Recursive Descent Parsing

Recursive Descent Parsing (RDP)

Reminder:

- Parser checks that its input (sequence of tokens) is syntactically correct
  * not just a sequence of legal words
  * sentence with correct structure: can be derived in the grammar of the language (derived from the initial variable S)

RDP:

- Tries to rebuild the syntax tree of the input
- The tree is rebuilt in a top-down way (from S downwards to tokens)
- Structure of parser reflects the structure of the language grammar
- If the grammar is recursive then the parser is recursive as well

How parser acts

Parser scans the stream of tokens from left to right:

t1 t2 t3 ………… tn EOF

→

At every stage:

- based on the **history** (already seen in the input) :

  updates its **prediction** (תחזית) for the rest –

  what should come next for the input to be correct?

- checks: does the **reality** (tokens that it sees in the input) **fits the current prediction?**

Two types of predictions - 1

The next token is of kind t :

t1 t2 t3 …… tk t ……. tn EOF

^

Checked by calling **`match(t)`** :

```
void match(token_kind t)
    {cur_token = next_token();
      if (cur_token -> kind != t)
           error();
    }
```

This function does not depend on the language's grammar

Two types of predictions - 2

A fragment starting at the next token is derived from variable X :

```
t1  t2  t3  …… tk  tk+1  …  tm….  tn  EOF
                 ^   |________|
                         X
```

NOTE: input contains tokens only, so variable X itself doesn't appear in the input !

Checked by calling **`parse_X()`**

This function does depend on the grammar:

reflects the derivation rules for variable X (X עבור "תפור")

Initial prediction

Two requirements:

- the entire input is correctly derived from S
- after the derivation is finished, EOF is reached

Hence, parser's main function is as follows:

```
void parser()
     {parse_S();
      match(EOF)
     }
```

Structure of parse_X()

1) Single derivation rule for variable

DECLARATION → TYPE VAR_LIST

t1 t2 t3 …… tk tk+1 … tm…. tn EOF

|_______|

DECLARATION   call parse_DECLARATION()

t1 t2 t3 …… tk tk+1 …………. tm…. tn EOF

|TYPE VAR_LIST|

DECLARATION

Function:

```
void parse_DECLARATION() { parse_TYPE(); parse_VAR_LIST() }
```

Structure of parse_X()

2) Several rules for same variable

S → id = E | <u>while id < E do S od</u>

E → id | num

t1 t2 t3 …… tk tk+1 … tm …. tn EOF

|_______|

S   call parse_S()

can be

t1 t2 t3 …… tk tk+1 ……. tm …. tn EOF

|__id = E__|

S

or

t1 t2 t3 …… tk tk+1 ……………………. tm …. tn EOF

<u>| while id < E do S od </u>|

S

Structure of parse_X()

Function

```
parse_S()  {
        t = next_token();
        switch(t) {
                case id:     match(=);
                              Parse_E();
                              break;
                case while:   match(id);
                             match(<);
                             parse_E();
                             match(do);
                             parse_S();
                             match(od);
                             break;
                default:     error();
        }
}
```

Explanation

- Call to parse_S(): <u>prediction</u> of a fragment derived from S
- <u>case id</u>: the next token is of type id
- <u>Check</u>: **can derivation from S start with id ?**
  id $\in$ First(S) ?
- Possible, if the rule S → id = E is used
- <u>Update the prediction</u>: after id, token = is expected, and then a fragment derived from E.

First(X)

First(X) = {t1, t2, … , tn}

Group of tokens such that for every i $(1 \leq i \leq n)$, there exists a derivation from X that starts with ti .

In other words: derivation from X may start with ti

Example: First(S) = {id , while}

Challenges

- Given G, how to compute First(X) for every variable X in G ?
  (simple if every rule for X starts with a token)
- What if a rule starts with a variable, e.g. $X \rightarrow Y\ t1\ t2\ Z$?

Dealing with ε-rules

Example

1. S → a S b
2. S → #
3. S → ε

- when parser applies the rule S → ε ?
- ε is not a token ! (hence $\varepsilon \notin First(S)$)

NOTE: also b , EOF $\notin$ First(S)

```
void parse_S()
        t = next_token();
        switch(t -> kind) {
        case a:   { print ("Rule 1"); parse_S();
                    match(b); break }
        case #:   { print ("Rule 2"); break; }
        case b: error()  ???
        case EOF: error()  ???
        }
}
```

Dealing with ε-rules

Consider:

t1 t2 t3 …… tk b … tm…. tn EOF

^ |_____|

S

Is this an error? Reminder: $b \notin First(S)$

```
void parse_S()
        t = next_token();
        switch(t -> kind) {
        case a:   { print ("Rule 1"); parse_S();
                    match(b); break }
        case #:   { print ("Rule 2"); break; }
        case b:  error() ???
        case EOF: error() ???
        }
}
```

Can apply here the rule $S \rightarrow \varepsilon$!

t1 t2 t3 …… tk b … tm…. tn EOF

^ |

ε

In fact, after tk there is something (ε) correctly derived from S – as predicted!

