Parsing

A parser is a program that, given an input sentence, “recognizes” whether or not that sentence is in the language of a given grammar. Works by reconstructing a derivation for the sentence. Parser “constructs” parse tree:

- explicitly – actual data structure is built; or
- implicitly – “semantic actions” are invoked at points corresponding to nodes in the tree, but no tree is actually built.

All parsers read input left-to-right, but they differ in how tree is constructed: top-down vs. bottom-up.

Any CFG can be parsed by a (nondeterministic) pushdown automaton (NFA + stack), but not necessarily by a deterministic one (much less an efficient one).

Top-down Parsing

Idea: construct parse tree by starting at start symbol and “guessing” each derivation until we reach a string that matches input.

Example Grammar:

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S | \text{while } E \text{ do } S | \text{print } E
\]

Token string:

if id then while true do print else print

Tree:

```
S
 ▲
  / \nif E then S else S
```

Input:

if id then while true do print else print

Action: Guess for \( S \)

Tree:

```
S
 ▲
  / \nif E then S else S
```

Input:

if id then while true do print else print

Action: Guess for \( E \)
Tree:

```
S
  if E then S else S
    id
```

Input:
id then while true do print else print

Action:
id matches; then matches; guess for S

Tree:

```
S
  if E then S else S
    id while E do S
    true
```

Input: true do print else print

Action:
true matches; do matches; guess for S

Tree:

```
S
  if E then S else S
    id while E do S
    print
```

Input: while true do print else print

Action:
while matches; guess for E

Tree:

```
S
  if E then S else S
    id while E do S
    print
```

Input: print else print

Action:
print matches; else matches; guess for S
Recursive-Descent Parsing

Implementation of top-down parser using a recursive procedure for each non-terminal.

For many languages, can make perfect guesses (avoid backtracking) by using 1-symbol lookahead. I.e., if

\[ A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n \]

choose correct \( \alpha_i \) by looking at first symbol it derives.

(If \( \epsilon \) is an alternative, choose it last.)

This approach is also called predictive parsing

R.D. parsers are easy to write by hand and reasonably efficient.

Often must massage grammar into suitable form (more later).

Not all languages can be parsed this way.

Recursive-Descent Example

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S | \text{while } E \text{ do } S | \epsilon
\]

\[
E \rightarrow \text{true} | \text{false} | \text{id}
\]

\[
s() \{
\text{if (tok == IF) }
\text{tok = lex(); /* get next input token */}
\text{e();}
\text{if (tok == THEN) }
\text{tok = lex();}
\text{s(); /* recursive call! */}
\text{if (tok == ELSE) }
\text{tok = lex();}
\text{s();}
\text{else error(); /* issue error message */}
\text{else error();}
\text{else if (tok == WHILE) }
\text{tok = lex();}
\text{e();}
\text{if (tok == DO) }
\text{tok = lex();}
\text{s();}
\text{else error();}
\text{else error();}
\text{*/ epsilon case falls out */}
\}
\]

\[
e() \{
\text{if (tok == TRUE || tok == FALSE || tok == ID)}
\text{tok = lex();}
\text{else error();}
\}
\]

Problems for Recursive-Descent Parsing

- Left recursion: a derivation
  \[ A \Rightarrow A\alpha \]
  causes parser to loop!

  Solution: Remove left recursion from grammar.

- Need to backtrack (inefficient) because one-symbol lookahead can’t “guess” correctly, e.g.:

  \[ S \rightarrow V := \text{int} \]
  \[ V \rightarrow \text{alpha } [' \text{int }'] | \text{alpha} \]

  Possible inputs: \( x := 77 \) or \( x[2] := 17 \).

  Which alternative should we choose for \( V \)?

  Solution: Left-factor the grammar.

