Recursive descent parsing

Each language L over alphabet A has an associated **recognition problem**: Given a finite sequence in A^*, determine whether it is in L.

Many, but not all, context free languages can be recognized using a simple technique called **recursive descent parsing**.

Definition t **is a prefix of** s if and only if there is a u such that $s = tu$.

The idea is this:

- Start with a suitable CFG $(A, N, P, n_{\text{start}})$ for L
- For each nonterminal n in N create a procedure n
- Roughly speaking, the job of procedure n is to try to remove from the input a suitable prefix described by nonterminal n.
- If there is no suitable prefix, the procedure may indicate failure by setting a flag f to false.
- We use variable s to represent the remaining input sequence.
- We'll mark the end of input with a sentinel symbol $\$$ not in $A \cup N$.

Example: (tree is the start nonterminal)

$$\begin{aligned} \text{tree} &\rightarrow [\ \text{moreTree} \\ \text{tree} &\rightarrow \textbf{id} \\ \text{moreTree} &\rightarrow] \\ \text{moreTree} &\rightarrow \text{tree moreTree} \end{aligned}$$

Variables:

- f is set to false if an error is encountered
- s is the remaining input. Ends with a $\$$.
- We assume that $t \in A^*$; so there is no $\$$ in t.

The main code. Is t in the language?

$f := \text{true}$ $\quad s := t\hat{}[\$]$ $\quad$ *tree*$()$ $\quad f := f \wedge (s(0) = \$)$

Where the procedures are

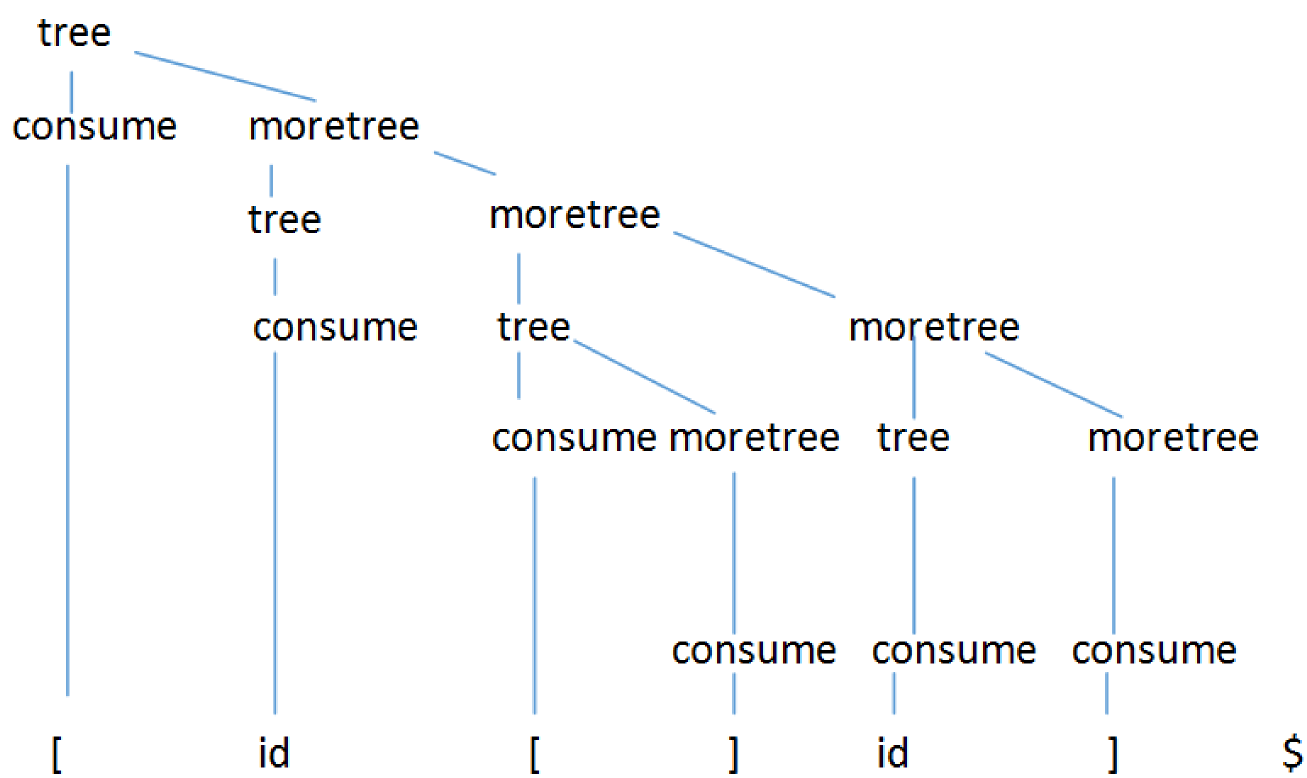
proc *tree*() // *Try to remove a prefix described by* tree.
  if $\neg f$ then return end if
  if $s(0) = [$ then *consume*() *moreTree*()
  else *expect*(**id**) end if
end *tree*