Dealing with ε-rules

In fact, after tk there is something correctly derived from S

t1 t2 t3 …… tk b … tm…. tn EOF

^ |

ε

This is correct <u>only</u> if token b can appear

<u>after</u> derivation from S is completed

See rule 1 - this indeed is possible $b \in$ Follow(S)

1. S → a S b
2. S → #
3. S → ε

Similar – for EOF

Dealing with ε-rules

```
void parse_S()
            t = next_token();
            switch(t) {
            case a:     { print ("Rule 1");  parse_S();
      match(b); break }
            case #:     { print ("Rule 2");  break; }
            case b:  error()  ???
            case EOF: error()  ???
            }
}
```

```
void parse_S()
            t = next_token();
            switch(t -> kind) {
            case a:     { print ("Rule 1");  parse_S();
                          match(b); break }
            case #:     { print ("Rule 2");  break; }
            case b:  case EOF:
                        { print ("Rule 3");
                          t = back_token();  break; }
            }
}
/* NOTE:  cases also for tokens in Follow(S) !  */
```

Why back_token?

- next_token returned token b
- b is not a part of what is derived from S (derived ε)
- parser made a step **beyond** S
- since parser is still inside parse_S (means: inside S) , it should step back

Follow(X)

Follow(X) = {t1, t2, … , tn}

Group of tokens such that for every i $(1 \leq i \leq n)$, there exists a derivation in which t_i appears after X

Example: Follow(S) = {b , EOF}

Challenges

- Given G, how to compute Follow(X) for every variable X in G ?
  (simple if every appearance of X is immediately followed by a token)
- What to do in cases such as $Y \rightarrow t1\ X\ Z$, or $Y \rightarrow t1\ X$?

Can recursive descent parser be constructed for every grammar?

No, there are some obstacles to be eliminated:

- Left common prefixes
- Left recursion

Left common prefixes – why this is a problem?

Example

DECL → TYPE LIST

TYPE → int | real

LIST → id | id , LIST

Left common prefix in rules for LIST:

two rules for <u>same</u> variable have a common beginning

```
parse_LIST()
{
    t = next_token();
    switch (t -> kind) {
      case id :  ????   /*  id ∈ First(LIST) ,  but...  not clear which rule for LIST to apply  */
      default : error()
    }
}
```

Elimination of left common prefixes

$X \rightarrow \alpha\ \beta_1$
$X \rightarrow \alpha\ \beta_2$
.....
$X \rightarrow \alpha\ \beta_n$

$\Rightarrow$

$X \rightarrow \alpha\ X'$
$X' \rightarrow \beta_1$
$X' \rightarrow \beta_2$
.....
$X' \rightarrow \beta_n$

Need:

- add a new variable X'
- replace rules by new ones

(an ε-rule may appear !)

Example

DECL → TYPE LIST
TYPE → int | real
LIST → id | id , LIST

$\Rightarrow$

DECL → TYPE LIST
TYPE → int | real
LIST → id LIST'
LIST' → ε | , LIST

Left recursion – why this is a problem?

Example

DECL → TYPE LIST
TYPE → int | real
LIST → id | LIST , id

Left recursion on LIST

```
parse_LIST()
{
    t = next_token();
    switch (t -> kind) {
      case id :  ????   not clear which rule for LIST to apply
      default : error()
    }
}
```

Elimination of left recursion

$X \rightarrow X\alpha$
$X \rightarrow \beta$

Generates sequences of the form $\beta\alpha^n$ $(n \geq 0)$

First, α's are produced; at the end of derivation β is added

Examples: $X \rightarrow \beta$ $X \rightarrow X\alpha \rightarrow \beta\alpha$ $X \rightarrow X\alpha \rightarrow X\alpha\alpha \rightarrow \beta\alpha\alpha$

Idea: first produce β, and then continue to production of α's

$X \rightarrow X\alpha$
$X \rightarrow \beta$

$\Longrightarrow$

$X \rightarrow \beta X'$
$X' \rightarrow \varepsilon$
$X' \rightarrow \alpha X'$

Need:

- add a new variable X'
- replace rules by new ones (ε-rule must appear!)

$X \rightarrow \beta X' \rightarrow (X' \rightarrow \varepsilon) \rightarrow \beta$

$X \rightarrow \beta X' \rightarrow (X' \rightarrow \alpha X') \rightarrow \beta\alpha X' \rightarrow (X' \rightarrow \varepsilon) \rightarrow \beta\alpha$

$X \rightarrow \beta X' \rightarrow (X' \rightarrow \alpha X') \rightarrow \beta\alpha X' \rightarrow (X' \rightarrow \alpha X') \rightarrow \beta\alpha\alpha X'$ $(X' \rightarrow \varepsilon) \rightarrow \beta\alpha\alpha$

Elimination of left recursion - example

$X \rightarrow X\alpha$
$X \rightarrow \beta$

$\Rightarrow$

$X \rightarrow \beta X'$
$X' \rightarrow \varepsilon$
$X' \rightarrow \alpha X'$

DECL → TYPE LIST
TYPE → int | real
LIST → id | LIST , id

$\Rightarrow$

DECL → TYPE LIST
TYPE → int | real
LIST → id LIST'
LIST' → ε | LIST' → , id LIST'

Left recursion on LIST : α = , id ; β = id