

Data Structures and Algorithms II

Lecture 8:

Recursive Descent Parsing and Parse Trees

- Recursive Descent Parsing.
- Returning a Parse Tree.

1

The Match Function

Match is a function that checks that the next token matches the (enumerated) type of its argument, and then gets the next token.

It can be quite simple. It assumes there is some global variable `nexttoken`, which has fields `type` and `text` which correspond to the next token in the input stream:

```
void match(TokenType t)
{
    if(nexttoken.type==t)
        getnexttoken();
    else error();
}
```

`getnexttoken` is the *scanner*. It obtains the next token in the file of characters. It may be quite complex, but here's a simple example if we assume each token is a single character, and no spaces:

3

Recursive Descent Parsing

How can top-down parser for given grammar be written?

Recursive descent parsers implement the grammar rules directly as functions. Given two rules, say:

```
term -> '(' expr ')'
term -> NUM
```

and token types: `OPENPAREN`, `CLOSEPAREN`, `NUM`, `OP`. Write a function called `term` that calls another function called `expr` as follows:

```
void term()
{
    if(nexttoken.type==OPENPAREN)
    {
        match(OPENPAREN);
        expr();
        match(CLOSEPAREN);
    }
    else if (nexttoken.type==NUM)
        match(NUM);
    else error(); // some error function
}
```

2

```
void getnexttoken()
{
    char c=cin.get();
    nexttoken.text[0]=c; nexttoken.text[1]='\0';
    switch (c)
    {
        case '+' : case '-' :
            nexttoken.type=OP; break;
        case '(' : nexttoken.type=OPENPAREN; break;
        case ')' : nexttoken.type=CLOSEPAREN; break;
        default: nexttoken.type=NUM;
    }
}
```

The error function could be quite simple:

```
void error()
{
    cerr << "Parse error - giving up";
    exit(0);
}
```

4

Example

In general all the tokens corresponding to LHS of a rule should have a corresponding function:

```
term -> '(' expr ')'  
term -> NUM  
expr -> NUM OP NUM
```

would have two functions:

```
void term()  
{  
  .. (as before)  
}
```

```
void expr()  
{  
  match(NUM);  
  match(OP);  
  match(NUM);  
}
```

5

If the symbol on the LHS can be repeated, say:

```
s -> n*  
n -> NUM ..
```

then we can have a loop:

```
void s()  
{  
  while(nexttoken.type==NUM)  
    n();  
}
```

as NUM is first symbol on LHS of n.

If there are several rules corresponding to a given symbol then we will have a function with an if statement, where it decides which rule to use based on the type of the *nexttoken* in the input. (See example on slide 2).

7

Generalising a Bit

How do we write the functions for a recursive descent parser?

Suppose we have a single rule starting with the symbol *s*. We write a function called *s*. For each symbol on the RHS we have a statement in the function.

- If that RHS symbol is a terminal symbol we call the “match” function to check it is the right type, and advance the input (*getnexttoken*).
- If that symbol is a non terminal symbol (say, *n*) we call a function *n* ().

So.. *s* -> NUM *n*
would result in a rule:

```
void s()  
{  
  match(NUM);  
  n();  
}
```

6

Generalising..

This leads to the following basic rule structure:

```
void n()  
{  
  if (next token can start first rule for n)  
    match rule 1 for n  
  else if (next token can start second rule  
    for n)  
    match rule 2 for n  
    ... ..  
  else if (next token can start i'th rule for n)  
    match rule i for n  
  else  
    error();  
}
```

(This scheme will only work if rules are NOT left recursive, and if one token lookahead is enough to choose between two rules.)

8

Example

```
term -> '(' expr ')'  
term -> NUM
```

Constructing rule for term:

```
void term()  
{  
    if(nexttoken.type=OPENPAREN)  
        // next token can start first rule for term  
        {  
            match(OPENPAREN);  
            expr();  
            match(CLOSEPAREN);  
            // so match terminal symbols, call fns  
            // for non terminals  
        }  
    else if (nexttoken.type=NUM)  
        // next token can start 2nd rule for term  
        match(NUM);  
    else error(); // some error function  
}
```

9

Returning the Parse Tree

- We also want to be able to return the parse tree.
- This can be done if we have a *general tree* ADT, which allows trees structures where a node can have any number of children. We also assume that each node can hold a string as its data.
- Our tree ADT will have (at least) the following operations:

```
class tree {  
public:  
    tree();  
    tree* addchild();  
    void adddata(char*);  
    // functions to access nodes..  
private:  
    // private data  
}
```

Addchild creates a new child node and returns a pointer to it. Adddata adds a string to a given node.

11

Starting it Up

The parser requires a global variable called nexttoken to be “ready” when it enters the main parse function, so our main program would contain:

```
tokentype nexttoken;  
  
void main()  
{  
    getnexttoken();  
    term(); // (our start symbol)  
    match(EOF);  
}
```

EOF is an extra token type corresponding to the end of file - match can be used to check that after the parse is finished we're at EOF.

10

- We could create a small tree using statements like:

```
tree* t = new tree;  
tree* child1, child2;  
child1 = t->addchild();  
child2 = t->addchild();  
t->adddata('alison');  
child1->adddata('sophie');  
..
```

- NOTE: It's easiest to deal with *pointers* to trees, hence `t->addchild()`; rather than `t.addchild()`;

12

Returning the Parse Tree

- We can use this to construct a parse tree and return it in arguments to the parse functions.
- Each function (eg. `term()`) will have an argument - a pointer to a tree. When the function has executed this should point to a parse tree for that expression (e.g., the `term`)
- When functions for other rules are called, these in turn will result in part of the tree being constructed; They must be added into the tree of the calling function, e.g.,

```
void term(tree* t)
{
    ...
    tree* child=t->addchild();
    expr(child);
    ...
}
```

13

Summary

- Recursive descent parsing involves writing a (possibly recursive) function for rule in a grammar.
- For each non-terminal on RHS call a corresponding function, and for each terminal “match” that terminal with token, and get next token.
- Use one token looked ahead to decide which of several rules with same LHS to use - simple if statement. Use while loop if symbol on RHS may be repeated.
- Can construct and return tree at same time. Each function will return a subtree, and symbols corresponding to terminal nodes will be added to the tree explicitly.

14