proc *moreTree*()
// *Try to remove a prefix described by* moreTree.
  if $\neg f$ then return end if
  if $s(0) =]$ then *consume*()
  else *tree*() *moreTree*() end if
end *moreTree*

proc *consume*() $s := s[1, ..s.\text{length}]$ end *consume*

proc *expect*(a)
  if $s(0) = a$ then *consume*() else $f := \text{false}$ end if
end *expect*

Here is an example call tree showing how this work in a successful recognition. Note how the call tree mimics the parse tree.



Specification of nonterminal procedures

The specification for procedures representing nonterminals

procedure $n()$ // *Try to remove a prefix described by* n
precondition: s is nonempty and ends with a \$
changes s, f
postcondition:
There are two possible outcomes

- Error: f is false and s still ends with a \$.
- Success: f is true and a prefix of s_0, described by n, has been removed. I.e., $\exists u \cdot s_0 = us$ and $n \overset{*}{\Longrightarrow} u$.

Choosing an outcome:

- If f_0 is false, Error is the only possible outcome.
- If f_0 is true but no prefix of s_0 is described by n, Error is the only possible outcome.
- If f_0 is true and $\exists t, u, v \in A^* \cdot n_{\text{start}}\$ \overset{*}{\Longrightarrow} tnv\$ \overset{*}{\Longrightarrow} tuv\$$ and $uv\$ = s_0$, then Success is the only possible outcome (and the prefix u removed should meet these conditions).
- Otherwise it doesn't matter which outcome is chosen.

Now assume ~~the initial value of~~ $\cancel{s}\ t \in A^*$. We can tell if $\cancel{s}\ t$ is in L as follows

$$f := \text{true}; \quad s := \cancel{s}\ t\,\hat{}\,[\$]; \quad n_{\text{start}}(); \quad f := f \wedge (s(0) = \$)$$

Some handy procedures

procedure *expect*($a : A$)
  // Try to remove a from the start of the s.
  precondition: s ~~contains~~ ends with a \$
  changes s, f
  postcondition:
    if f_0 and ($[a]$ is a prefix of s_0)
    then f and $s_0 = [a]\hat{}\,s$
    else $\neg f$ and s ~~contains~~ ends with a \$
  if $s(0) = a$ then *consume*()
  else $f :=$ false end if
end *expect*

procedure *consume*()
  // Remove the first item from s
  precondition: $s.\mathrm{length} > 0$ and $s(0) \in A$
  changes s
  postcondition: $s = s_0[1, ..s_0.\mathrm{length}]$
  $s := s[1, ..s.\mathrm{length}]$
end *consume*

Writing procedures that meet the specification

If a nonterminal n has productions

$$(n \to \alpha), (n \to \beta), (n \to \gamma) \in P,$$

we write a subroutine like this:

procedure $n()$
  // For specification see slide 4
  if $\neg f$ then return end if
  if ? then $[\![\alpha]\!]$
  else if ? then $[\![\beta]\!]$
  else if ? then $[\![\gamma]\!]$
  else $f := \text{false}$ end if
end n

where, for $a \in A$, $m \in N$, $\alpha, \beta \in (A \cup N)^*$

$$\begin{aligned} [\![a]\!] &= \text{“}\mathit{expect}(a)\text{”} \\ [\![m]\!] &= \text{“}m()\text{”} \\ [\![\epsilon]\!] &= \epsilon \\ [\![\alpha\beta]\!] &= [\![\alpha]\!] \hat{} [\![\beta]\!] \end{aligned}$$

