CS109B Notes for Lecture 5/5/95

Recursive-Descent Parsing

- Write one function A for each SC $< A >$.
  - Its goal is to consume a prefix of the available input (string of terminals) and return a parse tree with root $< A >$ and the consumed input as yield.
  - It also needs to return the unconsumed input.
- Function A first decides which production for $< A >$ to use at the root of the tree.
  - It may only use the next input symbol to decide.
- Having decided, A checks for each element of the selected production body, in turn.
  - Terminal?: check it is next on the input and consume it.
  - SC?: Call its function on the same input, then proceed on whatever remaining input is returned.

Why Recursive-Descent parsing?

- A simple-to-implement method that works for many realistic languages, provided the grammar is manipulated somewhat, as below.

Example: Here is a grammar for ML tuples, using terminal a ("atom") for all non-tuple components.

$< tuple > \rightarrow$ ($< elList >$)

$< elList > \rightarrow < element >$, $< elList >$
$< elList > \rightarrow < element >$

$< element > \rightarrow < tuple >$
$< element > \rightarrow$ `a`

Left-Factoring

For above example grammar:

- $<$ *tuple* $>$ gives us no choice of production.
- For $<$ *element* $>$, choose `a` on input `a`. Choose $<$ *tuple* $>$ on any input symbol that could be first in a string of $L(<tuple>)$, namely on `)` alone.
- But what to choose for $<$ *elList* $>$? Both bodies begin with $<$ *element* $>$, so the input gives no clue.
- Trick: *left-factor* the productions by introducing a new SC $<$ *tail* $>$ that generates the "tail" of either body, i.e., whatever follows $<$ *element* $>$ in that production.

(1) $<$ *tuple* $>$ $\rightarrow$ `(` $<$ *elList* $>$ `)`

(2) $<$ *elList* $>$ $\rightarrow$ $<$ *element* $>$ $<$ *tail* $>$

(3) $<$ *tail* $>$ $\rightarrow$ `,` $<$ *elList* $>$

(4) $<$ *tail* $>$ $\rightarrow$ ϵ

(5) $<$ *element* $>$ $\rightarrow$ $<$ *tuple* $>$

(6) $<$ *element* $>$ $\rightarrow$ `a`

Representing Parse Trees in ML

To complete our example, we'll write the functions for this grammar in ML. Below is a datatype `PT` for parse trees.

- The first component is always a node label.
  - e.g., `Leaf(",")` is a leaf node labeled comma.
- The constructors besides `Leaf` represent interior nodes with a label and 1–3 subtrees.

```
datatype PT =
    Three of string * PT * PT * PT |
    Two of string * PT * PT |
    One of string * PT |
    Leaf of string;
```

```
fun tuple("("::xs) =
        let
            val (ys,t) = elList(xs);
        in
            case ys of
                nil => raise Fail |
                (")"::zs) =>
                    (zs, Three("tuple", Leaf("("), t, Leaf(")"))) |
                _ => raise Fail
        end
    | tuple(_) = raise Fail
and
    elList(x::xs) =
        if x="(" orelse x="a" then
            let
                val (ys,t1) = element(x::xs);
                val (zs,t2) = tail(ys);
            in
                (zs, Two("elList", t1, t2))
            end
        else raise Fail
    | elList(_) = raise Fail
and
    element("a"::xs) = (xs, One("elmnt", Leaf("a")))
    | element(xs) =
        let
            val (ys,t) = tuple(xs);
        in
            (ys, One("elmnt", t))
        end
and
    tail(","::xs) =
        let
            val (ys,t) = elList(xs);
        in
            (ys, Two("tail", Leaf(","), t))
        end
    | tail(xs) = (xs, One("tail", Leaf("epsln")));

fun parse(s) =
        let
            val (ys,t) = tuple(explode(s));
        in
            printT(0,t)
        end;

parse("((a,a),(a,a))");
```

We also need the following exception to handle the case where the input is not in the language $L(<tuple>)$.

```
exception Fail;
```

- Code on p. 3, discussed in class.
- Critical decision: expanding $< tail >$, clearly production (3) is right on comma input. Production (4) is right only on symbols that can follow a "tail." That is only right-paren. Why?

Table-Driven Parser

Instead of mutually recursive functions, we can summarize the decisions in a table and write one program that will examine any table and any input and try to parse the input according to the table.

Example: For our grammar:

	a	,	(	)
$< tuple >$			1	
$< elList >$	2		2	
$< tail >$		3		4
$< element >$	6		5	

Parser Architecture

1. A stack of SC's and terminals representing goals that need to be found on the input.
   - Initially, stack consists of one SC, the one that represents the language being parsed.
2. A list of terminals: the remaining input characters.
   - Often, it is necessary to follow the input by an *endmarker* character, denoted `ENDM` in FCS. Our present example doesn't happen to need it, because the balancing right parenthesis tips the parser off that the end has been reached.