CSE 304/504 Compiler Design

Top-Down Parsing (Predictive Parsing)

YOUNGMIN KWON / TONY MIONE

Parsing

Parsing is the process of determining if a string of tokens can be generated by a grammar

For any context-free grammar, there is a parser that can parse a string of n tokens in $O(n^3)$ times.

For programming languages, we can generally construct a grammar that can be parsed quickly (in linear time).

Top-Down parsing

- Build parse trees from the root node to leave nodes.
- Simple (parsers can be made manually), but limited.

Bottom-Up parsing

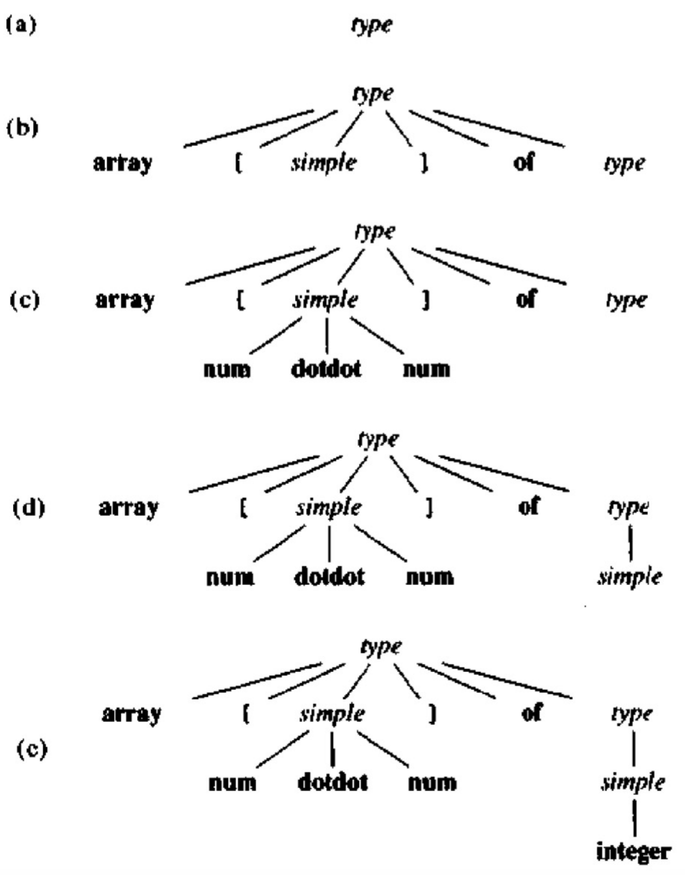
- Build parse trees from leaves towards the root.
- More complex (parsers are generated from software tools), but more generic.

Top-Down Parsing

array [num dotdot num] of integer

Start from the root, labeled with the starting nonterminal, repeatedly perform the following two steps.

- At node n, labeled with nonterminal A, select one of the productions for A and construct children at n for the symbols on the RHS of the production.
- Find the next node at which a subtree is to be constructed.



$$
\begin{aligned}
\textit{type} &\rightarrow \textit{simple} \\
&\mid\ \uparrow \textbf{id} \\
&\mid\ \textbf{array}\ [\ \textit{simple}\ ]\ \textbf{of}\ \textit{type} \\
\textit{simple} &\rightarrow \textbf{integer} \\
&\mid\ \textbf{char} \\
&\mid\ \textbf{num dotdot num}
\end{aligned}
$$

Predictive Parsing

Recursive Decent Parsing
- A top-down parsing method.
- For each nonterminal of a grammar, associate a procedure and execute it to process the input.

Predictive Parsing
- A recursive decent parsing method.
- The lookahead symbol unambiguously determines the procedure for each nonterminal.
- In the next example, we use an additional procedure match to advance the next input token if the argument matches the lookahead symbol.

