

Department of Computer Science
Australian National University

COMP3610

Principles of Programming Languages

Bottom-Up Parsing, and Scanning

Clem Baker-Finch
September 26, 2007

Contents

1	Bison and Flex	1
1.1	Bison Overview	2
1.2	Flex Overview	6
1.3	Using Bison and Flex Together	9
1.4	Left and Right Recursive Productions	12
1.5	Conflicts in Bison	14
2	The Theory of Bottom-Up Parsing	17
2.1	Push-down Automata	17
2.2	Extended Push-Down Automata	19
2.3	Deterministic Bottom-Up Parsing	22
2.4	Constructing <i>SLR</i> tables	28
3	Lexical Analysis and Finite State Automata	36
3.1	Languages and Automata	36
3.2	From Regular Expressions to Finite Automata	38
3.3	Making Them Deterministic	39

Chapter 1

Bison and Flex

Bison is an *LALR(1) parser generator*, so it is based on an *EPDA* with 1 symbol lookahead. It is the GNU version of *yacc* (yet another compiler-compiler). Bison takes as input a context-free grammar defining a language and produces a parser for that language.

A lexical analyser must be provided, either hand-coded (see example `calc1.y` via the unit web page) or built by a lexer generator such as Flex. Flex is the GNU version of `lex`. (Fast lex.) **Semantic actions** can be attached to each production and grammar symbols can have associated **semantic attributes**. The semantic actions consist of C code and they are executed when the right-hand side of a production has been recognised, i.e. at **reduce** actions.

Suppose file `xxx.y` is a Bison specification, consisting of a grammar with semantic actions and a hand-coded lexer. Bison produces a C program which is a parser and processor (depending on the semantic actions) for the specified grammar.

The process is as follows:

$$\left. \begin{array}{l} \text{grammar} \\ \text{semantic actions} \\ \text{lexer} \end{array} \right\} \Rightarrow \text{Bison} \Rightarrow \text{xxx.tab.c}$$

`xxx.tab.c` is a C program which can be compiled as usual to produce an executable parser.

$$\text{xxx.tab.c} \Rightarrow \text{gcc} \Rightarrow \text{executable.}$$

Flex is a *scanner* (or *lexer*) generator, based on finite state automata. The generated lexer is a C module. (Flex itself is built using Bison.) The input to Flex is a collection of specifications of lexical tokens, with semantic actions, again written in C.

The process is:

$$\left. \begin{array}{l} \text{regular expressions} \\ \text{semantic actions} \end{array} \right\} \Rightarrow \text{Flex} \Rightarrow \text{lex.yy.c}$$

Commonly, Flex and Bison are used together. To link them (common tokens, attributes etc.) Bison produces a header file `xxx.tab.h` which we **#include** in the Flex specification. The linking is completed by `gcc` to produce a single executable.

An example Flex specification follows. Note that the order of the regular expressions is significant: `[a-zA-Z]+` overlaps the preceding pattern.

```

%{
/* Simple flex specification.
 * Distinguishes some verbs and non-verbs.
 */
%}
%%

[\\t ]+ ; // ignore white space.

is |
am |
are |
were |
was |
be |
being |
been |
has |
have |
had |
go      {printf("%s: is a verb\\n", yytext);}

[a-zA-Z]+ {printf("%s: is not a verb\\n", yytext);}

. | \\n   {ECHO; /* normal default anyway */}

%%
main()
{ yylex(); }

```

1.1 Bison Overview

For more information, see the Bison tutorial/manual — on-line at the COMP3610 web site.

The basic job of Bison is to produce, from a set of grammar rules, a C function called `yyparse()` corresponding to those grammar rules. (*Warning:* the code generated by Bison has numerous identifiers starting with “yy” so avoid that convention with your own identifiers.) The embedded semantic actions (C code) are run as the productions are applied (i.e. at reduce actions).

Layout of a Bison specification:

```

%{
. . . C declarations . . .
%}
. . . Bison declarations . . .
%%
. . . Grammar rules . . .

