CMPE 152: Compiler Design

November 21 Lab

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak

www.cs.sjsu.edu/~mak

Top-Down Parsers

- The parser we hand-wrote for the Pascal interpreter and the parser that ANTLR generates are top-down.
- Start with the topmost nonterminal grammar symbol such as <PROGRAM> and work your way down recursively.
  - Top-down recursive-descent parser
  - Easy to understand and write, but are generally BIG and slow.

Top-Down Parsers, *cont'd*

- Write a parse method for a production (grammar) rule.
- Each parse method "expects" to see tokens from the source program that match its production rule.
  - Example: **`IF`** ... **`THEN`** ... **`ELSE`**
- A parse method calls other parse methods that implement lower production rules.
  - Parse methods consume tokens that match the production rules.

Top-Down Parsers, *cont'd*

- A parse is successful if it's able to derive the input string (i.e., the source program) from the production rules.
- All the tokens match the production rules and are consumed.

Bottom-Up Parsers

- A popular type of bottom-up parser is the shift-reduce parser.
  - A bottom-up parser starts with the input tokens from the source program.
- A shift-reduce parser uses a parse stack.
  - The stack starts out empty.
  - The parser shifts (pushes) each input token (terminal symbol) from the scanner onto the stack.

Bottom-Up Parsers, *cont'd*

- When what's on top of the parse stack matches the longest right hand side of a production rule:
- The parser pops off the matching symbols and …
- … reduces (replaces) them with the nonterminal symbol at the left hand side of the matching rule.
- Example: <term> ::= <factor> * <factor>
  - Pop off <factor> * <factor> and replace by <term>

Bottom-Up Parsers, *cont'd*

- Repeat until the parse stack is reduced to the topmost nonterminal symbol.
  - Example: <PROGRAM>
- The parser accepts the input source as being syntactically correct.
  - The parse was successful.

Example: Shift-Reduce Parsing

- Parse the expression **a + b*c** given the production rules:

<expression> ::= <simple expression>
<simple expression> ::= <term + <term>
<term> ::= <factor> | <factor> * <factor>
<factor> ::= <variable>
<variable> ::= <identifier>
<identifier> ::= a | b | c

In this grammar, the topmost nonterminal symbol is <expression>

Parse stack (top at right)	Input	Action
	a + b*c	shift
a	+ b*c	reduce
<identifier>	+ b*c	reduce
<variable>	+ b*c	reduce
<factor>	+ b*c	reduce
<term>	+ b*c	shift
<term> +	b*c	shift
<term> + b	*c	reduce
<term> + <identifier>	*c	reduce
<term> + <variable>	*c	reduce
<term> + <factor>	*c	shift
<term> + <factor> *	c	shift
<term> + <factor> * c		reduce
<term> + <factor> * <identifier>		reduce
<term> + <factor> * <variable>		reduce
<term> + <factor> * <factor>		reduce
<term> + <term>		reduce
<simple expression>		reduce
<expression>		accept

Why Bottom-Up Parsing?

- The shift-reduce actions can be driven by a table.
  - The table is based on the production rules.
  - It is almost always generated by a compiler-compiler.
- Like a table-driven scanner, a table-driven parser can be very compact and extremely fast.
- However, for a significant grammar, the table can be nearly impossible for a human to follow.

Why Bottom-Up Parsing?

- Error recovery can be especially tricky.
- It can be very hard to debug the parser if something goes wrong.
- It's usually an error in the grammar (of course!).