- Usually the boolean expressions are based on the first few items of s.
- The last case $f := \text{false}$ might be unreachable; in this case it is omitted.
- Note that $\{\neg f\}\ [\![\alpha]\!]\ \{\neg f\}$ is correct

Parsing our programming language

var $s : A^*\cdot$
var $f : \mathbb{B}\cdot$
procedure $main()$
    read the input into t, combining characters into symbols and throwing out comments and spaces
    $f := \mathrm{true}$
    $s := t\,\hat{}\,[\$]$
    $block()$
    $f := f \wedge (s(0) = \$)$
    { $f = (t$ is in the programing language$)$ }
    if f then print “yep” else print “nope” end if
end *main*

Nonterminal block

$$\text{block} \rightarrow \epsilon$$
$$\text{block} \rightarrow \text{command block}$$

procedure *block*() // Version 0
    // Try to remove a prefix described by *block* .
    // See the contract on slide 4
    if $\neg f$ then return end if
    if $s(0) \in FirstComm$ then
        command() *block*()
    end if
end *block*

where $FirstComm$ is $\{\mathbf{if}, \mathbf{while}\} \cup \mathcal{I}$.

Why this works:

- When $\text{block} \rightarrow \text{command block}$ is appropriate, $s(0)$ is in $\{\mathbf{if}, \mathbf{while}\} \cup \mathcal{I}$;
  - ∗ you can see this by looking at all the productions for $command$.
- When $\text{block} \rightarrow \epsilon$ is appropriate, $s(0) \in \{\$, \mathbf{end}, \mathbf{else}\}$;
  - ∗ you can see this by looking at all the places $block$ is used in the grammar;
  - ∗ thus $s(0)$ is not in $\{\mathbf{if}, \mathbf{while}\} \cup \mathcal{I}$.
- Thus it is never right to pick the $\text{block} \rightarrow \epsilon$ production when $s(0)$ is in $FirstComm$

Note that we can apply tail recursion removal, if we want.

procedure *block*() // Version 1
    // Try to remove a prefix described by *block* .
    // See the contract on slide 4
    while $f \wedge s(0) \in \{\mathbf{if}, \mathbf{while}\} \cup \mathcal{I}$ do
        command()
    end while
end *block*

Also acceptable would be

procedure *block*() // Version 2
    // Try to remove a prefix described by *block* .
    // See the contract on slide 4
    while $f \wedge s(0) \in \{\mathbf{if}, \mathbf{while}\} \cup \mathcal{I}$ do
        command()
    end while
    if $s(0) \notin \{\$, \mathbf{end}, \mathbf{else}\}$ then $f := \mathrm{false}$ end if
end *block*

We can either detect the error here (Version 2) or leave the error to be detected later (Versions 0 and 1).

The command nonterminal

$$\text{command} \rightarrow i := \text{exp} \qquad \text{for all } i \in \mathcal{I}$$
$$\text{command} \rightarrow \textbf{if}\ \text{exp}\ \textbf{then}\ \text{block}\ \textbf{else}\ \text{block}\ \textbf{end if}$$
$$\text{command} \rightarrow \textbf{while}\ \text{exp}\ \textbf{do}\ \text{block}\ \textbf{end while}$$

procedure *command*()
  // Try to remove the a prefix described by *command* .
  // See the contract for n a few slides back.
  if $\neg f$ then return end if
  if $s(0) = \textbf{if}$ then
    consume() *exp*() *expect*(**then**) *block*() *expect*(**else**)
    block() *expect*(**end**) *expect*(**if**)
  elseif $s(0) = \textbf{while}$ then
    consume() *exp*() *expect*(**do**) *block*() *expect*(**end**)
    expect(**while**)
  else if $s(0) \in \mathcal{I}$ then
    consume() *expect*(**:=**) *exp*()
  else
    $f := \text{false}$
  end if
end *command*

Parsing expressions

Recall that the rules for expressions are

$$\begin{aligned} \text{exp} &\to \text{comparand} \\ \text{exp} &\to \text{comparand} < \text{comparand} \end{aligned}$$

