Part III : Parsing

- From Regular to Context-Free Grammars
- Deriving a Parser from a Context-Free Grammar
- Scanners and Parsers
- A Parser for EBNF
- Left-Parsable Grammars

From Regular to Context-Free Grammars

- Regular languages are limited in that they cannot express nesting.
- Therefore, finite-state machines cannot recognise context-free grammars.
- But let's try it anyway!
- **Example**: The grammar:

```
        A = "a" A "c" | "b".
```

  leads after simplification to the following ``recogniser'':

From Regular to Context-Free Grammars (2)

```
if (sym == "a") {
  next();
  if (sym == A) next(); else error();
  if (sym == "c") next(); else error();
} else if (sym == "b") {
  next();
} else {
  error();
}
```

- This is bogus, of course, since we have treated the non-terminal symbol *A* as a terminal.
- But it leads to a natural extension of our recognising algorithm.

Deriving a Parser from a Context-free Grammar

- To derive a parser from a context-free grammar written in EBNF style:
- Introduce one function `void A()` for each nonterminal `A`.
- The task of `A()` is to recognise sub-sentences derived from `A`, or to issue an error if no `A` was found.
- Translate all regular expressions on the right-hand side of productions as before, with the addition that
  - every occurrence of a nonterminal `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 ».

Deriving a Parser from a Context-free Grammar (2)

Example: The grammar:

```
A = "a" A "c" | "b".
```

now leads to the function:

```
void A() {
  if (sym == "a") {
    next();
    A();
    if (sym == "c") next(); else error();
  } else if (sym == "b") {
    next();
  } else {
    else error();
  }
}
```

Scanners and Parsers

- Most compilers in practice have both a scanner for the lexical syntax and a parser for the context-free syntax
- Advantages: Separation of concerns, better modularity.

Component	Input	Output
Scanner	Characters	Symbols
Parser	Symbols	Syntax-Tree

A Parser for EBNF

- We write the parser as an extension of the Scanner.

```
package ebnf;
import java.io.*;
class Parser extends Scanner {
    public Parser(InputStream in) {
        super(in);
        nextSym();
    }
```

- Now, simply translate each production according to the given scheme.
- Here's the production for parsing a whole grammar:

```
/* syntax = {production} <eof> */
public void syntax() {
   while (sym != EOF) {
     production();
   }
}
```

A Parser for EBNF (2)

And here are the productions for the other nonterminals.

```
/* production = identifier "=" expression "." */
void production() {
    if (sym == IDENT) nextSym();
    else error("illegal start of production");
    if (sym == EQL) nextSym();
    else error("`=' expected");
    expression();
    if (sym == PERIOD) nextSym();
    else error("`.' expected");
}
/* expression = term {"|" term} */
void expression() {
    term();
    while (sym == BAR) {
        nextSym();
        term();
    }
}
```

A Parser for EBNF (3)

```
/* term = {factor} */
void term() {
while (sym == IDENT || sym == LITERAL ||
       sym == LPAREN || sym == LBRACK || sym == LBRACE)
    factor();
}
/* factor = identifier  |  string  |
*            "(" expression ")" | "[" expression "]"  |
             "{" expression "}" */
void factor() {
    switch (sym) {
    case IDENT:
        nextSym();
        break;
    case LITERAL:
        nextSym();
        break;
```

A Parser for EBNF (4)

```
        case LPAREN:
            nextSym();
            expression();
            if (sym == RPAREN) nextSym();
            else error("`)' expected");
            break;
        case LBRACK:
            nextSym();
            expression();
            if (sym == RBRACK) nextSym();
            else error("`]' expected");
            break;
        case LBRACE:
            nextSym();
            expression();
            if (sym == RBRACE) nextSym();
            else error("`}' expected");
            break;
        default:
            error("illegal start of factor");
        }
    }
```

Left-Parsable Grammars

As before, the grammar must be *left-parsable* for this scheme to work.

