1 Part IV: Parsing

- Bottom-Up Parsing
- Parsing with JavaCUP
- Top-Down Parsing
- Error-Recovery

2 Scanners and Parsers

- Most compilers in practice have both a scanner for the lexical syntax and a parser for the context-free syntax.
- better modularity
- separation of concerns
- Characters ⇒Scanner ⇒Tokens
- Tokens ⇒Parser ⇒Syntax-Tree

3 From EBNF to BNF

For building parsers (especially bottom-up) a BNF grammar is often better, than EBNF. But it's easy to convert an EBNF Grammar to BNF:

- Convert every repetition { E } to a fresh non-terminal X and add
  X = ϵ | X E.
- Convert every option [E] to a fresh non-terminal X and add
  X = ϵ | E.
  (We can convert X = A [E] B. to X = A E B | A B.)
- Convert every group (E) to a fresh non-terminal X and add
  X = E.
- We can even do away with alternatives by having several productions with the same non-terminal.
  X = E | E'. becomes X = E. X = E'.

4 Bottom-Up Parsing

- A bottom-up parser builds a derivation from the terminal symbols, working toward the start symbol.
- It consists of a *stack* and an *input*.
- Four actions:
  - *shift*, which pushes the next token onto the stack
  - *reduce*, removes Y1, ...,Yk, which are the right-hand side of some production X = Y1 ... Yk. From the top of the stack and replaces them by X.
  - *accept*, ends the parser with success.
  - *error*, ends the parser with an error message.
- Question: How does the parser know, which action to invoke.

5 Simple Answer: Operator Precedence

- Suitable for languages of the form Expression = Operand Operator Operand with operands of varying precedence and associativity.
- Principle (token is the next input token):

```
if (token is an operand) shift;
else if (stack does not contain an operator) shift;
else {
    top = (topmost operator of stack);
    if (precedence(top) < precedence(token)) shift;
    else if (precedence(top) > precedence(token)) reduce;
    else if (top and token are right associative) shift;
    else if (top and token are left associative) reduce;
    else error;
}
```


6 The Parser Generator JavaCUP

The original version is from http://www.cs.princeton.edu/ appel/modern/java/CUP/, but we use a local modified version.

- generates a class Parser.java, which contains the parser.
- generates a class Parser.tables, which contains the parsing tables.
- generates a class Tokens.java, which is suitable to be used by the scanner.
- if there are situations, where it wouldn't know, whether to shift or to reduce, it reports a conflict.

7 An Expression Parser in JavaCUP

```
package expression;
import jaco.framework.*;
action code {: :};
parser code {:
        Scanner scanner;
        public Parser(Scanner scanner)
        {
            this.scanner = scanner;
        }
        public void report_error(String msg, Object o)
        {
            /* report error */
        }
```


8 An Expression Parser in JavaCUP (2)

```
        public void report_fatal_error(String msg, Object o)
        {
            /* report error and throw exception */
        }
:};
init with {: :};
scan with {: return scanner.nextToken(); :};
```

9 An Expression Parser in JavaCUP (3)

```
terminal PLUS, MINUS, TIMES, DIV, LPAREN, RPAREN;
terminal NUMLIT;
non terminal Expression, Term, Factor;
start with Expression;

Expression ::= Expression PLUS Term
             | Expression MINUS Term
             | Term
             ;
Term       ::= Term TIMES Factor
             | Term DIV Factor
             | Factor
             ;
Factor     ::= NUMLIT
             | LPAREN Expression RPAREN
             ;
```


10 A shift-reduce Conflict

If we enter the grammar

```
Expression ::= Expression PLUS Expression
             ;
```

without precedence JavaCUP will tell us:

```
*** Shift/Reduce conflict found in state #4
   between Expression ::= Expression PLUS Expression (*)
   and     Expression ::= Expression (*) PLUS Expression
   under symbol PLUS
   Resolved in favor of shifting.
```

Telling JavaCUP that PLUS is left associative helps!