Rewrite these rules to postpone the decision about which production to use until it matters

$$\begin{aligned} \text{exp} &\to \text{comparand exp0} \\ \text{exp0} &\to \epsilon \\ \text{exp0} &\to < \text{comparand} \end{aligned}$$

Write the procedures

procedure *exp*()
  // Try to remove a prefix described by *exp*.
  if $\neg f$ then return end if
  comparand()   *exp0*()
end *exp*

procedure *exp0*()
  // Try to remove a prefix described by *exp0*
  if $s(0) = <$ then *consume*() *comparand*() end if
end *exp0*

In-line the call to exp0 to get

procedure *exp*()
  // Try to remove a prefix described by *exp*.
  if $\neg f$ then return end if
  comparand()
  if $s(0) = <$ then *consume*() *comparand*() end if
end *exp*

$$\begin{aligned}
\text{comparand} &\rightarrow \text{term} \\
\text{comparand} &\rightarrow \text{term} + \text{comparand} \\
\text{comparand} &\rightarrow \text{term} - \text{comparand}
\end{aligned}$$

rewrite as

$$\begin{aligned}
\text{comparand} &\rightarrow \text{term comparand0} \\
\text{comparand0} &\rightarrow + \text{ term comparand0} \\
\text{comparand0} &\rightarrow - \text{ term comparand0} \\
\text{comparand0} &\rightarrow \epsilon
\end{aligned}$$

Write the procedures

procedure *comparand*()
  // Try to remove a prefix described by *comparand.*
  if $\neg f$ then return end if
  term()   *comparand0*()
end *comparand*

procedure *comparand0*()
  // Try to remove a prefix described by *comparand0.*
  if $\neg f$ then return end if
  if $s(0) \in \{+, -\}$ then *consume*() *term*() *comparand0*()
  end if
end *comparand*

After tail recursion removal and inlining, we have

procedure *comparand*()
  // Try to remove a prefix described by *comparand.*
  if $\neg f$ then return end if
  term()
  while $f \wedge s(0) \in \{+, -\}$ do *consume*() *term*() end while
end *comparand*

Term is similar to comparand

$$\begin{aligned}\text{term} &\rightarrow \text{factor}\\ \text{term} &\rightarrow \text{factor} * \text{term}\\ \text{term} &\rightarrow \text{factor} \,/\, \text{term}\end{aligned}$$

procedure *term*()
  // Try to remove a prefix described by *term*.
  if $\neg f$ then return end if
  factor()
  while $f \wedge s(0) \in \{*, /\}$ do *consume*() *factor*() end while
end *term*

$$\begin{aligned}\text{factor} &\rightarrow n && \text{for all}^{1}\ n \in \mathcal{N}\\ \text{factor} &\rightarrow i && \text{for all } i \in \mathcal{I}\\ \text{factor} &\rightarrow (\,\text{exp}\,)\end{aligned}$$

procedure *factor*()
  // Try to remove a prefix described by *factor*.
  if $\neg f$ then return end if
  if $s(0) \in \mathcal{N}$ then *consume*()
  elseif $s(0) \in \mathcal{I}$ then *consume*()
  elseif $s(0) = ($ then *consume*() *exp*() *expect*( ))
  else $f :=$ false
  end if
end *factor*

Exercise: find a variant expression that shows that we have no infinite loops or infinite recursion.

[1] Recall that $\mathcal{N}$ is a finite subset of $\mathbb{N}$.

Generating machine code for expressions

Suppose we want to compile code for a stack machine

- The job of the code generated by procedures $factor$, $term$, $comparand$, and exp is to push a value.
- We'll ignore type checking and existence of variables
- We need the following instruction sequences
  - $*$ $\mathrm{push}(n)$ pushes a number n on to the stack
  - $*$ $\mathrm{fetch}(i)$ pushes the value of variable i onto the stack
  - $*$ mul pops two values off the stack, multiplies them and pushes the result. div is similar to mul

```
procedure factor()
    if ¬f then return end if
    if s(0) ∈ 𝒩 then m := m^push(s(0)) consume()
    elseif s(0) ∈ ℐ then m := m^fetch(s(0)) consume()
    elseif s(0) = ( then consume() exp() expect( ) )
    else f := false end if
end factor
```