Two conditions:

- In an alternative T_1 | ... | T_n, the terms do not have any common start symbols.
- If some part **exp** of a regular expression contains the empty string then **exp** cannot be followed by any symbol that is also a start symbol of **exp**.
- We now formalize what this means by defining for every symbol X the sets first (X), follow (X) and nullable (X).

From EBNF to simple BNF

It's easy to convert an EBNF grammar to BNF:

- Convert every repetition {E} in a EBNF grammar to a fresh nonterminal symbol X_{rep} and add the production:

  $$X_{rep} = E\ X_{rep} \mid \in$$

- Convert every option [E] in the grammar to a fresh nonterminal symbol X_{opt} and add the production:

  $$X_{opt} = E \mid \in$$

- We can even do away with the alternatives by having several productions for the same nonterminal symbol. Example:

  $$X_{rep} = E\ X_{rep} \mid \in$$

becomes

$$X_{rep} = E\ X_{rep}$$
$$X_{rep} = \in$$

Definition of first, follow and nullable

Given a context free language, define

nullable	the set of non-terminal symbols that can derive the empty string.
first(X)	the set of terminal symbols that can begin strings derived from X.
follow(X)	the set of terminal symbols that can immediately follow X. That is, t ∈ follow(X) if there is a derivation that contains X t.

Formal definition of first, follow and nullable

first, follow and nullable are the smallest sets for which these properties hold.

```
for each production $X = Y_1 \dots Y_k$, $1 \le i < j \le k$:
    if $\{ Y_1, \dots, Y_k \} \subseteq$ nullable
      $X \in$ nullable
    if $\{ Y_1, \dots Y_{i-1} \} \subseteq$ nullable
      first( X ) = first( X ) $\cup$ first( $Y_i$ )
    if $\{ Y_{i+1}, \dots, Y_k \} \subseteq$ nullable
      follow( $Y_i$ ) = follow( $Y_i$ ) $\cup$ follow( X )
    if $\{ Y_{i+1}, \dots, Y_{j-1} \} \subseteq$ nullable
      follow( $Y_i$ ) = follow( $Y_i$ ) $\cup$ first( $Y_j$ )
```

Algorithm for computing first, follow and nullable

Simply replace equations by assignments and iterate until no change.

```
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 $X = Y_1 \dots Y_k$, $1 \leq i < j \leq k$:
    if { $Y_1, \dots, Y_k$ }  nullable
      nullable := nullable $\cup$ { X }
    if { $Y_1, \dots Y_{i-1}$ } $\subseteq$ nullable
      first( X ) := first( X ) $\cup$ first( $Y_i$ )
    if { $Y_{j+1}, \dots, Y_k$ } $\subseteq$ nullable
      follow( $Y_i$ ) := follow( $Y_i$ ) $\cup$ follow( X )
    if { $Y_{i+1}, \dots, Y_{j-1}$ } $\subseteq$ nullable
      follow( $Y_i$ ) := Follow( $Y_i$ ) $\cup$ first( $Y_j$ )
until nullable' = nullable & first' = first & follow' = follow.
```

LL(1) Grammars

- 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

  X = E2

  then

  first(E1) ∩ first(E2) = {}

  and either

  neither E1 nor E2 is nullable

  or

  exactly one E_i is nullable and
  first(X) ∩ follow(X) = {}
- LL(1) stands for "left-to-right parse, leftmost derivation, 1 symbol lookahead".
- Recursive descent parsers work only for LL(1) grammars.

Converting to LL(1)

- Example: Grammar for arithmetic expressions over arrays.

```
E = E "+" T | E "-" T | T
T = T "*" F | T "/" F | F
F = ident | ident "[" E "]" | "(" E ")"
```

- Is this grammar LL(1)?

Techniques :

- Eliminate left recursion. E.g.

  E = E "+" T | E "-" T | T

  becomes

  E = T { "+" T | "-" T}

- Left-factoring: E.g.

  F = ident | ident "[" E "]" | ...

  becomes

  F = ident (∈ | "[" E "]") | ...

  or, using an option:

  F = ident ["[" E "]"] | ...