- These problems arise naturally in expression grammars. (Can usually prevent them in statement grammars by careful language design.)
Eliminating Immediate Left Recursion

Replace left-recursive productions of the form

\[ A \rightarrow A\alpha | \beta \]

which generate sentences of the form

\[ \beta, \beta\alpha, \beta\alpha\alpha, \ldots \]

by the right-recursive productions

\[ A \rightarrow \beta A' \]
\[ A' \rightarrow \alpha A' | \epsilon \]

Yields different parse trees but same language:

Example: Arithmetic Expressions

\[ T \rightarrow T * F | T / F | F \]

becomes

\[ T \rightarrow F T' \]
\[ T' \rightarrow * F T' | / F T' | \epsilon \]

Expression Parsing Using Recursive Descent(I)

Consider a simple grammar of arithmetic expressions, with precedence expressed by multiple levels of non-terminals:

\[ E \rightarrow E + T | T \]
\[ T \rightarrow T * F | F \]
\[ F \rightarrow (E) | id \]

To prepare it for recursive-descent parsing, we first remove left-recursion:

\[ E \rightarrow TE' \]
\[ E' \rightarrow + TE' | \epsilon \]
\[ T \rightarrow FT' \]
\[ T' \rightarrow * FT' | \epsilon \]
\[ F \rightarrow (E) | id \]

This leads directly to this code for \( E \) and \( E' \) (\( T \) and \( T' \) are similar):

```c
E() {
    T();
    E1();
}

E1() {
    if (tok == '+') {
        tok = lex();
        t();
        E1();
    }
    else if (tok == '*') {
        tok = lex();
        t();
        E1();
    }
    else if (tok == '/') {
        tok = lex();
        t();
        E1();
    }
    else if (tok == '(') {
        tok = lex();
        t();
        E();
        tok = lex();
        t();
        E1();
    }
    else if (tok == ')') {
        tok = lex();
        t();
        E1();
    }
    else if (tok == 'id') {
        tok = lex();
        t();
        E1();
    }
    else {
        // error case
    }
}
```

Fairly General Algorithm

(Works unless \( A \Rightarrow A \) or \( A \Rightarrow \epsilon \). Fully general algorithm exists, but is complicated!)

- arrange non-terminals in some order \( A_1, \ldots, A_n \).
- for \( i = 1 \) to \( n \) do
  - for \( j = 1 \) to \( i-1 \) do
    - for any production \( A_i \rightarrow A_j \alpha \)
      - replace it by substituting defn.
        of \( A_j \) into r.h.s., i.e., by
          \[ A_i \rightarrow \beta_1 \alpha | \ldots | \beta_m \alpha \]
      - where current productions for \( A_j \) are
        \[ A_j \rightarrow \beta_1 | \ldots | \beta_m \]
    - eliminate any immediate left-recursion in \( A_i \)

Example: Replace \( A \rightarrow Sc \) \( d \)

by \( A \rightarrow Aac | bc \) \( d \)

and then by \( A \rightarrow bcA' | dA' \)
\[ A' \rightarrow acA' | \epsilon \]

Left-factoring

Easy to remove common prefixes by left-factoring, creating new non-terminals.

Change

\[ V \rightarrow \alpha\beta | \alpha\gamma \]

to

\[ V \rightarrow \alpha V' \]
\[ V' \rightarrow \beta | \gamma \]

Example:

\[ S \rightarrow V := \text{int} \]
\[ V \rightarrow \text{alpha } [' int '] | \text{alpha} \]

becomes

\[ S \rightarrow V := \text{int} \]
\[ V \rightarrow \text{alpha } V' \]
\[ V' \rightarrow [' int '] | \epsilon \]
Expression parsing using recursive-descent (II)

We can simplify this code (and improve its performance) by turning the recursions into iterations, e.g.:

```c
e1() {
  if (tok == '+') {
    tok = lex(); t(); e1();
  }
  /* epsilon case falls through */
}
```

becomes:

```c
e1() {
  while (tok == '+') {
    tok = lex(); t();
  }
}
```

and then inlining functions now called only once, e.g.:

```c
e() { t(); e1(); }
```

becomes:

```c
e() {
  t();
  while (tok == '+') {
    tok = lex(); t();
  }
}
```

Expression parsing using recursive-descent (III)

Performing the same transformation on \( t \) and \( t_1 \), and adding the usual recursive descent code for \( f \), we get:

```c
e() {
  t();
  while (tok == '+') { lex(); t(); }
}
```

```c
t() {
  f();
  while (tok == '*') { lex(); f(); }
}
```

```c
f() {
  if (tok == ID) lex();
  else if (tok == '(') {
    lex();
    e();
    if (tok == ')') lex();
    else error();
  }
  else error();
}
```