Pseudo-code for a predictive parser

```
procedure type ;
begin
    if lookahead is in { integer, char, num } then
        simple
    else if lookahead = '↑' then begin
        match('↑'); match(id)
    end
    else if lookahead = array then begin
        match(array); match('['); simple ; match(']'); match(of); type
    end
    else error
end;
```

```
procedure simple ;
begin
    if lookahead = integer then
        match(integer)
    else if lookahead = char then
        match(char)
    else if lookahead = num then begin
        match(num); match(dotdot); match(num)
    end
    else error
end;

procedure match(t: token);
begin
    if lookahead = t then
        lookahead := nexttoken
    else error
end;
```

Predictive Parsing: procedure FIRST

Predictive parsing relies on what first symbols can be generated by the RHS of a production.

FIRST(α)

- Let α be the RHS of a production for nonterminal A
- FIRST(α) returns the set of tokens that appear as the first symbol of the strings generated from α.
- For recursive decent parsing without backtracking, if there are more than one productions, their FIRST sets must be disjoint.
  - E.g. for A -> α | β, FIRST(α) ∩ FIRST(β) = ∅
- Example:

$$\begin{aligned} \text{FIRST}(\textit{simple}) &= \{\ \textbf{integer},\ \textbf{char},\ \textbf{num}\ \} \\ \text{FIRST}(\uparrow \textbf{id}) &= \{\ \uparrow\ \} \\ \text{FIRST}(\textbf{array}\ [\ \textit{simple}\ ]\ \textbf{of}\ \textit{type}) &= \{\ \textbf{array}\ \} \end{aligned}$$

Designing a Predictive Parser

The procedures for nonterminals do two things

1. Decide which production to use by looking at the lookahead and FIRST(α).
   - If there are conflicts, we cannot parse the grammar with this parsing method.
   - If lookahead doesn't appear in any of the FIRST sets, use the ϵ-Production.
2. Procedures mimic the RHS of a production
   - Nonterminals result in a call to the procedure for the nonterminal.
   - Tokens matching the lookahead results in reading the next input.
   - If the token does not match the lookahead, an error is declared.

Designing a Predictive Parser: Extension to a syntax directed translation

1. Construct a predictive parser, ignoring the actions in productions
2. Copy the action from the translation scheme to the parser
   - If an action appears after a grammar symbol X, copy the action after implementing X.
   - If an action appears at the beginning of a production, copy it before implementing the production.

Left Recursion

A problem with left-recursive grammars

- Infinite recursion will occur in recursive decent parsers.
- `expr -> expr + term`
- The leftmost symbol on the RHS is the same as the LHS of the production
- The parser may look like

```
procedure expr;
begin
    if lookhaed is in FIRST('expr + term') then
    begin
        expr; match('+'); term;
    end
end
```

Fixing the Left Recursion Problem

Change Left Recursive Grammar to Right Recursive one

- A -> A α | β
- A -> β R
  R -> α R | ϵ

Example

- expr -> expr + term | term
- A = expr, α = + term, β = term
- expr -> term rest
  rest -> + term rest | ϵ

Adapting the Translation Scheme

If semantic actions are in left recursive productions, carry them along in the production

Example

```
expr -> expr + term { print('+') }
expr -> expr - term { print('-') }
expr -> term
term -> 0 { print('0') }
…
term -> 9 { print('9') }
```

