CS 153: Concepts of Compiler Design

December 5 Class Meeting

Department of Computer Science
San Jose State University

Fall 2017
Instructor: Ron Mak

www.cs.sjsu.edu/~mak

Presentation Schedule

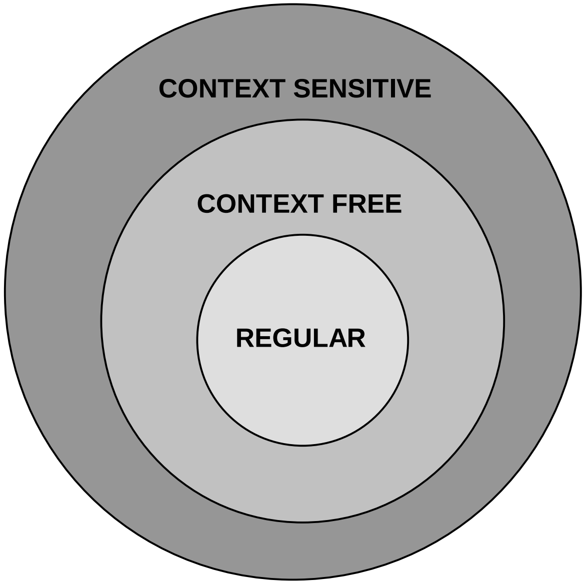
- Thursday, Dec. 7
  - Alex Kong
  - No Name 1
  - No Name 3
  - No Name 4

Final Exam

- Tuesday, December 19
  - 7:15-9:30 **AM** in MH 222
- It will be similar to the midterm.
  - Covers the entire semester.
  - Emphasis on the second half.

Context-Free Grammars

- Every production rule has a <u>single nonterminal</u> for its left-hand side.
  - Example: <simple expression> ::= <term> + <term>
- Whenever the parser matches the right-hand side of the rule, it can <u>freely reduce</u> it to the nonterminal symbol.
  - <u>Regardless of the context</u> of where the match occurs.
- A language is context-free if it can be defined by a context-free grammar.
- Context-free grammars are a subset of context-sensitive grammars.



Context-Sensitive Grammars

- Context-sensitive grammars are more powerful than context-free grammars.
  - They can define more languages.

- Production rules can be of the form

  <A><B><C> ::= <A><b><C>

  - The parser is allowed to reduce <b> to <B> only in the context of <A> and <C>.

Context-Sensitive Grammars: Example

<A><B><C> ::= <A><b><C>

- We can attempt to capture the language rule:
  - "An identifier must have been previously declared to be a variable before it can appear in an expression."
- In an expression, the parser can reduce
  - <identifier> to <variable>

  only in the context of a prior <variable declaration> for that identifier.

Context-Sensitive Grammars, *cont'd*

- Context-sensitive grammars are <u>extremely unwieldy</u> for writing compilers.
- Alternative:
  Use context-free grammars and rely on <u>semantic actions</u> such as building symbol tables to provide the context.

Top-Down Parsers

- The parser we hand-wrote for the Pascal interpreter and the parser that JavaCC 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 <u>consume tokens</u> that match the production rules.

Top-Down Parsers, *cont'd*

- A parse is <u>successful</u> if it's able to <u>derive the input string</u> (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!).

Lex and Yacc

- Lex and Yacc
  - "Standard" compiler-compiler for Unix and Linux systems.
- Lex automatically generates a scanner written in C.
  - Flex: free GNU version
- Yacc ("Yet another compiler-compiler") automatically generates a parser written in C.
  - Bison: free GNU version
  - Generates a bottom-up shift-reduce parser.

Example: Simple Interpretive Calculator

- Yacc file (production rules): **calc.y**

We'll need to define the **NUMBER** token.

```
...
%token NUMBER
%left '+' '-' /* left associative, same precedence */
%left '*' '/' /* left associative, higher precedence */

%%

exprlist: /* empty list */
    | exprlist '\n'
    | exprlist expr '\n' {printf("\t%lf\n", $2);}
    ;

expr: NUMBER         {$$ = $1;}
    | expr '+' expr {$$ = $1 + $3;}
    | expr '-' expr {$$ = $1 - $3;}
    | expr '*' expr {$$ = $1 * $3;}
    | expr '/' expr {$$ = $1 / $3;}
    | '(' expr ')'  {$$ = $2;}
    ;

%%
```

```
#include <stdio.h>
#include <ctype.h>

int main(int argc, char *argv[])
{
    progname = argv[0];
    yyparse();
}
```