$term$, $comparand$, and exp are similar to each other

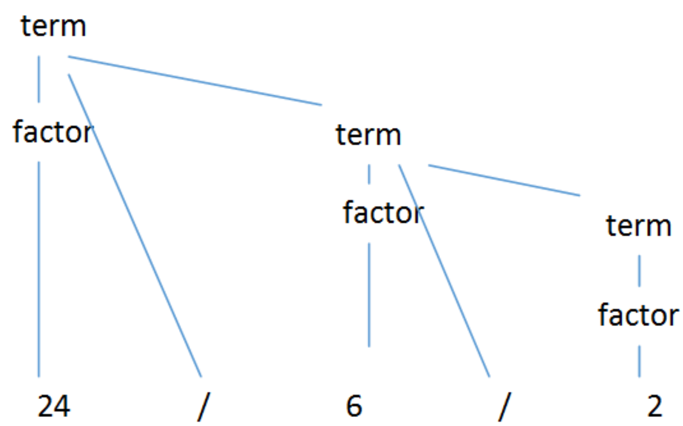
```
procedure term()
    if ¬f then return end if
    factor()
    while f ∧ s(0) ∈ {∗, /} do
        val op := s(0)      consume()      factor()
        if op = ∗ then m := m^mul else m := m^div end if
    end while
end term
```

What about associativity?

We want $-$ and $/$ to be left associative. E.g., $24/6/2$ should generate the same code as $(24/6)/2$.

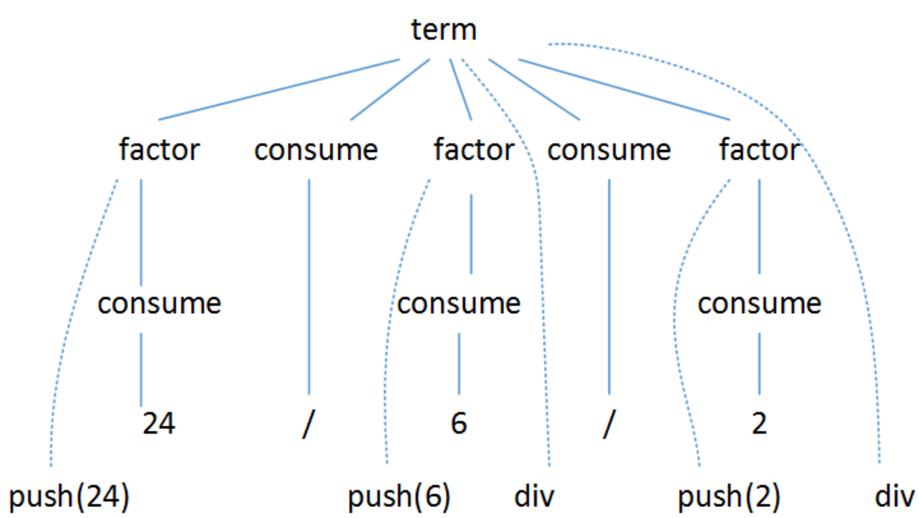
Our original grammar gets associativity “wrong” for $/$ and $-$.

Consider the parse tree for $term \stackrel{*}{\Longrightarrow} 24/6/2$.



This seems to associate the /s the wrong way.

However, if you trace the actions of the compiler, you will see that the code generated for $24/6/2$ is correct because the operation is emitted at the right time.



If we look at a version without tail-call optimization, the choice is clearer.

procedure *term*()
    if $\neg f$ then return end if
    factor()
    term0()
end *term*
procedure *term0*()
    if $\neg f$ then return end if
    if $s(0) \in \{*, /\}$ then
        val $op := s(0)$ *consume*()
        factor()
        // (a) emit instruction here for left associativity
        term0()
        // (b) emit instruction here for right associativity
    end if
end *term0*

Precedence

We need that $a + b * c + d * e$ generates the same code as $a + (b * c) + (d * e)$. Because of the way the grammar treats expressions, it does.

