

Compilers

Lecture 9

Murdoch J. Gabbay, Heriot-Watt University, Scotland

Remember:

Forget YACC.

YACC? What's that?

Recursive descent parsing is here!

Recursive descent parsing

To each non-terminal associate a **recogniser function** (typically, it is recursive).

Basically, it's as if a non-terminal **is** a recogniser function which recognises the things that non-terminal can become (via production rules).

To parse, pop a symbol off the input stream into the lookahead variable, then invoke the recogniser function associated with the start symbol.

Match each terminal on the right-hand side of a production against the look-ahead symbol.

For each non-terminal, invoke the associated recogniser function.

Recursive descent parsing

You **succeed** when

- the recognizer function you first invoked (for the start symbol) returns true, and
- the symbol stream is empty.

Unlike table driven parsing, recursive descent does not maintain a stack of grammar symbols.

This information instead is maintained in the depth of recursive calls of recogniser functions, **i.e.** within the run-time environment of the parser.

Recogniser 1

Let's recognise this grammar:

$$\langle A \rangle ::= (\langle A \rangle) \mid \langle B \rangle$$
$$\langle B \rangle ::= a \mid b$$

Requires two recogniser functions A and B (gasp!).

A recognises strings of the form

$$(\langle A \rangle)$$
$$\langle B \rangle$$

B recognises strings of the form

a

b

Recogniser 1: code

```
#include <stdio.h>
#include <stdlib.h>
char ch; /* Holds current symbol */
char next()
{
    char c;
    /* Skip white space */
    do c = getchar(); while (c == ' ');
    printf("%c", c);
    return c;
} /* End next */
int B()
{
    if ((ch == 'a') || (ch == 'b')){
        ch = next();
        return 1;
    }
    else
        return 0;
} /* End B */
```

Recogniser 1: more code

```
int A()
{
    if (ch == '('){
        ch = next();
        if (A()){
            if (ch == ')'){
                ch = next();
                return 1;}
            else
                return 0;}
        else
            return 0;}
    else
        return B();
} /* End A */
void main()
{
    ch = next();
    if (A() && (ch == '\n'))
        printf("SUCCESS\n");
    else
        printf("FAILURE\n");
} /* End main */
```

Notes on recogniser 1

Lexical and syntax analysis are merged.

`next` (via `getchar`) pops a character off standard input.

The top-level recognizer is represented by \bar{A} .

Success corresponds to

- \bar{A} returns 1, and
- next symbol is `\n` (newline)

We use newline to denote 'end-of-string'. A compiler would probably use the end-of-file marker instead.

Recogniser 1 in action

```
a
a
SUCCESS
(b)
(b)
SUCCESS
(((a)))
(((a)))
SUCCESS
((b))
((b))
FAILURE
(a, b)
(a, FAILURE
(a) b
(a)bFAILURE
()
()FAILURE
```

Recogniser 2

Here's a grammar:

$$\langle A \rangle ::= (\langle A \rangle) \mid \langle B \rangle \mid \langle C \rangle$$
$$\langle B \rangle ::= x \langle B \rangle \mid y$$
$$\langle C \rangle ::= x \langle C \rangle \mid z$$

Let's left-factor it because it's ambiguous otherwise:

$$\langle A \rangle ::= (\langle A \rangle) \mid x \langle D \rangle \mid \langle E \rangle$$
$$\langle D \rangle ::= x \langle D \rangle \mid \langle E \rangle$$
$$\langle E \rangle ::= y \mid z$$

Code!

```
int E()
{
    if ((ch == 'y') || (ch == 'z')){
        ch = next();
        return 1;}
    else
        return 0;
} /* End E */
int D()
{
    if (ch == 'x'){
        ch = next();
        return D();}
    else
        return E();
} /* End D */
```

More code!!

```
int A()
{
    if (ch == '('){
        ch = next();
        if (A()){
            if (ch == ')'){
                ch = next();
                return 1;}
            else
                return 0;}
        else
            return 0;}
    else
        if (ch == 'x'){
            ch = next();
            return D();}
        else
            return E();
} /* End A */
```

Yet more code!!!

```
void main()  
{  
    ch = next();  
    if (A() && (ch == '\n'))  
        printf("SUCCESS\n");  
    else  
        printf("FAILURE\n");  
} /* End main */
```

Code in action!!!!

```
Y
Y
SUCCESS
Z
Z
SUCCESS
X X XY
XXXY
SUCCESS
XX Z
XXZ
SUCCESS
(x x z)
(xxz)
SUCCESS
(x x y z)
(xxyz)
FAILURE
```

Constructing LEAD sets

The construction of $\text{LEAD}(\alpha)$ is as follows:

If α is of the form: $b\beta$

then b is in $\text{LEAD}(\alpha)$.