```

```
%%
    . . . Additional C code
```

C declarations: This section contains declarations of macros, functions, variables etc. that are used in the semantic actions. Any `#include` directives go here, too. In the C parser produced by Bison, this stuff will appear before the definition of `yyparse`. No checking or processing of any C code is done by Bison.

Bison declarations: This section contains definitions of terminal symbols, semantic attributes, precedence, associativity, etc.

Grammar rules: This section contains a set of context-free productions with associated semantic actions.

Additional C code: Any material in this section is copied directly to `xxx.tab.c`, following the definition of `yyparse`. Generally it includes:

- a definition of `main` (which will include a call to `yyparse`)
- a definition of an error reporting function called `yyerror`
- a definition of a lexer called `yylex`, if it is being hand-coded rather than being produced by Flex.

An Example Bison Specification

```
/* Reverse polish notation calculator.  calc1.y */
/* Hand-coded lexer rather than flex-generated. */

%{
#define YYSTYPE double
#include <math.h>
%}

%token NUM

%%

input
: /* empty */
| input line
;
line
: '\n'
| exp '\n' { printf ("\t= %.2f\n", $1); }
;
exp
: NUM          { $$ = $1;          }
| exp exp '+'  { $$ = $1 + $2;     }
| exp exp '-'  { $$ = $1 - $2;     }
```

```

    | exp exp '*'      { $$ = $1 * $2;      }
    | exp exp '/'      { $$ = $1 / $2;      }
    | exp exp '^'      { $$ = pow ($1, $2); }
    | exp 'n'          { $$ = -$1;          } /* Unary negation */
;
%%

/* Lexical analyzer returns a double floating point
   number on the stack and the token NUM, or the
   character read if not a number. Skip blanks
   and tabs, return 0 for EOF. */

#include <ctype.h>

yylex ()
{
    int c;

    /* skip white space */
    while ((c = getchar ()) == ' ' || c == '\t')
        ;
    /* process numbers */
    if (c == '.' || isdigit (c))
    {
        ungetc (c, stdin);
        scanf ("%lf", &yylval);
        return NUM;
    }
    /* return end-of-file */
    if (c == EOF)
        return 0;
    /* return single chars */
    return c;
}

main ()
{
    yyparse ();
}

#include <stdio.h>

yyerror (char *s) /* Called by yyparse on error */
{
    printf ("\terror: %s\n", s);
}

```

Grammar Specifications in Bison

Productions have the syntax:

```
non-terminal : rhs | rhs | . . . | rhs;
```

Each right-hand side is a sequence of 0 or more terminal or non-terminal symbols.¹ The terminal symbols can be literal *characters* in quotes (e.g. '+') or **tokens** returned by the lexer.

Semantic actions can appear in the right-hand sides (preferably at the end) and are a sequence of C statements enclosed in braces, { . . . }.

Of course

```
nt : rhs1 | rhs2 ;
```

is the same as

```
nt : rhs1 ;
nt : rhs2 ;
```

In Bison, non-terminals are often called “types”. The common layout style is demonstrated in `calc1.y` above.

Semantic Attributes

The parser produced by Bison assumes every grammar symbol to have an attribute of type `YYSTYPE`. By default, `YYSTYPE` is `int` but it can be changed, as in the postfix calculator `calc1.y`, by defining `YYSTYPE`, e.g.:

```
#define YYSTYPE double
```

in the C declarations part of the Bison specification. The code in a semantic action can refer to the semantic attributes using the `$$`, `$1`, `$2`, . . . notation. In a production:

```
nt : sym1 sym2 sym3 . . .
```

`$$` is the semantic attribute of the left-hand side, `nt`,

`$1` is the semantic attribute of `sym1`,

`$2` is the semantic attribute of `sym2`, etc.

Most often, we don't want the same attribute type for all grammar symbols. For example, numeric literals may be `int`, `double` etc., string constants may be `char *`, identifiers may be symbol table pointers etc. In that case we can put a `%union` declaration in the Bison declaration section to specify all the types of attributes. For example, (in `calc3.y`):

```
%union {
    double dval; int vblno; }
```

Then, using `%token` and `%type` declarations, each variant can be associated with the appropriate grammar symbol. For example (`calc3.y`):

¹Convention: use a comment `/* empty */` to indicate an empty right-hand side.

```

%token <vblno> NAME
%token <dval> NUM

%type <dval> expression

```

If the same symbol requires several attributes, its variant in the `%union` should be a *structure*. The components can be extracted with the usual dot-notation, e.g. `$3.label`.