11 Using Precedence

```
terminal PLUS, MINUS, TIMES, DIV, LPAREN, RPAREN;
terminal NUMLIT;
non terminal Expression, Term, Factor;
precedence left PLUS, MINUS;
precedence left TIMES, DIV;
start with Expression;

Expression ::= Expression PLUS Expression
             | Expression MINUS Expression
             | Expression TIMES Expression
             | Expression DIV Expression
             | NUMLIT
             | LPAREN Expression RPAREN
             ;
```


12 Precedence

- A terminal has the given precedence (or lowest if unsepcified)
- a production has the precedence of its last terminal (lowest if unspecified, give if explicitely annotated).
- In a shift/reduce conflict
  - if the production has higher precedence reduce
  - if the terminal has higher precedence shift
  - if they are equal use associativity

13 The if-then-else Problem

A standard problem in parsing is the if-then-else:

```
Statement ::= IF LPAREN Expression RPAREN Statement
            | IF LPAREN Expression RPAREN Statement ELSE Statement
            | ... other statements ...
            ;
```

reports a shift-reduce conflict. It resolves in favor of shifting, which is what we want. We can

- Give ELSE higher priority than the production: precedence left ELSE.
- Tell JavaCUP on the command line to expect one conflict: −expect 1.
- Fix the grammar properly!

14 The if-then-else Solution

```
Statement ::= IF LPAREN Expression RPAREN Statement
            | NoShortIf
            ;
NoShortIf ::= IF LPAREN Expression RPAREN NoShortIf ELSE Statement
            | ... other statements ...
            ;
```


15 A reduce-reduce Conflict

These conflicts are less common and often indicate a problem of the language rather than the grammar.

```
Expression  ::= MExpression
              | DExpression
              ;
MExpression ::= IDENT TIMES IDENT
              | IDENT
              ;
DExpression ::= IDENT DIV IDENT
              | IDENT
              ;
```

```
*** Reduce/Reduce conflict found in state #4
  between MExpression ::= IDENT (*)
  and     DExpression ::= IDENT (*)
  under symbols: {EOF}
  Resolved in favor of the first production.
```


16 first(X), follow(X) and nullable

- first(X) are the terminals X can start with.
  - A terminal t is in first(X) if there is a parse tree, such that t is the leftmost leaf under X.
  - ϵ leaves do not count.
  - Example:
    A = "b" "c" | B "d".
    B = "a" | ϵ.
    first(A) = { b, a, d }
- follow(X) are terminals which can follow X.
  - A terminal t is in follow(X) if there is a parse tree such that t is the leftmost leaf after the leaves under X
  - Again, ϵ leaves do not count.
  - Example: follow(B) = { d }
- A non-terminal is nullable if it can derive the empty string (it may have only ϵ-leaves (Example: B is nullable)

17 Exercise

S = E $.
E = T "+" E | T.
T = "x".

Find the first and follow sets for T and E. Are there any nullable non-terminals?

18 How to compute first(X) and follow(X)?

... A = B "x" C. ...

- first(B) $\subseteq$ first(A).
- if B is nullable then x $\in$ first(A).
- Naive method: compute first, follow and nullable for right-hand side and from that for A.
- Does not work for recursion!

  E = E "+" T | T.
- Idea: Start with empty sets and add elements until all conditions are satisfied.
- This is called a fixpoint algorithm (It runs until there are no more changes, until the solution is fix).

19 Formal Definition: first(X), follow(X), nullable

first(X), follow(X) and nullable are the smallest sets with the following properties:

For each production X = Y1 .. Yk, $1 \leq i,\ \ j \leq k$:
  if { Y1, ..., Yk } $\subseteq$ nullable
    X $\in$ nullable
  if { Y1, ..., Yi–1 } $\subseteq$ nullable
    first(X) = first(X) $\cup$ first(Yi)
  if { Yi+1, ..., Yk } $\subseteq$ nullable
    follow(Yi) = follow(Yi) $\cup$ follow(X)
  if { Yi+1, ..., Yj–1 } $\subseteq$ nullable
    follow(Yi) = follow(Yi) $\cup$ first(Yj)

20 Algorithm for first(X), follow(X) and nullable

```
nullable = ∅;
for each terminal t { first(t) = t; follow(t) = ∅; }
for each nonterminal Y { first(Y) = ∅; follow(Y) = ∅; }
repeat {
    nullable' = nullable; first' = first; follow' = follow;
    for each production X = Y1 .. Yk, 1 ≤ i, j ≤ k {
        if { Y1, ..., Yk } ⊆ nullable
            nullable = nullable ∪ X;
        if { Y1, ..., Yi–1 } ⊆ nullable
            first(X) = first(X) ∪ first(Yi);
        if { Yi+1, ..., Yk } ⊆ nullable
            follow(Yi) = follow(Yi) ∪ follow(X);
        if { Yi+1, ..., Yj–1 } ⊆ nullable
            follow(Yi) = follow(Yi) ∪ follow(Yj);
    }
until (nullable = nullable', first = first', follow = follow');
```