If α is of the form: $B\beta$

then add $\text{LEAD}(B)$ to $\text{LEAD}(\alpha)$.

If α is of the form: $BX\beta$

and $B \Rightarrow \varepsilon$

then add $\text{LEAD}(X)$ to $\text{LEAD}(\alpha)$.

If α is of the form: B

and $B \rightarrow \beta$

then add $\text{LEAD}(\beta)$ to $\text{lead}(\alpha)$.

Predicting left-factoring: general case

$$\langle A \rangle ::= \alpha_1 \mid \dots \mid \alpha_n$$

left-factoring required if the LEAD sets for α_i and α_j are not disjoint, i.e.

$$\text{LEAD}(\alpha_i) \cap \text{LEAD}(\alpha_j) \neq \{\}$$

for some $1 \leq i \leq n, 1 \leq j \leq n, i \neq j$.

Specific case

Consider again recogniser 2:

$$\text{LEAD}(\langle\langle A \rangle\rangle) = \{(\}$$

$$\text{LEAD}(\langle B \rangle) = \{x, y\}$$

$$\text{LEAD}(\langle C \rangle) = \{x, z\}$$

Applying the left-factoring check gives:

$$\text{LEAD}(\langle B \rangle) \cap \text{LEAD}(\langle C \rangle)$$

$$= \{x, y\} \cap \{x, z\}$$

$$= \{x\}$$

So left-factoring is required.

Recogniser 3

Our grammar:

$$\langle A \rangle ::= (\langle A \rangle) \mid \langle B \rangle y$$
$$\langle B \rangle ::= x \mid \langle B \rangle y$$

Our grammar transformed to eliminate left-recursion:

$$\langle A \rangle ::= (\langle A \rangle) \mid \langle B \rangle y$$
$$\langle B \rangle ::= x \langle C \rangle$$
$$\langle C \rangle ::= y \langle C \rangle \mid \varepsilon$$

Why eliminate left-recursion?

A grammar is not left recursive when for each nonterminal any leftmost series of expansions terminates.

We repeatedly expand the first nonterminal and stop when the first symbol is a terminal, or the list produced is empty (ϵ).

If the grammar is left recursive, our recursive descent parser could loop forever.

Code!

```
int C()
{
    if (ch == 'y'){
        ch = next();
        return C();}
    else
        return 1; /* null case */
} /* End C */
int B()
{
    if (ch == 'x'){
        ch = next();
        return C();}
    else
        return 0;
} /* End B */
```

More code!!

```
int A()  
{  
    if (ch == '('){  
        ch = next();  
        if (A()){  
            if (ch == ')'){  
                ch = next();  
                return 1;}  
            else  
                return 0;}  
        else  
            return 0;}  
    else  
        if (B()){  
            return (ch == 'y');}  
        else  
            return 0;  
} /* End A */
```

Yet more code!!!

```
void main()  
{  
    ch = next();  
    if (A() && (ch == '\n'))  
        printf("SUCCESS\n");  
    else  
        printf("FAILURE\n");  
  
} /* End main */
```

Recogniser 3 in action!!!!

```
xyy
xyy
FAILURE
```

Parser	Next ch	Rest
1. call A()	x	y y \n
2. call B()	x	y y \n
if(ch == 'x')...	x	y y \n
3. call C()	y	y \n
if(ch == 'y')...	y	y \n
4. call C()	y	\n
if(ch == 'y')...	y	\n
5. call C()	\n	
5. 1	\n	
4. 1	\n	
3. 1	\n	
2. 1	\n	
1. return(ch == 'y')	\n	
FAILURE		

Recogniser 3, in action

Hang on, that's wrong.

xyy should be produced by our grammar:

$$\langle A \rangle \Rightarrow \langle B \rangle y \Rightarrow \langle B \rangle yy \Rightarrow xyy$$

Looks like our recogniser isn't recognising.

Recogniser 3, failure analysis

The problem is with the context in which $\langle C \rangle$ is defined, *i.e.*

$$\langle A \rangle ::= (\langle A \rangle) \mid \langle B \rangle y$$
$$\langle B \rangle ::= x \langle C \rangle$$
$$\langle C \rangle ::= y \langle C \rangle \mid \varepsilon$$

An instance of $\langle C \rangle$ is either empty (ε) or takes the form “y ...”.

Furthermore from the definitions of $\langle B \rangle$ and $\langle A \rangle$ an instance of $\langle C \rangle$ **must be** followed by a y.