Bison Declarations

The main purpose of the Bison declaration section is to declare :

- terminal symbols of the grammar
- semantic attributes
- start symbol (default is the first non-terminal in the grammar)

The standard declaration of a terminal NUM is as a token:

```
%token NUM
```

Bison converts this to a `#define` directive in the parser, e.g. `#define NUM 258`. In the unusual event that all symbols have the same semantic attributes, we can give an appropriate definition of `YYSTYPE`, as in `calc1.y` and `calc2.y`. More often this is not the case, so we use a `%union` declaration (as on a previous page). The variant associated with each terminal symbol can be specified in the `%token` specification.

```

%token <vblno> NAME
%token <dval> NUM

```

indicates that NUM has a `double` attribute and NAME has an `int` attribute. To specify which variant is the attribute of a non-terminal symbol, use a `%type` declaration:

```
%type <dval> expression
```

The Bison declaration section can specify associativity and precedence information, too:

```

%nonassoc '='
%left '+' '-'
%left '*' '/'
%right '^'

```

says that `+`, `-`, `*`, `/` are left associative, `^` is right associative and `=` is non-associative. Their relative precedence is implied by the order of these declarations, lowest to highest. (i.e. `=` lowest, `^` highest.)

1.2 Flex Overview

For more information, see the Flex tutorial/manual — on-line at the COMP3610 web site.

The basic job of Flex is to take a sequence of regular expressions and produce a C function called `yylex`, corresponding to those regular expressions and their associated semantic actions.

Layout of a Flex specification

```
%{
    . . . C declarations . . .
}%
    . . . Flex definitions . . .
%%
    . . . Flex rules . . .
%%
    . . . Additional C code
```

C declarations

This section corresponds to the Bison C-declarations section and serves the same purpose. In the lexer (C program) produced by Flex, this stuff appears before the definition of `yylex`. In particular, for linking with parsers produced by Bison, the `#include "xxx.tab.h"` goes here.

Flex Definitions

As in regular definitions, this section allows us to name some regular expressions which may be used later. For example:

```
DIGIT  [0-9]
LETTER [A-Za-z]
```

When used in regular expressions in the Flex rules, “`{DIGIT}`” and “`{LETTER}`” will be expanded according to these definitions.

Flex Rules

This is the main section, consisting of a sequence of regular expressions and associated semantic actions.

Flex Regular Expressions

(Incomplete — see the manual for full details.)

Single characters:

- `x` matches `x`;
- `\t` matches a tab — similarly for other C character-escape sequences;
- `.` matches any character *except newline*;
- `<<EOF>>` matches end-of-file

Let `r` and `s` be regular expressions:

- `r*` matches 0 or more `r`-patterns;
- `r+` matches 1 or more `r`-patterns;
- `r?` matches 0 or 1 `r`-pattern;
- `rs` matches an `r`-pattern followed by an `s`-pattern;
- `r|s` matches either an `r`-pattern or an `s`-pattern;
- `{name}` matches the regular expression defined with that name.
- `^r` matches an `r`-pattern at the beginning of a line;
- `r$` matches an `r`-pattern at the end of a line;

The unary postfix operators (`*`, `+`, `?`) are of higher precedence than concatenation `rs`, which is of higher precedence than alternation `r|s`. Parentheses can be used to over-ride precedence.

Example 1.2.1

`red|hot*` is the same as: `(red)|(ho(t*))`

Classes

It would be tedious to write out common regular expressions like `a|b|...z|A|B|...Z` in full again and again, so Flex provides character classes:

- `[xyz]` matches any of the characters 'x', 'y' or 'z';
- `[a-z]` matches any lower-case letter;

A common example is `[\t\n]` which matches space, tab, newline (i.e. white-space).

Flex also has *negated* classes:

- `[^0-9]` matches any characters *except* a digit.