21 LR(0) Parsing

- Idea: Use a DFA applied to the *stack* to decide whether to shift or to reduce.
- The states of the DFA are sets of LR(0) items.
- An LR(0) item is of the form [X = A _ B], where X is a non-terminal and A,B are strings of terminals and non-terminals (possibly empty).
- An LR(0) item describes a possible situation during parsing, where
  - X=AB. is a production, which is currently possible.
  - A is on the stack.
  - B is in the input.
  - the _ describes the border between stack and input.
- Example: [E = T _ "+" E]

22 LR(0) Parsing (2)

- Principle:
  - *shift*, in a state where [X = A _ b B] if the next token is b.
  - *reduce*, in a state [X = A _]
- The resulting parser is called LR(0), since it parses left-to-right, describes a rightmost derivation. The 0 means, that the parser uses no lookahead on the input.

23 SLR Parsing

- Problem: Some states contain shift and reduce items.
- Example:

  S = E $.
  E = T "+" E | T.
  T = "x".
- LR(0) state construction gives a state containing the items

  [E = T _ "+" E]
  [E = T _]
- If we see "+" as the next input token should we shift or reduce?
- Solution: Reduce only if the symbol is in follow(E).
- The resulting parser is called *simple LR* or *SLR*.
- The number of states is the same as in LR(0).

24 LALR(1) Parsing

- Sometimes, in specific states not all terminals from follow(X) can really occur.
- Idea: Propagate state-specific follow symbols.
- Reduce only if the symbol is in the state specific follow symbols.
- The resulting parser is called LALR(1) for Look-Ahead-LR.
- The number of states is the same as in LR(0) and SLR.
- This is, what JavaCUP uses (also yacc, bison).
- If an LALR(1) parser generator gives a conflict, then for all practical purposes it cannot know, what to do in certain situations.

25 LR(1) Parsing

- LR(1) parsing refines the notion of state. A state is now a set of LR(1) items, where each item is of the form [X = A _ B ; c] and c is a terminal.
  - X=AB. is a production, which is currently possible.
  - A is on the stack.
  - B c is in the input.
  - the _ describes the border between stack and input.
- The rest of the construction is similar to LR(0), except that we reduce in a state with item [X = A _ ; c] only if the next input token is c.
- The result is called LR(1) parsing, because now we use one token lookahead to make our decision.
- LR(1) parsers are slightly more powerful than LALR(1) parsers.
- But, there are many more LR(1) states than LR(0) states. Often we have a *state explosion*

26 Grammar in JavaCUP

```
terminal PLUS, NUMLIT;
non terminal Expression, Term;
start with Expression;

Expression  ::= Term PLUS Expression
              | Term
              ;
Term        ::= NUMLIT
              ;
```


27 States in JavaCUP

The option –dump_states yields the following output

```
START lalr_state [0]: {
  [Expression ::= (∗) Term , {EOF }]
  [Expression ::= (∗) Term PLUS Expression , {EOF }]
  [Term ::= (∗) NUMLIT , {EOF PLUS }]
  [$START ::= (∗) Expression EOF , {EOF }]
}
transition on Expression to state [3]
transition on NUMLIT to state [2]
transition on Term to state [1]
lalr_state [1]: {
  [Expression ::= Term (∗) , {EOF }]
  [Expression ::= Term (∗) PLUS Expression , {EOF }]
}
transition on PLUS to state [5]
```



```
lalr_state [2]: {
  [Term ::= NUMLIT (∗) , {EOF PLUS }]
}
lalr_state [3]: {
  [$START ::= Expression (∗) EOF , {EOF }]
}
transition on EOF to state [4]
lalr_state [4]: {
  [$START ::= Expression EOF (∗) , {EOF }]
}
```

lalr_state [5]: {
[Expression ::= (∗) Term , {EOF }]
[Expression ::= Term PLUS (∗) Expression , {EOF }]
[Expression ::= (∗) Term PLUS Expression , {EOF }]
[Term ::= (∗) NUMLIT , {EOF PLUS }]
}
transition on Expression to state [6]
transition on NUMLIT to state [2]
transition on Term to state [1]
lalr_state [6]: {
[Expression ::= Term PLUS Expression (∗) , {EOF }]
}