Recogniser 3, failure analysis

A parser which supports **one-symbol-lookahead without backtracking** cannot know whether a y should be matched with the definition for $\langle A \rangle$, or $\langle C \rangle$, **i.e.**

$$\langle A \rangle ::= \dots \langle B \rangle y$$

$$\langle C \rangle ::= \dots y \langle C \rangle$$

This is the **context clash** problem.

FOLLOW sets

FOLLOW(A) is defined by:

If A is the start symbol then put EOS (end-of-string) in FOLLOW(A).

If there is a rule

$$B ::= \alpha A \beta$$

then add LEAD(β) to FOLLOW(A).

If there is a rule

$$B ::= \alpha A, \quad \text{or} \quad B ::= \alpha A \beta$$

where $\beta \Rightarrow \varepsilon$, then add FOLLOW(B) to FOLLOW(A).

Predicting context clashes: general case

If A has a null production rule, i.e.

$$A ::= \varepsilon,$$

and $\text{LEAD}(A) \cap \text{FOLLOW}(A) \neq \{\}$ then recursive descent is **not** possible.

Context clashes: specific case

Consider $\langle C \rangle$ in recognizer 3:

$$\text{LEAD}(\langle C \rangle) = \{y\}$$

$$\text{FOLLOW}(\langle C \rangle) = \{y\}$$

so:

$$\text{LEAD}(\langle C \rangle) \cap \text{FOLLOW}(\langle C \rangle) = \{y\}$$

The grammar for recognizer 3 **can not** be parsed by recursive descent.

Recogniser 4

The grammar:

$$\langle A \rangle ::= (\langle A \rangle) \mid \langle B \rangle z$$
$$\langle B \rangle ::= x \mid \langle B \rangle y$$

The grammar transformed to eliminate left-recursion:

$$\langle A \rangle ::= (\langle A \rangle) \mid \langle B \rangle z$$
$$\langle B \rangle ::= x \langle C \rangle$$
$$\langle C \rangle ::= y \langle C \rangle \mid \varepsilon$$

($\text{LEAD}(\langle C \rangle) \cap \text{FOLLOW}(\langle C \rangle) = \{ \}$; recursive descent may be applicable.)

Code!

```
int C()
{
    if (ch == 'y'){
        ch = next();
        return C();}
    else
        if (ch == 'z'){ /* FOLLOW(C) == {'z'}
            then return 1; /* null case */
            else
                return 0;
        }
} /* End C */
int B()
{
    if (ch == 'x'){
        ch = next();
        return C();}
    else
        return 0;
} /* End B */
```

More code!!

```
int A()
{
    if (ch == '('){
        ch = next();
        if (A()){
            if (ch == ')'){
                ch = next();
                return 1;
            }
            else
                return 0;
        }
        else
            return 0;
    }
    else
        if (B()){
            if (ch == 'z'){
                ch = next();
                return 1;
            }
            else return 0;
        }
    else
        return 0;
} /* End A */
```

Yet more code!!!

```
void main()
{
    ch = next();
    if (A() && (ch == '\n'))
        printf("SUCCESS\n");
    else
        printf("FAILURE\n");
} /* End main */
```

Recogniser 4 in action

```
((xyz))  
((xyz))  
SUCCESS  
xyz  
xyz  
SUCCESS  
xz  
xz  
SUCCESS  
xy  
xy  
FAILURE
```

Propositional language 1

Our grammar:

$$\begin{aligned} \langle \text{Prop} \rangle & ::= \langle \text{Prop} \rangle \& \langle \text{Term} \rangle \mid \\ & \quad \langle \text{Prop} \rangle \vee \langle \text{Term} \rangle \mid \\ & \quad \langle \text{Term} \rangle \\ \langle \text{Term} \rangle & ::= \sim \langle \text{Prop} \rangle \mid \\ & \quad (\langle \text{Prop} \rangle) \mid \\ & \quad \text{t} \mid \text{f} \end{aligned}$$

Propositional language 2

Transformed grammar:

$$\begin{aligned}\langle \text{Prop} \rangle & ::= \langle \text{Term} \rangle \langle \text{PropP} \rangle \\ \langle \text{PropP} \rangle & ::= \& \langle \text{Term} \rangle \langle \text{PropP} \rangle \mid \\ & \quad \vee \langle \text{Term} \rangle \langle \text{PropP} \rangle \mid \\ & \quad \text{null} \\ \langle \text{Term} \rangle & ::= \sim \langle \text{Prop} \rangle \mid \\ & \quad (\langle \text{Prop} \rangle) \mid \\ & \quad \text{t} \mid \text{f}\end{aligned}$$

Prop 1 code

```
int Term()
{
    if (ch == '~') {
        ch = next();
        return Prop();
    }
    else
        if (ch == '(') {
            ch = next();
            if (Prop())
                if (ch == ')') {
                    ch = next();
                    return 1;
                }
            else return 0;
        }
        else
            return 0;
    else
        if ((ch == 't') || (ch == 'f')) {
            ch = next();
            return 1;
        }
        else return 0;
}
```

