }
 }
```

Recursive descent example

```
/* Function term
   Parses strings in the language generated by the
   rule:   <term> → <factor> {(* | /) <factor>}           */

void term() {

   /* Parse the first factor */

   factor();  /* factor() is it's own function
               representing the non-terminal <factor>*/

/* As long as the next token is * or /, call lex to
   get the next token, and parse the next term */

    while (nextToken == MULT_OP || nextToken == DIV_OP)
    {
        lex();
        factor();
    }
 }
```

Recursive descent example

- ***<u>Note: by convention each subroutine leaves the next token to be processed in nextToken</u>***
- A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse
  - The correct RHS is chosen on the basis of the next token of input (the lookahead)
  - The next token is compared with the first token that can be generated by each RHS until a match is found
  - If no match is found, it is a syntax error
- Let's now look at an example of how the parser would handle multiple RHS's.

Recursive descent example

```
/* Function factor
   Parses strings in the language generated by the
   rule:    <factor> -> id  |  (<expr>)  */
 void factor()
 {
   if (nextToken == IDENT)
   /* For the RHS id, call lex to get the next token*/

      lex();

   /* If the RHS is ( <expr> ) - call lex to pass over
      the leftparenthesis, call expr, and check for
      the right parenthesis */

   else if (nextToken == LEFT_PAREN)
   {
     lex();
     expr();
     if (nextToken == RIGHT_PAREN)
        lex();
     else
        error();
   }

  else error(); /* Neither RHS matches */

 }
```

Recursive descent parsers

- Tips when implementing a recursive descent parser from an EBNF description.
  - Each non-terminal becomes a function / method / subroutine, and can be written without worrying (independently) where it was called from
  - Each of these functions assumes lex has been called prior to it and so a nextToken is already available for it to consider. Idea is to call lex at beginning of the program once AND just after checking / processing a terminal / token

Recursive descent parsers

- Let's “execute” the three methods of code we just examined assuming the main program does the following (for a couple of inputs):