Limitations

- Elimination of left recursion and left factoring work often, but not always.
- - Example:

```
S  =  { A }.
A  =  id ":=" E.
E  =  {id}.
```

- This language cannot be given an LL(1) grammar. But it is LL(2), i.e. can be parsed with 2 symbols look-ahead.
- Generally LL(k) is a true subset of LL(k+1).
- But LL(1) is the only interesting case.

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 symbol.
- This works only if the grammar is LL(1).

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*.
- Two basic actions:

  shift : push next symbol from input on stack

  reduce: remove symbols $Y_n,\ldots,Y_1$ which form right-hand side of some production

  $$X = Y_1 \ldots Y_n$$

  from top of stack and replace by X.
- Other actions: accept, error.
- Question: How does the parser know when to shift and when to reduce?

Simple Answer : Operator Precedence

- Suitable for languages of the form

  Expression = Operand Operator Operand.

  with operands of varying precedence and associativity.

- Principle:

```
let IN be next input symbol.
if IN is an operand then
    shift
else if stack does not contain an operator then
    shift
else
    let TOP be the topmost operator on stack.
    if precedence(TOP) < precedence(IN) then shift
    else if precedence(TOP) > precedence(IN) then reduce
    else if IN and TOP are both right associative then shift
    else if IN and TOP are both left associative then reduce
    else error
```

More General Answer : 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 nonterminal symbol and A,B are possibly empty strings of symbols.
- An LR(0) item describes a possible situation during parsing, where
  - X = AB is a possible production for the current derivation
  - A is on the stack
  - B remains in the input.
  - Hence, the "." describes the border between stack and input.

More General Answer : LR(0) Parsing (2)

- Principle:
  - *Shift* in a state which contains the item [X = *A* . b B] if the next input symbol is b.
  - *Reduce* in a state which contains the item [X = *A* .]
- Example: See Appel, Figure 3.20
- The resulting parser is called LR(0) since it parses input left-to-right and describes a right-most derivation in reverse. The 0 means that the parser uses no lookahead on the input.

SLR Parsing

- Problem: Some states contain both shift and reduce items.
- Example: Consider the grammar:

```
S = E $
E = T + E
E = T
T = (E)
T = x
```

- LR(0) state construction gives a state containing the items

```
E = T. + E
E = T.
```

- If we see "+" as next input symbol, should we shift or reduce?
- Solution: Reduce only if input symbol is in follow(E).
- The resulting parser is called "simple LR", or *SLR*.

LR(1) Parsing

- Even more powerful than SLR is 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]

- This models the following situation

  X = AB is a production of the grammar
  A is on the stack
  Bc is part of the 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 symbol is c.
- The result is called LR(1) parsing, since it reads input left-to-right, describes a right-most derivation in reverse, and uses 1 look-ahead symbol.

