Parsing algorithms

Given grammar, want to parse input programs
- check legality
- produce AST representing structure
- be efficient

Kinds of parsing algorithms:
- top-down
- bottom-up

Top-down parsing

Build parse tree for input program from the top (start symbol) down to leaves (terminals)
- leftmost derivation

Basic issue:
- when replacing a nonterminal with some r.h.s., how to pick which r.h.s.?

E.g.

```
Stmt    ::= Call | Assign | If | While
Call    ::= Id
Assign  ::= Id := Expr
If      ::= if Test then Stmts end |
            if Test then Stmts else Stmts end
While   ::= while Test do Stmts end
```

Solution: look at input tokens to help decide

Predictive parsing

Predictive parser:
top-down parser that can select correct rhs looking at at most *k* input tokens (the **lookahead**)

Efficient:
- no backtracking needed
- linear time to parse

Implementation of predictive parsers:
- table-driven parser
  - PDA: like table-driven FSA, plus stack to do recursive FSA calls
- recursive descent parser
  - each nonterminal parsed by a procedure
  - call other procedures to parse sub-nonterminals, recursively

LL(*k*) grammars

Can construct predictive parser automatically/easily if grammar is **LL(*k*)**
- **L**eft-to-right scan of input, **L**eftmost derivation
- ***k*** tokens of lookahead needed, ≥ 1

Some restrictions:
- no ambiguity
- no **common prefixes** of length ≥ *k*:
  ```
  S ::= if Test then Ss end |
        if Test then Ss else Ss end | ...
  ```
- no **left recursion**:
  ```
  E ::= E Op E | ...
  ```
- a few others

Restrictions guarantee that, given *k* input tokens, can always select correct rhs to expand nonterminal

Eliminating common prefixes

Can **left factor** common prefixes to eliminate them

- create new nonterminal for common prefix and/or different suffixes

Before:

```
If       ::= if Test then Stmts end |
             if Test then Stmts else Stmts end
```

After:

```
If       ::= if Test then Stmts IfCont
IfCont   ::= end | else Stmts end
```

Grammar a bit uglier

Easy to do by hand in recursive-descent parser

Eliminating left recursion

Can rewrite grammar to eliminate left recursion

Before:

```
E ::= E + T | T
T ::= T * F | F
F ::= id | ...
```

After:

```
E ::= T ECont
ECont ::= + T ECont | ε
T ::= F TCont
TCont ::= * F TCont | ε
F ::= id | ...
```

After, in sugared form:

```
E ::= T { + T }
T ::= F { * F }
F ::= id | ...
```

Sugared form pretty readable still

Concrete syntax tree not as close to abstract syntax tree

Easy to implement in hand-written recursive descent

Table-driven predictive parser

Can automatically convert grammar into PREDICT table

PREDICT(*nonterminal*, *input-sym*) ⇒ *production*

- selects the right production to take given a nonterminal to expand and the next token of the input

E.g.

```
stmt  ::= ① if expr then stmt else stmt |
          ② while expr do stmt |
          ③ begin stmts end
stmts ::= ④ stmt ; stmts | ⑤ ε
expr  ::= ⑥ id
```

PREDICT

	if	then	else	while	do	begin	end	id	;
stmt	①			②		③			
stmts	④			④		④	⑤		
expr								⑥	

Constructing PREDICT table

Compute FIRST for each r.h.s.

- FIRST(`RHS`) =
  {all tokens that can appear first in a derivation from `RHS`}

In case `RHS` can be empty:

Compute FOLLOW for each nonterminal

- FOLLOW(`X`) =
  {all tokens that can appear right after `X` in a derivation}

FIRST and FOLLOW computed mutually recursively

	FIRST	FOLLOW
S ::= **if** E **then** S **else** S		
| **while** E **do** S		
| **begin** Ss **end**		
Ss ::= S ; Ss		
| ε		
E ::= **id**		

FIRST ⇒ PREDICT

Another example

Sugared:

```
E ::= T { + T }
T ::= F { * F }
F ::= - F | id | ( E )
```

Unsugared:

```
E  ::= ① T E′
E′ ::= ② + T E′ | ③ ε
T  ::= ④ F T′
T′ ::= ⑤ * F T′ | ⑥ ε
F  ::= ⑦ - F | ⑧ id | ⑨ ( E )
```

	FIRST	FOLLOW
E ::= T E′		
E′ ::= + T E′		
| ε		
T ::= F T′		
T′ ::= * F T′		
| ε		
F ::= - F		
| id		
| (E)		

PREDICT

	id	+	-	*	/	(	)
E							
E′							
T							
T′							
F							

PREDICT and LL(1)

If PREDICT table has at most one entry in each cell, then grammar is LL(1)

- always exactly one right choice
  ⇒ fast to parse and easy to implement
- LL(1) ⇒ each column labelled by 1 token

Can have multiple entries in each cell

- e.g. if common prefixes or left recursion or ambiguity
- can patch table by hand, if know “right” answer
- or use more powerful parsing techniques

Recursive descent parsers

Write subroutine for each non-terminal

- each subroutine first selects correct r.h.s. by peeking at input tokens
- then consume r.h.s.
  - if terminal symbol, verify that it’s next & then advance
  - if nonterminal, call corresponding subroutine
- construct & return AST representing r.h.s.