Example 1.2.2

Design regular expressions to match C single-line and multiple-line comments. (This is not trivial. See the Flex manual for an alternative using more advanced features of Flex.)

Note 1.2.3

The ordering of the sequence of regular expressions in the Flex rules section is significant. For example:

```
begin      { return BEGINSYM ; }
[a-zA-Z]+  { return IDENTIFIER ; }
```

behaves appropriately but

```
[a-zA-Z]+  { return IDENTIFIER ; }
begin      { return BEGINSYM ; }
```

will never return `BEGINSYM` because “`begin`” will match the pattern `[a-zA-Z]+` first. On the other hand, Flex matches as much as possible, so for example, in either case above, “`beginning`” will be recognised as an `IDENTIFIER`.

1.3 Using Bison and Flex Together

Recall that Bison specifications include declarations of tokens (`%token ...`) and semantic attributes (`%union ...`). How do we arrange that the lexical analyser `yylex` generated by Flex, returns such tokens and initialises such semantic attributes?

First, running Bison with the `-d` command-line option creates a *header* file. That is, the command `bison -d xxx.y` produces two files, `xxx.tab.c` and `xxx.tab.h`. That header file can be `#include`-ed in the Flex specification (C declarations part).

The header file contains:

- a definition of `YYSTYPE` as the union type of semantic attributes;
- definitions of all the tokens as numeric constants;
- declaration of an `extern` variable `yylval` of type `YYSTYPE`.

consequently the tokens declared in the Bison specification can be referred to in the Flex specification. In general we use statements like:

```
return NAME ;
```

in the Flex semantic actions. This causes the function `yylex` to return those tokens (to `yyparse` which calls `yylex`).

Semantic attributes are passed back through the variable `yylval`. That is, giving `yylval` some value prior to the `return token` statement makes that the value of the semantic attribute of the returned token. If we have recognised a `NUM` for example, how do we get its *value* to return in `yylval`? The lexer generated by Flex sets the variable `char * yytext` to contain the string just recognised. Hence we can process `yytext` however we wish to calculate the attribute `yylval`. For numeric literals we might use `yylval = atof (yytext) ;`

Example part 1 (Flex)

```
/* file name : calc3.1 */

/* lexer for calculator grammar with variables and real values. */
/* See bison specification calc3.y */
/* derived from "lex and yacc" ch3-03.1 (page 65) */

%{
#include "calc3.tab.h"
%}

DIGIT [0-9]

%%

{DIGIT}+("."{DIGIT}+)? {
```

```

        yylval.dval = atof(yytext); return NUM; }

[ \t]      /* ignore whitespace */

[a-z]      yylval.vblno = yytext[0] - 'a'; return NAME;

<<EOF>>    return 0; /* signal end of dialogue */

\n         return yytext[0];
.          return yytext[0];

%%

```

Example part 2 (Bison)

```

/* file name : calc3.y */

/* Calculator grammar with variables and float values. */
/* Variables are single letters and symbol table is a */
/* simple array of double, indexed by variable names. */
/* Associativity and precedence declarations used to */
/* disambiguate the grammar. */
/* Derived from "lex and yacc" ch3-03.y (page 64) */

%{

#include <stdio.h>

double vbtable[26];

%}

%union {
    double dval;
    int vblno;
}

%token <vblno> NAME
%token <dval> NUM
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%type <dval> expr

%%

input

```

```

: /* empty */
| input line
;

line
: '\n'
| NAME '=' expr '\n' { vbltable[$1] = $3; }
| expr '\n' { printf ("\t= %.2f\n", $1); }
;

expr
: expr '+' expr { $$ = $1 + $3; }
| expr '-' expr { $$ = $1 - $3; }
| expr '*' expr { $$ = $1 * $3; }
| expr '/' expr { if ($3 == 0.0)
                  yyerror("divide by zero");
                  else
                    $$ = $1 / $3; }
| '-' expr %prec UMINUS { $$ = - $2; }
| '(' expr ')' { $$ = $2; }
| NUM { $$ = $1; }
| NAME { $$ = vbltable[$1]; }
;

%%

main ()
{
    yyparse ();
}

yyerror (char *s) /* Called by yyparse on error */
{
    printf ("\terror: %s\n", s);
}

```

Header file produced by Bison from calc3.y

```

typedef union {
    double dval;
    int vblno;
} YYSTYPE;
#define NAME 258
#define NUM 259
#define UMINUS 260

extern YYSTYPE yylval;

```

1.4 Left and Right Recursive Productions

Suppose we have a grammar including a list of numbers separated by commas, and as the semantic action we wish to print them out *in order*. The grammar in `list.y` below achieves that objective.