28 Debugging JavaCUP

Calling debug_parse() in ParserTest instead of parse() (You can use –debug), using input 5+3 yields

Initializing parser
Current Symbol is #8
Shift under term #8 to state #2
Current token is #2
Reduce with prod #3 [NT=2, SZ=1]
Goto state #1
Shift under term #2 to state #5
Current token is #8
Shift under term #8 to state #2
Current token is #0

Reduce with prod #3 [NT=2, SZ=1]
Goto state #1
Reduce with prod #2 [NT=1, SZ=1]
Goto state #6
Reduce with prod #1 [NT=1, SZ=3]
Goto state #3
Shift under term #0 to state #4
Current token is #0
Reduce with prod #0 [NT=0, SZ=2]
Goto state #–1

29 If again

Statement ::= IF IDENT Statement
| IF IDENT Statement ELSE Statement
| RETURN NUMLIT SEMI
;

reports a shift-reduce conflict. Why?

if x **if** y **return** 3; **else return** 7;

30 If Solution

Partition statements and allow in then-branch no short ifs.

```
Statement ::= ShortIf
           |  NoShortIf
           ;
ShortIf    ::= IF IDENT Statement
           ;
NoShortIf  ::= IF IDENT NoShortIf ELSE Statement
           |  RETURN NUMLIT
           ;
```


31 Error Recovery

- After an error, the parser should be able to continue processing.
- Processing is for finding other errors, not for code generation or interpretation. These get disabled after the first error.
- Question: How can the parser recover from an error and resume normal parsing?

32 Error Recovery in Bottom-Up

- There are different schemes. The following is implemented in JavaCUP, yacc, bison.
- Introduce a special symbol error.
- The author of a parser can use error in productions.
- For instance:

```
Statement =  Assignment
          |  IfStatement
          |  error ";"
          ;
```


33 Error Recovery in Bottom-Up (2)

- If the parser encounters an error, it will pop the stack until it gets into a state, where error is legal.
- At this point it shifts error onto the stack.
- Then, the input tokens are skipped, until the next input token is one that can legally follow the new state.
- This scheme is very dependent on a good choice of error productions.
- Assume a production Statement = error ";"
  - The parser encounters error inside a statement. It will pop the stack until it expects a statement.
  - At this point it shifts error onto the stack.
  - Then, the input tokens are skipped, until ";" is found.

34 Where to put error

- Different people recommend different things.
- It is a good idea to have a terminal after error to ensure termination.
- Examples:

```
Statement  ::= error SEMI
            |  LBRACE error RBRACE
            ;
Expression ::= LPAREN error RPAREN
            ;
```

- The generated parser will tell you the exact position of the error.

35 Semantic Actions

- A parser usually does more than just recognize syntax.
- It could:
  - Evaluate code (simple interpreter)
  - Emit code (single pass compiler)
  - Build an internal data structure (multi pass compiler, interpreter)
- Generally, a parser performs *semantic actions*
- In a machine-generated bottom-up parser, they are added to the grammar submitted to the parser generator.
- In a recursive descent parser, semantic actions are embedded in the recognizer routines.

36 An Interpreter for Expressions

```
terminal PLUS, MINUS, TIMES, DIV, LPAREN, RPAREN;
terminal Integer NUMLIT;

non terminal Program;
non terminal Integer Expression, Term, Factor;
precedence left PLUS, MINUS;
precedence left TIMES, DIV;

start with Program;
```


37 An Interpreter for Expressions (2)

```
Program    ::= Expression:e
               {: System.out.println(e.intValue()); :}
            ;
Expression ::= Expression:e PLUS Term:t
               {: RESULT = new Integer(e.intValue() + t.intValue()); :}
            |  Expression:e MINUS Term:t
               {: RESULT = new Integer(e.intValue() - t.intValue()); :}
            |  Term:t
               {: RESULT = t; :}
            ;
```

38 An Interpreter for Expressions (3)

```
Term    ::= Term:t TIMES Factor:f
            {: RESULT = new Integer(t.intValue() * f.intValue()); :}
          | Term:t DIV Factor:f
            {: RESULT = new Integer(t.intValue() / f.intValue()); :}
          | Factor:f
            {: RESULT = f; :}
          ;
Factor  ::= NUMLIT:n
            {: RESULT = n; :}
          | LPAREN Expression:e RPAREN
            {: RESULT = e; :}
          ;
```