Generating code for assignment commands

Instruction

- $\mathrm{store}(i)$ pops a value off the stack and stores it in the location for identifier i.

procedure *command*()

  ...

  elseif $s(0) \in \mathcal{I}$ then

    val $i := s(0)$ *consume*()

    expect(**:=**)

    exp()

    $m := m\hat{\ }\mathrm{store}(i)$

  else

  ...

Generating code for while commands

Instructions:

- $\mathrm{branch}(a)$ branches to instruction a
- $\mathrm{condBranch}(d)$ pops the stack and branches to d if the former top was false.
- I'll assume that the length of $\mathrm{condBranch}(d)$ does not depend on d.

If the expression compiles to a sequence x and the block compiles to a sequence y, the while-loop compiles to a sequence

$$
\begin{array}{rcl}
a & : & x \\
b & : & \mathrm{condBranch}(d) \\
c & : & y \\
 & & \mathrm{branch}(a) \\
d & : &
\end{array}
$$

procedure *command*()

  ...

  elseif $s(0) = \mathbf{while}$ then *consume*()

    val $a := m.\mathrm{length}$ *exp*() *expect*(**do**)

    val $b := m.\mathrm{length}$ $m := m\hat{}\mathrm{condBranch}(0)$

    val $c := m.\mathrm{length}$ *block*()

    $m := m\hat{}\mathrm{branch}(a)$

    val $d := m.\mathrm{length}$ $m[b, ..c] := \mathrm{condBranch}(d)$

    expect(**end**) *expect*(**while**)

  elseif

  ...

The rest of the compiler

I'll leave the rest of the compiler as an exercise:

- If commands,
- expression
- comparand
- block

Going further: Think about how you could

- Add variable declarations
- Add simple types and type checking
- Add procedures and procedure calls
- Add arrays
- Add classes and objects

When can we use recursive descent?

When can we use recursive descent parsing?

When it is possible to choose between the productions for a nonterminal based on

- Information already seen
- The next few symbols of input

In particular there is a set of grammars for which RDP is particularly easy. These grammars allow the choice to be made by looking only at the next item of input.

Such a grammar is called “LL(1)”.

LL(1)

Recall: If a nonterminal n has productions

$$(n \to \alpha), (n \to \beta), (n \to \gamma) \in P,$$

we write a subroutine like this:

procedure $n()$
// Try to remove a prefix described by n .
  if $\neg f$ then return end if
  if ? then $[\![\alpha]\!]$ else if ? then $[\![\beta]\!]$ else if ? then $[\![\gamma]\!]$
  else $f := \mathrm{false}$ end if
end

Often the guard only needs to look at the next input symbol.

Associate with each production $n \to \alpha$ with a "selector set" $\mathrm{sel}(n \to \alpha) \subseteq A \cup \{\$\}$

procedure $n()$
// Try to remove a prefix described by n .
  if $\neg f$ then return end if
  if $s(0) \in \mathrm{sel}(n \to \alpha)$ then $[\![\alpha]\!]$
  else if $s(0) \in \mathrm{sel}(n \to \beta)$ then $[\![\beta]\!]$
  else if $s(0) \in \mathrm{sel}(n \to \gamma)$ then $[\![\gamma]\!]$
  else $f := \mathrm{false}$ end if
end n

If for all distinct productions $n \to \alpha$, $n \to \beta$, $\mathrm{sel}(n \to \alpha) \cap \mathrm{sel}(n \to \beta) = \emptyset$, then the grammar is called $LL(1)$, and we can write a recursive descent parser for it.

Computing selector sets:

- First symbols: If $\alpha \stackrel{*}{\Longrightarrow} at$ with $a \in A$ then $a \in \mathrm{sel}(n \to \alpha)$
- Following symbols: $a \in \mathrm{sel}(n \to \alpha)$ if $\alpha \stackrel{*}{\Longrightarrow} \epsilon$ and $a \in A \cup \{\$\}$ can follow n in a derivation from $n_{\mathrm{start}}\$$ — i.e. if there is a derivation
$$n_{\mathrm{start}}\$ \stackrel{*}{\Longrightarrow} tnau \Longrightarrow t\alpha au \stackrel{*}{\Longrightarrow} tau$$
with $t \in A^*$, $u \in (A \cup \{\$\})^*$.
- Nothing else is in $\mathrm{sel}(n \to \alpha)$

Example: The start symbol is B

$$\begin{array}{lll} B \to CB & & B \to \epsilon \\ C \to \mathbf{id} := E & & C \to \mathbf{if}\ E\ \mathbf{then}\ B\ D\ \mathbf{end\ if} \\ D \to \mathbf{else}\ B & & D \to \epsilon \\ E \to \mathbf{id} & & \end{array}$$

The selector set of $B \to CB$ is the first symbols of CB which are $\{\mathbf{id}, \mathbf{if}\}$.