Note that the main productions:

```
list  : list ',' NUM
      | NUM
```

are *left-recursive*.

```
/* file name : list1.y */
/* left recursive production demonstration */

%{
#define YYSTYPE double
%}

%token NUM

%%

input : list '\n'
      | input list
      ;
list  : list ',' NUM {printf ("\t= %.2f\n", $3);}
      | NUM         {printf ("\t= %.2f\n", $1);}
      ;

%%
main ()
{
  yyparse ();
}

yyerror (char *s) /* Called by yyparse on error */
{
  printf ("\terror: %s\n", s);
}
```

The grammar in `list2.y` defines the same language but is *right-recursive*:

```
list  : NUM ',' list
      | NUM
```

This results in the numbers being printed out in *reverse order*. (Why?)

```
/* file name : list2.y */
```

```

/* right recursive production demonstration */

%{
#define YYSTYPE double
%}

%token NUM

%%

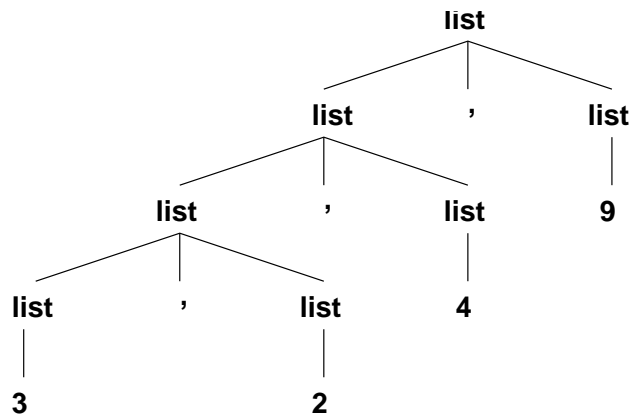
input : list '\n'
      | input list
      ;
list  : NUM ',' list {printf ("\t= %.2f\n", $1);}
      | NUM          {printf ("\t= %.2f\n", $1);}
      ;

%%
main ()
{
    yyparse ();
}

yyerror (char *s) /* Called by yyparse on error */
{
    printf ("\terror: %s\n", s);
}

```

Association and precedence declarations: Alternatively, we could declare `','` to be left associative as in `list3.y`. This forces the string “3, 2, 4, 9” to be parsed as:



Exercise 1.4.1

Draw parse trees for sample string, using `list1.y`, `list2.y` and `list3.y` and attach the semantic actions as extra leaves. Note the order of visits in a depth first traversal.

Exercise 1.4.2

Change `list3.y` so that `','` is right associative. What is the result?

```

/* file name : list3.y */
/* left associative production demonstration */

%{
#define YYSTYPE double
%}

%token NUM
%left ','

%%

input : list '\n'
      | input list
      ;
list  : list ',' list
      | NUM      {printf ("\t= %.2f\n", $1);}
      ;

%%
main ()
{
    yyparse ();
}

yyerror (char *s) /* Called by yyparse on error */
{
    printf ("\terror: %s\n", s);
}

```

1.5 Conflicts in Bison

It is quite common that the grammar given to Bison is not *LALR(1)*. Bison reports this problem as either

- a *shift/reduce* conflict
- a *reduce/reduce* conflict

which means that it cannot construct a deterministic parser for that grammar. The grammar is probably (but not necessarily) ambiguous. (*Detailed information about the conflicts can be gathered using Bison's `verbose` option.*)

We will look at this in more detail in the following section of the notes, but here is an overview. *LR* parsers operate on **states** which are collections of coherent **LR-items**, which are productions

with a marker in the right-hand side to show where the parse is up to, together with a set of terminal **follower** symbols (i.e. they may follow the production in this context). For example:

$$E \rightarrow E + \bullet T \quad [), +]$$

indicates that we have parsed “ $E+$ ” and are about to parse a T . Bison computes all of these states, which then become the stack symbols in the *EPDA* model. If the top-of-stack state contains an item like:

$$A \rightarrow \dots \bullet a \dots \quad [\dots]$$

and the next input token is a then the parsing action is a **shift**, just as in the *EPDA*.