LALR(1) Parsing

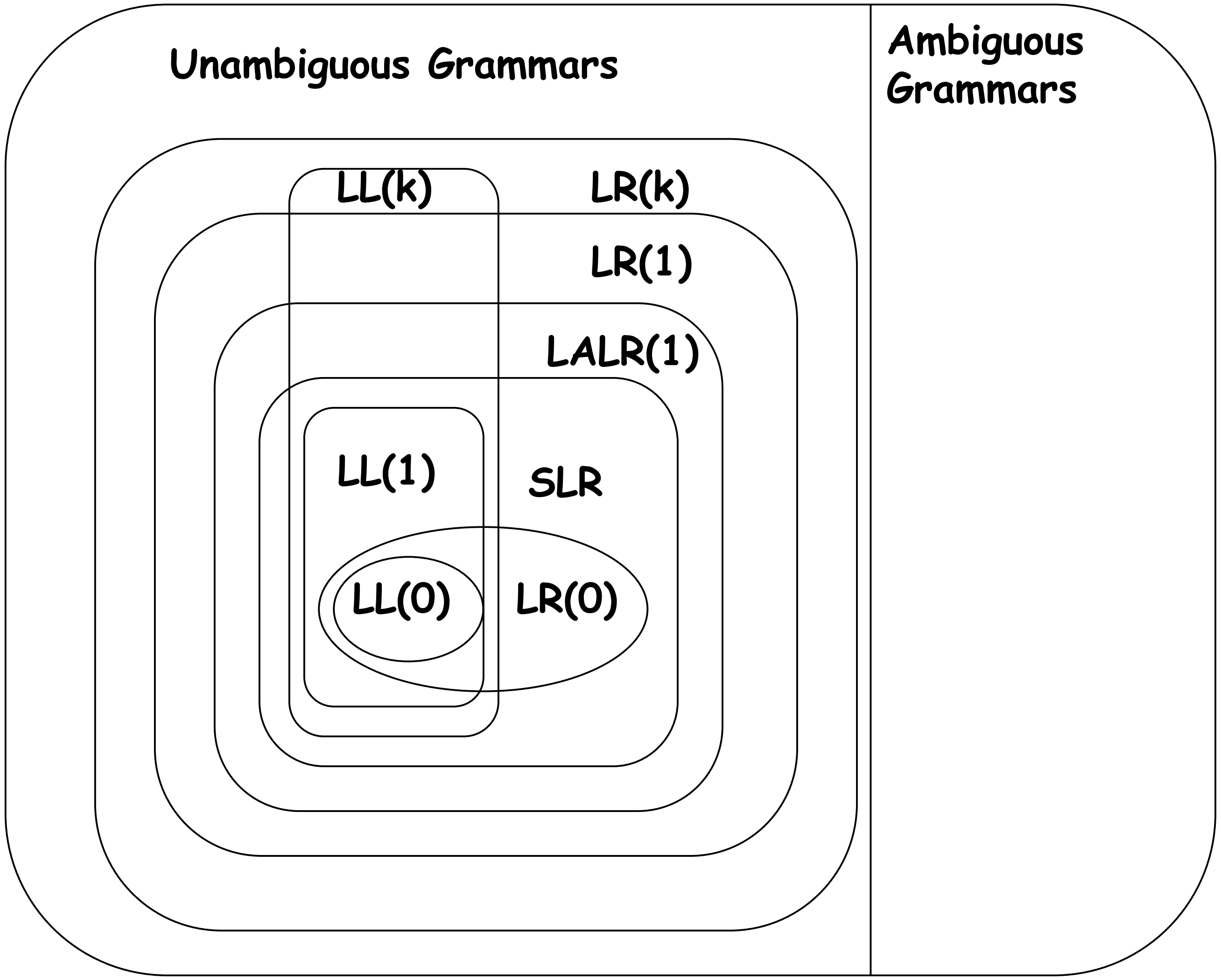
- LR(1) Parsers are more powerful than SLR parsers.
- But: There are many more LR(1) states than LR(0) states. Often, there is a problem with *state explosion*.
- Solution: Merge states that differ only in their lookahead symbol.
- Example: The two states,

  {[X = A.B ; c]}, {[X = A.B ; d]}

  become:

  {[X = A.B ; c], [X = A.B ; d]}
- The resulting parser is called LALR(1) for Look-Ahead-LR(1).
- The LALR(1) technique is the basis of most parser generators, e.g. Yacc, Bison, JavaCUP.

A Hierarchy of Grammar Classes



Example Parser Specification

```
Terminal ID, WHILE, BEGIN, END, DO, IF, THEN, ELSE,
SEMI, ASSIGN;

non terminal   prog, stm, stmlist;

start with     prog;

prog ::=   stmlist;

stm ::= ID ASSIGN ID
    |   WHILE ID DO stm
    |   BEGIN stmlist END
    |   IF ID THEN stm
    |   IF ID THEN stm ELSE stm;

stmlist ::= stm
        |   stmlist SEMI stm;
```

Pragmatics

- Is the grammar of J0 LL(1) or LR(1)?
- It's not even un-ambiguous!
- Problem:

  Block = {Statement}

  Statement = "if" "(" Expression ")" Statement ["else" Statement]

- How do we parse

```
if (x != 0) if (x < 0) then y = -1 else y = 1
```

- ?