The selector set of $B \to \epsilon$ is the symbols that can follow B which are $\{\mathbf{else}, \mathbf{end}, \$\}$.

Exercise. Show that the following grammar, with start symbol B, is not LL(1)

$$\begin{array}{lll} B \to CB & B \to \epsilon & \\ C \to \mathbf{id} := E & C \to \{B\} & C \to \mathbf{if}\ E\ \mathbf{then}\ C\ D \\ D \to \mathbf{else}\ C & D \to \epsilon & \\ E \to \mathbf{id} & & \end{array}$$

If a grammar is not LL(1), we can still often use recursive descent, e.g., by looking more symbols ahead.

Here are a few tricks of the trade to make a grammar LL(1), or at least more suitable for RDP.

- Factor: Example: Replace

$$\begin{aligned} \text{command} &\rightarrow \textbf{id} := \text{exp} \\ \text{command} &\rightarrow \textbf{id} \text{ (args)} \end{aligned}$$

  with

$$\begin{aligned} \text{command} &\rightarrow \textbf{id} \text{ more} \\ \text{more} &\rightarrow := \text{exp} \\ \text{more} &\rightarrow \text{(args)} \end{aligned}$$

  More generally, replace productions

$$\begin{aligned} n &\rightarrow \alpha \cancel{a} \beta \\ n &\rightarrow \alpha \cancel{b} \gamma, \end{aligned}$$

  where ~~$a, b \in A$ and~~ $\alpha, \beta, \gamma \in (A \cup N)^*$, with

$$\begin{aligned} n &\rightarrow \alpha p \\ p &\rightarrow \cancel{a} \beta \\ p &\rightarrow \cancel{b} \gamma, \end{aligned}$$

  where p is a fresh nonterminal.

- Remove left recursion: Example: Replace

$$\begin{aligned}\text{type} &\rightarrow \text{type}\,[\,]\\ \text{type} &\rightarrow \textbf{int}\end{aligned}$$

with

$$\begin{aligned}\text{type} &\rightarrow \textbf{int}\ \text{type0}\\ \text{type0} &\rightarrow [\,]\ \text{type0}\\ \text{type0} &\rightarrow \epsilon\end{aligned}$$

More generally, replace

$$\begin{aligned}n &\rightarrow n\alpha\\ n &\rightarrow \beta,\end{aligned}$$

where $\alpha, \beta \in (A \cup N)^*$, with

$$\begin{aligned}n &\rightarrow \beta p\\ p &\rightarrow \alpha p\\ p &\rightarrow \epsilon,\end{aligned}$$

where p is a fresh nonterminal.

Most formats can be parsed by recursive descent, one way or another.

Tools

While writing recursive descent parsers is straightforward for simple grammars, it can be error prone and tedious as grammars evolve and get larger.

Luckily there are a large number of tools that convert grammars to parsers. Examples:

- JavaCC.
  - ∗ Allows grammars in which the RHS of each production is a regular expression.
  - ∗ Produces recursive descent parsers written in Java or C++.
  - ∗ Calculates the guard expressions automatically for most grammars
  - ∗ Allows the programmer to intervene in cases the automatic rules don't handle
  - ∗ Allows the programmer to annotate the grammar with bits of Java (or C++) code that are interpolated into the parser.
- ANTLR 4
  - ∗ Similar to JavaCC
  - ∗ Automatic treatment of left recursion and operator precedence
- Yacc/Bison
  - ∗ Produces bottom-up parsers
  - ∗ Handles a large class of grammars automatically
  - ∗ No need to factor or remove left recursion