Example: Simple Calculator, *cont'd*

- Lex file (token definitions): **calc.l**

```
%{
#include "calc.tab.h"
extern lineno;
%}
%option noyywrap

%%

[ \t]                       {;}  /* skip blanks and tabs */
[0-9]+\.?|[0-9]*\.[0-9]+ {sscanf(yytext, "%lf", &yylval); return NUMBER;}
\n                          {lineno++; return '\n';}
.                           {return yytext[0];}  /* everything else */
```

- Commands:

```
yacc –d calc.y
lex calc.l
cc –c *.c
cc –o calc *.o
./calc
```

Demo

Course Review

- Lectures and PowerPoint slide sets
- Reading assignments
- Homework assignments
- Compiler project

Course Review

- Good understanding of compiler concepts
  - Front end: parser, scanner, and tokens
  - Intermediate tier: symbol table and parse trees
  - Back end: interpreter and code generator
  - The ANTLR 4 compiler-compiler
- Basic understanding of Pascal

Course Review

- What is the overall architecture of a compiler or an interpreter?
  - What are the source language-independent and -dependent parts?
  - What are the target machine-independent and -dependent parts?
- How can we manage the size and complexity of a compiler or an interpreter during its development?

Course Review

- What are the main characteristics of a top-down recursive-descent parser?
- Of a bottom-up parser?

- What is the basic control flow through an interpreter as a source program is read, translated, and executed?
- Through a compiler for code generation?

Course Review

- How do the various components work with each other?
  - parser ←→ scanner
  - scanner ←→ source program
  - parser ←→ symbol table
  - parser ←→ parse tree
  - executor
code generator ←→ symbol table
parse tree

Course Review

- What information is kept in a symbol table?
  - When is a symbol table created?
  - How is this information structured?
  - How is this information accessed?
- What information is kept in a parse tree?
  - When is a parse tree created?
  - How is this information structured?
  - How is this information accessed?

Course Review

- What is the purpose of the
  - symbol table stack
  - runtime stack
  - runtime display
  - operand stack
  - parse stack

Course Review

- Define or explain
  - syntax and semantics
  - syntax diagrams and BNF
  - syntax error handling
  - runtime error handling
  - type checking

Course Review

- Deterministic finite automaton (DFA)
  - start state
  - accepting state
  - transitions
  - state transition table
  - table-driven DFA scanner

Course Review

- What information is kept in an activation record or stack frame?
  - How is this information initialized?
  - What happens during a procedure or function call?
- How to pass parameters
  - by value
  - by reference
- ... with an interpreter vs. with generated object code.

Course Review

- The Java Virtual Machine (JVM) architecture
- Runtime stack
- Stack frame
  - operand stack
  - local variables array
  - program counter

Course Review

- The Jasmin assembly language instructions
  - explicit operands
  - operands on the stack
  - standard and “short cut”
  - type descriptors

Course Review

- Jasmin assembler directives:
  - `.class`
  - `.super`
  - `.limit`
  - `.field`
  - `.var`
  - `.method`
  - `.line`
  - `.end`

Course Review

- Basic concepts of the ANTLR 4 compiler-compiler
- Tokens specification with regular expressions
- Production rules
  - labelled alternates
- Tree node visitors
  - Overriding visit methods.

Course Review

- Code generation and code templates
  - expressions
  - assignment statements
  - conditional statements
  - looping statements
  - arrays and records

Course Review

- Compiling procedures and functions
  - fields and local variables
  - call and return
  - passing parameters

Course Review

- Multipass compilers
  - type checking pass with the visitor pattern
  - optimization pass
  - code generation pass with the visitor pattern

Course Review

- Integrating Jasmin routines with Java routines
  - Pascal runtime library
- Instruction selection
- Instruction scheduling
- Register allocation
  - spilling values
  - live variables

Course Review

- Optimization for performance
  - constant folding
  - constant propagation
  - strength reduction
  - dead code elimination
  - loop unrolling
  - common subexpression elimination

Course Review

Was this course “deep” enough?