PL/0 parser is recursive descent

PL/0 scanner routines:

- `Token* Get();`
- `Token* Peek();`
- `Token* Read(SYMBOL expected_kind);`
- `bool CondRead(SYMBOL expected_kind);`

Example

```
Stmt    ::= Assign | If
Assign  ::= id := Expr ;
If      ::= if Expr then Stmt [else Stmt] end ;
```

```
Stmt* Parser::ParseStmt() {
   Token* t = scanner->Peek();
   switch (t->kind()) {
      case IDENT:
         return ParseAssignStmt();
      case IF:
         return ParseIfStmt();
      default:
         Plzero->syntaxError("expecting a stmt");
   }
}
```


Example, continued

```
If      ::= if Expr then Stmt [else Stmt] end ;
```

```
IfStmt* Parser::ParseIfStmt() {
   scanner->Read(IF);
   Expr* test = ParseExpr();
   scanner->Read(THEN);
   Stmt* then_stmt = ParseStmt();
   Stmt* else_stmt;
   if (scanner->CondRead(ELSE)) {
      else_stmt = ParseStmt();
   } else {
      else_stmt = NULL;
   }
   scanner->Read(END);
   scanner->Read(SEMICOLON);
   return new IfStmt(test, then_stmt, else_stmt);
}
```


Example, continued

```
Sum     ::= Term { (+ | -) Term }
```

```
Expr* Parser::ParseSum() {
   Expr* expr = ParseTerm();
   for (;;) {
      Token* t = scanner->Peek();
      if (t->kind() == PLUS || t->kind() == MINUS) {
         scanner->Get();  // skip over operator
         Expr* arg = ParseTerm();
         expr = new BinOp(t->kind(), expr, arg);
      } else {
         break;
      }
   }
   return expr;
}
```


Yacc

`yacc`: “**y**et **a**nother **c**ompiler-**c**ompiler”

Input: grammar, possibly augmented with action code
Output: C functions to parse grammar and perform actions

LALR(1) parser generator
- practical bottom-up parser
- more powerful than LL(1)

`yacc++`, `bison`, `byacc` are modern updates of `yacc`

Yacc input grammar

Examples:

```
assignstmt: IDENT GETS expr
          ;
ifstmt: IF test THEN stmts END
      | IF test THEN stmts ELSE stmts END
      ;

expr: term
    | expr '+' term
    | expr '-' term
    ;

factor: '-' factor
      | IDENT
      | INTEGER
      | INPUT
      | '(' expr ')'
      ;
```


Yacc with actions

Examples:

```
assignstmt: IDENT GETS expr
              { $$ = new AssignStmt($1, $3); }
          ;
ifstmt: IF test THEN stmtlist END
              { $$ = new IfStmt($2, $4); }
      | IF test THEN stmts ELSE stmts END
              { $$ = new IfElseStmt($2, $4, $6); }
      ;

expr: term { $$ = $1; }
    | expr '+' term
              { $$ = new BinOp(PLUS, $1, $3); }
    | expr '-' term
              { $$ = new BinOp(MINUS, $1, $3); }
    ;

factor: '-' factor { $$ = new UnOp(MINUS, $2); }
      | IDENT { $$ = new VarRef($1); }
      | INTEGER { $$ = new IntLiteral($1); }
      | INPUT { $$ = new InputExpr; }
      | '(' expr ')' { $$ = $2; }
      ;
```


Error handling

How to handle syntax error?

Option 1: quit compilation
- \+ easy
- \- inconvenient for programmer

Option 2: error recovery
- \+ try to catch as many errors as possible on one compile
- \- avoid streams of spurious errors

Option 3: error correction
- \+ fix syntax errors as part of compilation
- \- hard!!

Panic mode error recovery

When find a syntax error, skip tokens until reach a “landmark”
- landmarks in PL/0: **`end`**, **`;`**, **`)`**, **`then`**, **`do`**, ...
- once a landmark is found, will have gotten back on track

In top-down parser, maintain set of landmark tokens as recursive descent proceeds
- landmarks selected from terminals later in production
- as parsing proceeds, set of landmarks will change, depending on the parsing context

Example of error recovery

Grammar augmented with landmark sets

```
program  ::= stmt{EOF}
stmt     ::= if expr{then} then stmt |
             while expr{do} do stmt |
             begin stmts{end} end
stmts    ::= stmt{;} ; stmts | ε
expr     ::= id | ( expr{)} )
```

Sample input

```
if x then begin while ( begin ...
```