=>

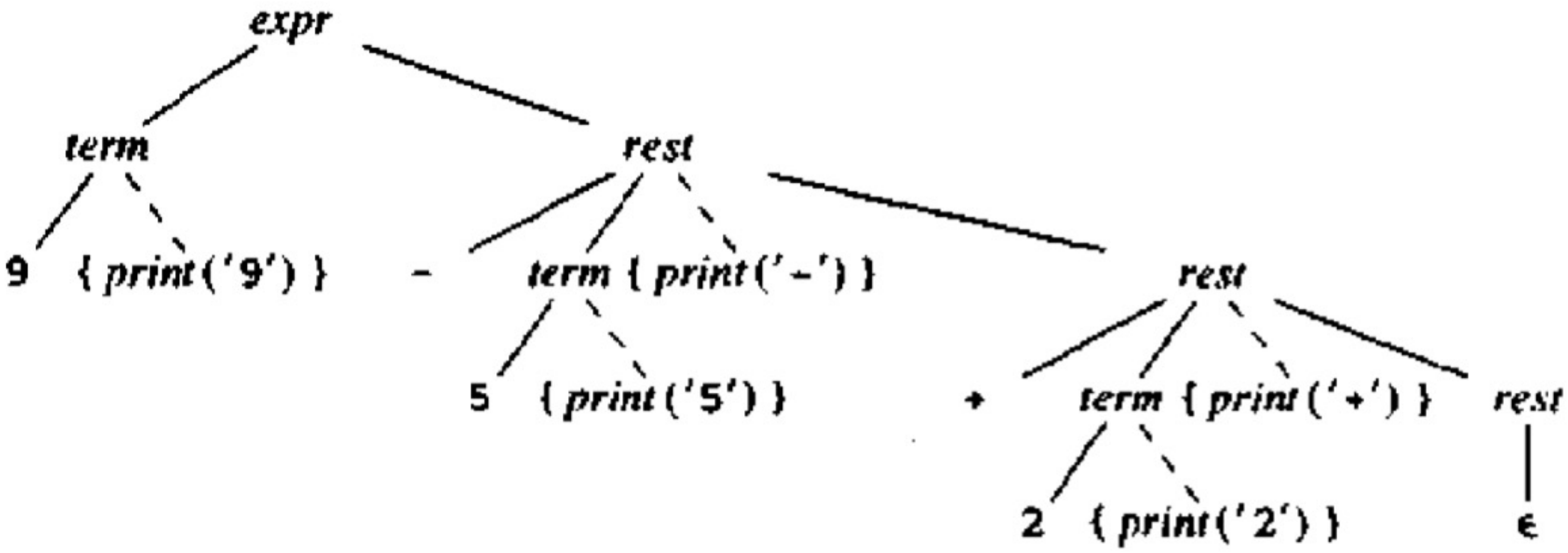
```
expr -> term rest
rest -> + term { print('+') } rest
rest -> - term { print('-') } rest
rest -> ϵ
term -> 0 { print('0') }
…
term -> 9 { print('9') }
```

- A -> A α | A β | γ
- A -> γ R
  R -> α R | β R | ϵ

A = expr,
α = + term { print('+') },
β = - term { print('-') },
γ = term

Adapting the Translation Scheme

Example: Translation of 9-5+2 into 95-2+



Procedures for expr, term, and rest

```
expr()
{
      term(); rest();
}

rest()
{
      if (lookahead == '+') {
            match('+'); term(); putchar('+'); rest();
      }
      else if (lookahead == '-') {
            match('-'); term(); putchar('-'); rest();
      }
      else ;
}
```

```
term()
{
      if (isdigit(lookahead)) {
            putchar(lookahead); match(lookahead);
      }
      else error();
}
```

Lexical Analyzer

It converts input to a stream of tokens.

Lexeme: a sequence of input characters that comprise a single token.

Insulates parser from the lexeme representation of tokens.

Frees parsers from removing white space and comments.

- Removing white spaces from the grammar can be unnecessarily complex.

For numbers, return num and its value as an attribute.

For identifiers, return id and its symboltable entry as an attribute.

For keywords, need to check if a lexeme is a keyword or an identifier.

- Easier if the keywords are reserved.