Pragmatics (2)

- Exercise: Rewrite the grammar so that it becomes unambiguous.
- Pragmatic solutions:
  - Recursive descent: Apply longest match rule
  - LR: Have priorities of rules. E.g., with earlier rules taking precedence over later ones:

```
Statement = "if" "(" Expression ")" Statement "else"
            Statement
Statement = "if" "(" Expression ")" Statement
```

Relative Advantages of Top-Down and 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 grammars
	-	Needs tool to generate
	-	Less flexible to embed in compiler
	-	Depends on quality of tool

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

Error Diagnosis

- When encountering an illegal input program, the parser needs to give an error message.
- What error message should be given for:

```
x [i) = 1;
```

- and for:

```
x = if (a < b) 1 else 2;
```

- It's often helpful to include the actual input that was encountered. E.g.

```
"{" expected but identifier found
```

- We can use the `representation` function in the Scanner for this task.

Error Recovery

- After an error, the parser should be able to continue processing.
- Processing is for finding other errors, not for generating code.

  $\Rightarrow$ Code generation will be disabled.
- Question: How can the parser recover from an error and resume normal parsing?
- - Two elements of the solution:
  - Skip part of the input
  - Reset the parser into a state where it can process the rest of the input.

Error Recovery for Recursive Descent

- Let $X_1 \dots X_n$ be the current stack of executing parsing methods.
- Idea: Skip the input to a symbol in FOLLOW(X_i) for some i and unwind the stack until the return point of X_i.
- In practice, it's often good enough to have a fixed set of *stop symbols* which terminate a skip. E.g., for JO:

  ";", "}", ")", EOF

- In practice, it's also important to skip sub-blocks completely.

Example:

```
if x < 0 { ... }
   ^ '(' expected but identifier found
```

Should not skip to "}"!

- This can be achieved by counting opening parentheses and braces.

A Skip Procedure

```
void skip () {
          int nparens = 0;
          while (true) {
            switch (token) {
            case EOF    : return;
            case SEMI   : if (nparens == 0) return;
                          break;
            case LPAREN:
            case LBRACE: nparens++; break;
            case RPAREN:
            case RBRACE: if (nparens == 0) return;
                          nparens--; break;
           }
           nextToken();
         }
       }
```

Rewinding the Stack

- Problem: How do we rewind the stack in a recursive descent parser?
- Surprisingly simple solution: Simply continue parsing, as if nothing had happened!
- Eventually, the parser will reach a state where it can accept the stop symbol that's first on the input after a skip. That is, it will *re-synchronize*.
- Necessary for termination: The parser should not invoke a method X() unless the next input symbol is in first(X).
- But this will generate lots of spurious error messages until the parser re-snychronizes!
- Solution: After a skip, don't print any error messages until the parser has consumed at least one other input symbol.

Handling Syntax Errors

- Introduce a global variable **`pos`** for the position of next input token (either in lines, columns, or in number of characters from start).
- **`pos`** should be maintained by scanner.
- Introduce a global variable **`skipPos`** for the position we last skipped to.

```
int skipPos = -1
```

- Now define a procedure for handling syntax errors as follows.

```
/** Generate a syntax error unless one was
    already reported at the current skip position,
    then skip. */
private void syntaxError(String msg) {
    if (pos != skipPos) errorMsg(pos, msg);
    skip();
    skipPos = pos;
}
```

- This is very simple and works well in practice.

Bottom-Up Error Recovery

- Various schemes are possible. Here's the one implemented in Yacc, Bison, JavaCUP:
- Introduce a special symbol **`error`**.
- The author of a parser can use **`error`** in productions.
- For instance:

```
Block = "{" {Statement}"}"
      | "{" {Statement} error "}"
```

Bottom-Up Error Recovery (2)

- If the parser encounters an error, it will pop the stack until it gets into a state where `error` is a legal next symbol.

```
Block = "{" {Statement} . error "}"
```

- At this point, `error` will be shifted:

```
Block = "{" {Statement} error . "}"
```

- Then, the input symbols are skipped until the next input symbol is one that can legally follow in the new state.
- This scheme is very dependent on a good choice of error productions.