```
concatenate a terminationSymbol to end of input
lex();
expr();
if (nextToken is terminationSymbol)
        syntactically correct
else
        error(“extra stuff after valid program”);
```

Bottom Up parsers

- Bottom-up parsers
  - Given a right sentential form, α, determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation
  - This substring is called the handle
  - This is backwards from the way top down parsing works
  - The most common bottom-up parsing algorithms are in the LR family (L=left-to-right scan of input, R= rightmost derivations)

Bottom Up parsers

- Bottom-up parsers
  - The whole process is: Starting with the input sentence (all terminal symbols) (also this is the last sentential form in a derivation), produce the sequence of sentential forms (up) until all that remains is the start symbol.
  - One step in the process of bottom-up parsing is: given a right sentential form, find the correct RHS to reduce to get the previous right-sentential form in the derivation
  - What are we reducing a RHS to?

Bottom Up parsers

Example:

E -> E + T | T

T -> T * F | F

F -> (E) | id

Bottom Up parsers

Example:

E -> E + T | T

T -> T * F | F

F -> (E) | id

- Note: Sometimes it is not obvious which RHS to reduce in each step. For instance, the right-sentential form E + T * id includes more than one RHS. Which RHSs are included in this sentential form?
- And why can we not choose the others?

Bottom Up parsers

- The correct choice of the RHS to reduce in any given step is called the handle (of that right-sentential form.)
- The text contains intuition on how to find the handle --- we won't cover that here. Just know that the handle of any right-sentential form is unique.

Bottom Up parsers

- Most bottom-up parsers are LR --- what is LR again?
- They use small programs and a parsing table.

Bottom Up parsers

- LR parsers consist of
  - A parse stack
  - An input string (the sentence to be determined if it is syntactically correct)
  - A parse table created from the grammar beforehand. Its rows are states, and its columns are terminal and nonterminal symbols.
  - So, given an input symbol and a state, the program will lookup in the table what to do.

arithmetic expression grammar (same one as on earlier slide)

- 1. E -> E + T
- 2. E -> T
- 3. T -> T * F
- 4. T -> F
- 5. F -> (E)
- 6. F -> id

- For next slide, R means reduce, S means shift. e.g. R4 means *reduce using production 4* above. S6 means *shift the next symbol of input onto the stack and push state 6 onto the stack*.

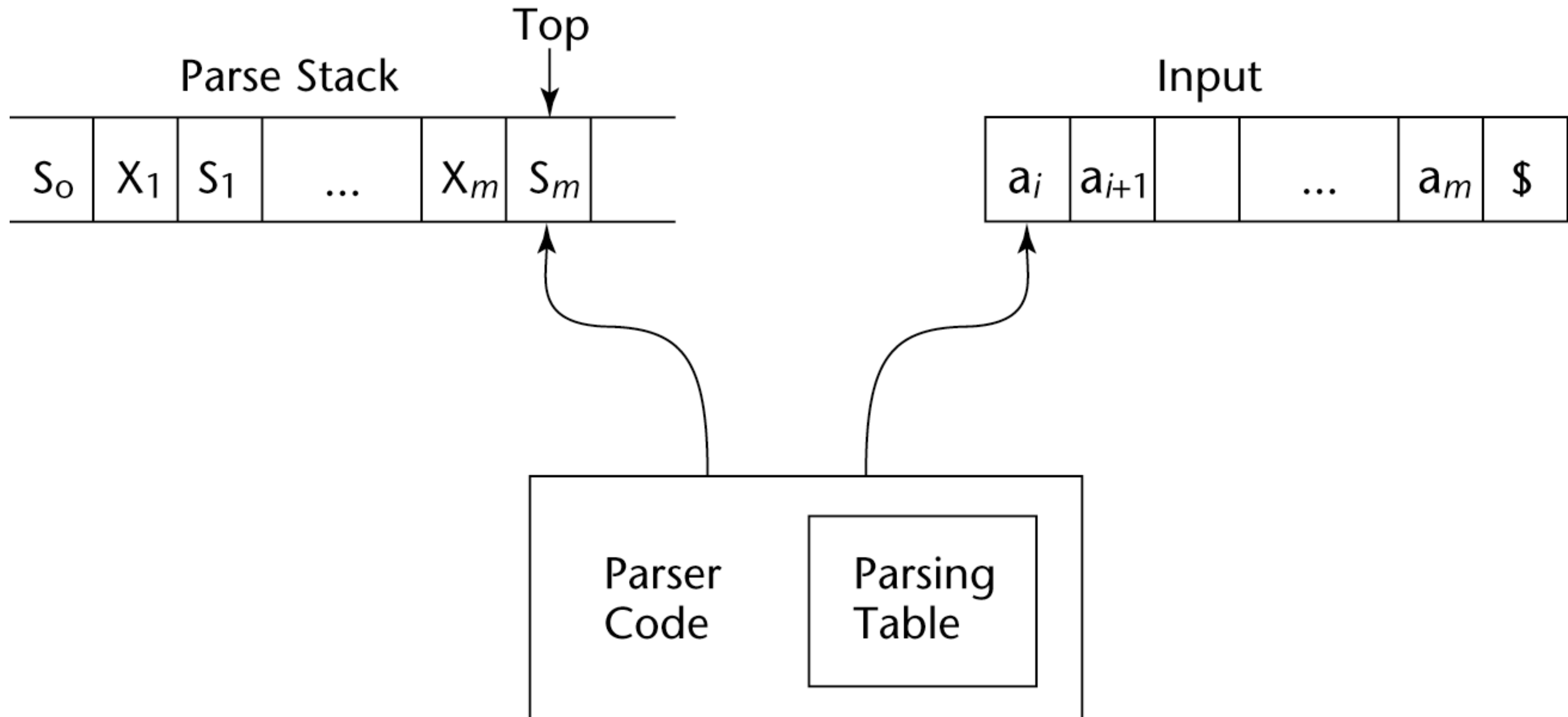
LR parsing table for arithmetic expression grammar the S4 in row: State 0, should be under (not *

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

ACTION & GOTO

- The GOTO portion of the table is used to determine which state to push onto the stack after a reduction.
- The ACTION portion of the table is used to determine whether to shift or reduce based on the current state and next input symbol.
- Blank cells in the table imply syntax errors.
- accept means the sentence is syntactically correct.
- The stack starts with only state 0 on it.
- The input string is the complete sentence followed by a termination symbol, usually a $.

LR Parser structure



- Top of stack is to the right, the next input symbol for the input is the leftmost symbol. S's are State #'s and X's are grammar symbols.

Let's go through an example parse

- Before we go through an example parse a few things should be stated.

- Recall that a shift pushes the next input symbol on the stack and then pushes the specified state on the stack as well. That's pretty straightforward.

Let's go through an example parse

- A reduce is more complex. When a reduce occurs,

- 1. the handle (the whole RHS) is popped off the stack (along with each state per symbol in the handle)
- 2. then the LHS is pushed onto the stack
- 3. followed by another state pushed onto the stack. The state to be pushed is determined by the GOTO portion of the parse table.
  - column is the LHS just pushed
  - row is the state that was on the top of the stack after the handle and its associated states were popped.

Let's go through an example parse

- Now we're ready to go through an example. We'll determine if

id + id * id

is syntactically correct for the example grammar a few slides ago.

We'll do this by viewing the table on screen and I'll show the stack and input and action on the board.

Also, I'll write on the board what happens during a shift and a reduce.

Let's go through an example parse

- How about we determine if

(id) id + id

is syntactically correct for the example grammar a few slides ago.

We'll do this by viewing the table on screen and I'll show the stack and input and action on the board.

Also, I'll write on the board what happens during a shift and a reduce.