39 Top-Down Parsing

- Regular languages are limited in that they cannot express nesting.
- Therefore, finite state machines cannot recognize context-free grammars.
- Let's try it anyway: A = ident A numlit | numlit. leads after simplification to the following parser:

```
void A() {
    if (token == IDENT) {
        nextToken();
        A();
        if (token == NUMLIT) nextToken(); else error();
    } else if (token == NUMLIT)
        nextToken();
    else
        error();
}
```


40 Deriving a Parser from EBNF

To derive a parser from a context-free grammar written in EBNF style:

- Introduce one function **void** A() for each non-terminal A
- The task of A() is to recognize sub-sentences derived from A, or issue an error if no A was found.
- Translate all regular expressions on the right-hand-side of productions as before, but
  - every occurrence of a non-terminal B maps to B()
  - Recursion in the grammar translates naturally to recursion in the parser.
- This technique of writing parsers is called parsing by *recursive descent* or *predictive parsing*.

41 A Parser for Expressions

```
Expression = Expression ( "-" | "+" ) Term | Term.
Term       = Term ( "*" | "/" ) Factor | Factor.
Factor     = numlit | "(" Expression ")".
```

```
void Expression() {
    if (token == NUMLIT || token == LPAREN) {
        Expression();
        if (token == MINUS || token == PLUS)
            nextToken();
        else error();
        Term();
    } else {
        Term();
    }
```

42 Eliminating Left Recursion

Expression = Term { ("−" | "+") Term }.
Term = Factor { ("*" | "/") Factor }.
Factor = numlit | "(" Expression ")".

```
void Expression() {
    Term();
    while (token == MINUS || token == PLUS) {
        nextToken();
        Term();
    }
}
```


43 Another Problem

Factor = ident | ident "[" Expression "]" | numlit.

```
void Factor() {
    if (token == IDENT) {
        ??
    } else {
        if (token == NUMLIT)
            nextToken();
        else error();
    }
}
```


44 Left Factoring

Factor = ident ["[" Expression "]"] | numlit.

```
void Factor() {
    if (token == IDENT) {
        if (token == LBRACKET) {
            nextToken();
            Expression();
            if (token == RBRACKET)
                nextToken();
            else error();
        }
    } else {
        if (token == NUMLIT)
            nextToken();
        else error();
    }
}
```


45 LL(1) Grammar

- Definition: A simple BNF grammar is LL(1) if for all nonterminals X: if X appears on the left-hand side of two productions X=E1. and X=E2. then
  - first(E1) ∩ first(E2) = ∅.
  - either (neither E1 nor E2 is nullable) or (exactly one, say E1 is nullable and first(X) ∩ follow(E2) = ∅.
- LL(1) stands for "left-to-right-parse, leftmost derivation, 1 symbol lookahead".
- Recursive descent parsers work only for LL(1) grammars.
- Elimination of left recursion and left-factoring work often, but not always.

46 Error Recovery for Top-Down

- We choose a set of stop-symbols, e.g. } ;)
- If we encounter an error, we call skip(), give an error message and return.
  - skip() skips the input to the next stop symbol.
  - It also skips subblocks { ... } completely.
- We do not print two error messages for the same position.

```
{
    a = 5 * (3 4);
}
```


47 Summary Top-Down Parsing

- A context-free grammar can be converted directly into a program scheme for a recursive descent parser.
- A recursive-descent parser builds a derivation top down, from the start symbol towards the terminal symbols.
- Weakness: Must decide what to do based on first input token.
- This works only if the language is LL(1).

48 A Hierarchy of Grammar Classes

- $LR(k+1) > LR(k)$ (Further lookahead helps)
- $LL(k+1) > LL(k)$ (Further lookahead helps)
- $LR(k) > LL(k)$ (LR is better than LL)
- $LR(1) > LALR(1) > SLR > LR(0)$

49 Top-Down / Bottom-Up

Top-Down

+ easy to write by hand.
+ flexible embedding in compiler possible.
- harder to maintain.
- error recovery can be tricky.
- deep recursion can be inefficient.

Bottom-Up

+ larger class of languages and grammar.
- needs tool to generate
- less flexible to embed in compiler
- depends on quality of tool

Mixtures are possible. Some commercial compilers use recursive descent, with operator precedence for expressions to get rid of deep recursion.