Prop 1 code

```
int PropP()
{
    if ((ch == '&' ) || (ch == 'v' )){
        ch = next();
        if (Term())
            return PropP();
        else
            return 0;}
    else
        /* follow(PropP) == {')', '\n'} */
        if ((ch == ')') || (ch == '\n'))
            return 1; /* null case */
        else
            return 0;
}
int Prop()
{
    if (Term()) return PropP();
    else
        return(0);
}
```

Prop 1 code

```
void main()  
{ ch = next();  
  if (Prop() && (ch == '\n'))  
      printf("SUCCESS\n");  
else  
    printf("FAILURE\n");}
```

Propositional language 2

Grammar:

$$\begin{aligned} \langle \text{Prop} \rangle & ::= \langle \text{Prop} \rangle \text{ and } \langle \text{Term} \rangle \mid \\ & \quad \langle \text{Prop} \rangle \text{ or } \langle \text{Term} \rangle \mid \\ & \quad \langle \text{Term} \rangle \\ \langle \text{Term} \rangle & ::= \text{not } \langle \text{Prop} \rangle \mid \\ & \quad (\langle \text{Prop} \rangle) \mid \\ & \quad \text{true} \mid \text{false} \end{aligned}$$

Propositional language 2

Transformed grammar:

$$\begin{aligned}\langle \text{Prop} \rangle & ::= \langle \text{Term} \rangle \langle \text{PropP} \rangle \\ \langle \text{PropP} \rangle & ::= \text{and } \langle \text{Term} \rangle \langle \text{PropP} \rangle \mid \\ & \quad \text{or } \langle \text{Term} \rangle \langle \text{PropP} \rangle \mid \\ & \quad \text{null} \\ \langle \text{Term} \rangle & ::= \text{not } \langle \text{Prop} \rangle \mid \\ & \quad (\langle \text{Prop} \rangle) \mid \\ & \quad \text{true} \mid \text{false}\end{aligned}$$

Prop 2 code (LEX definitions)

```
%{  
#include <stdio.h>  
#include <stdlib.h>  
/* symbols */  
#define AND      256  
#define OR       257  
#define NOT      258  
#define OPEN     259  
#define CLOSE    260  
#define TRUE     261  
#define FALSE    262  
#define EOS      263 /* End-Of-String */  
%}  
%%
```

Prop 2 code (LEX rules)

```
[\t ]+ /* ignore white space */ ;
"and" { return AND; }
"or" { return OR; }
"not" { return NOT; }
" (" { return OPEN; }
")" { return CLOSE; }
"true" { return TRUE; }
>false" { return FALSE; }
\n { return EOS; }
%%
```

Prop 2 code (LEX subroutines)

```
int tok; /* Next symbol */
int Prop();
int PropP();
int Term();
int Term()
{
    if (tok == NOT){
        tok = yylex();
        return Prop();}
    else
        if (tok == OPEN){
            tok = yylex();
            if (Prop())
                if (tok == CLOSE){
                    tok = yylex();
                    return 1;}
                else return 0;
            else
                return 0;}
        else
            if ((tok == TRUE) || (tok == FALSE)){
                tok = yylex();
                return 1;}
            else return 0;        }
```

...and a few more subroutines

```
int PropP()
{
    if ((tok == AND) || (tok == OR)) {
        tok = yylex();
        if (Term())
            return PropP();
        else
            return 0;
    }
    else
        /* follow(PropP) == {CLOSE,EOS} */
        if ((tok == CLOSE) || (tok == EOS))
            return 1; /* null case */
        else
            return 0;
}
int Prop()
{
    if (Term()) return PropP();
    else
        return(0);
}
```

...and a few more subroutines

```
void main()
{ tok = yylex();
  if (Prop() && (tok == EOS))
      printf("SUCCESS\n");
else
    printf("FAILURE\n");}
```

Prop 1 and 2 in action

- Prop 1:

```
(t & ~f) v ~t  
(t&~f)v~t  
SUCCESS  
(t & ~f) v & f  
(t&~f)v&FAILURE
```

- Prop 2:

```
(true and not false) or not true  
SUCCESS  
(true and not false) or and false  
FAILURE
```

Summary

The $LL(k)$ and $LR(k)$ parser classification.

Recursive descent parsing $LL(1)$.

Exploiting LEX in building recursive descent parsers.

Examples available on my homepage:

<http://www.gabbay.org.uk>

`recognizer1.c` `recognizer2.c`

`recognizer3.c` `recognizer4.c`

`prop1.c` `prop2.l`