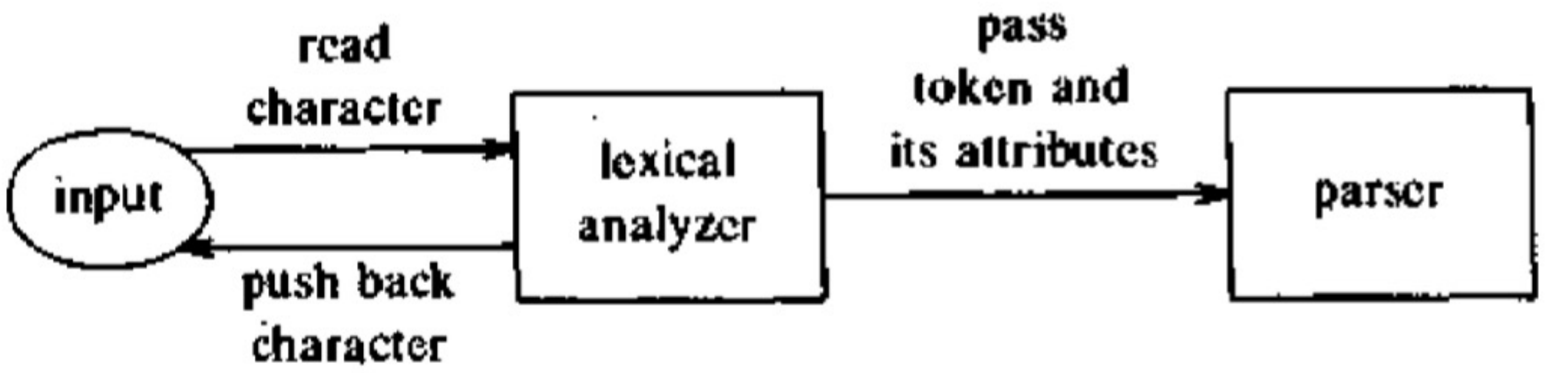
Interface to Lexical Analyzer

In some situations, the lexical analyzer has to read some characters ahead before it can decide.

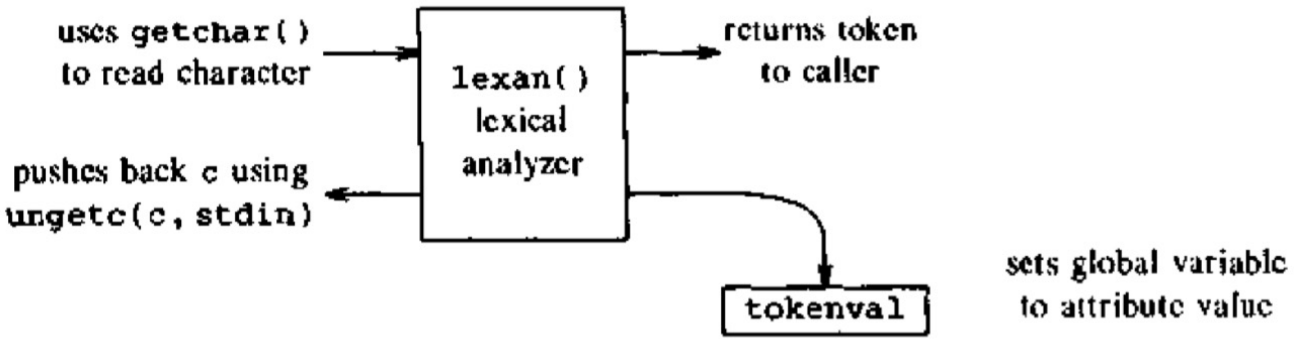
- e.g. to distinguish >= and >, after reading > the lexical analyzer needs to read one more character.
- The extra characters have to be pushed back onto the input.

The parser holds the produced tokens and their attributes in a token buffer.

- Commonly the buffer holds just one token and a procedure call from the parser to the lexical analyzer would work.



A Lexical Analyzer



Updating the grammar and semantic actions for the factor

- factor -> (expr)
         | NUM { print (NUM.value) }

Procedure for factor

```
factor()
{
	if (lookahead == '(') {
		match('('); expr(); match(')');
	}
	else if (lookahead == NUM) {
		printf(" %d ", tokenval); match(NUM);
	}
	else error();
}
```

Symbol Table

Stores information about various source language constructs.

- lexeme for the id,
- type of the id (e.g. procedure, variable, label),
- its position in storage, …

Interface

- `insert(s, t):` returns index of the new entry for string s, token t.
- `lookup(s):` returns index of the entry for string s, or an invalid index if s is not found.

Questions?