A state containing an item like:

$$A \rightarrow \dots a \bullet \dots \quad [\dots]$$

is pushed onto the stack, rather than simply pushing the token a .

If a state containing an item like:

$$B \rightarrow \dots \bullet \quad [\dots]$$

(i.e. the position is at the end of the production) and the next symbol is in the follower set, the parser action is to **reduce** using that production, again as in the *EPDA* model. The effect is to remove states from the stack corresponding the sequence of positions of the production. This must take it back to a state containing an item like:

$$C \rightarrow \dots \bullet B \dots \quad [\dots]$$

Since we have now recognised a B -form, we push a state containing:

$$C \rightarrow \dots B \bullet \dots \quad [\dots]$$

If Bison constructs a state containing *both* items:

$$A \rightarrow \dots \bullet a \dots \quad [\dots] \quad \text{and} \quad B \rightarrow \dots \bullet \quad [\dots a \dots]$$

then with this state on top of the stack and an a as next token, the parser cannot know whether to shift (as suggested by the A -production) or reduce (as suggested by the B -production). This is a **shift/reduce conflict**.

If Bison constructs a state containing *both* items:

$$B \rightarrow \dots \bullet \quad [\dots a \dots] \quad \text{and} \quad C \rightarrow \dots \bullet \quad [\dots a \dots]$$

then with this state on top of the stack and an a as next token, the parser cannot know whether to reduce by the A -production or reduce by the B -production. This is a **reduce/reduce conflict**.

Example 1.5.1

The *dangling-else* problem is a standard example of a shift/reduce conflict.

```
stmt → if bexp then stmt
      | if bexp then stmt else stmt
```

leads to a state containing *both* these items:

$$\begin{aligned} stmt &\rightarrow \mathbf{if} \ bexp \ \mathbf{then} \ stmt \ \bullet \ [\dots \mathbf{else} \ \dots] \\ stmt &\rightarrow \mathbf{if} \ bexp \ \mathbf{then} \ stmt \ \bullet \mathbf{else} \ stmt \ [\dots] \end{aligned}$$

Shift/reduce conflicts should be resolved by modifying the grammar. In fact, Bison has a default resolution of all shift/reduce conflicts: it always chooses to shift. In most cases that may be what we want, but you should be very careful that it really is the appropriate choice.

The following grammar fragment demonstrates a *reduce/reduce* conflict.

$$\begin{aligned} expression &\rightarrow constant \mid variable \mid \dots \\ variable &\rightarrow \mathbf{ident} \mid \dots \\ constant &\rightarrow \mathbf{ident} \mid \mathbf{num} \mid \dots \end{aligned}$$

While parsing an *expression*, the parser cannot tell whether an **ident** is a *variable* or a *constant*. Bison will construct a state containing the items:

$$\begin{aligned} variable &\rightarrow \mathbf{ident} \bullet \ [\dots] \\ constant &\rightarrow \mathbf{ident} \bullet \ [\dots] \end{aligned}$$

with overlapping follower sets (because *variable* and *constant* both appear as complete expressions). In this case, the grammar is intended to distinguish between two different uses of an **ident**. This is impossible – it depends on its *context*. The general solution is to not attempt to make such distinctions in the grammar. In this case we might write:

$$expression \rightarrow \mathbf{ident} \mid \mathbf{num} \mid \dots$$

The distinction between constant identifiers and variable identifiers is a static semantic matter and must be handled by the semantic actions. The obvious solution is to distinguish between constant identifiers and variable identifiers from their *declarations* and keep that information in a *symbol table*.

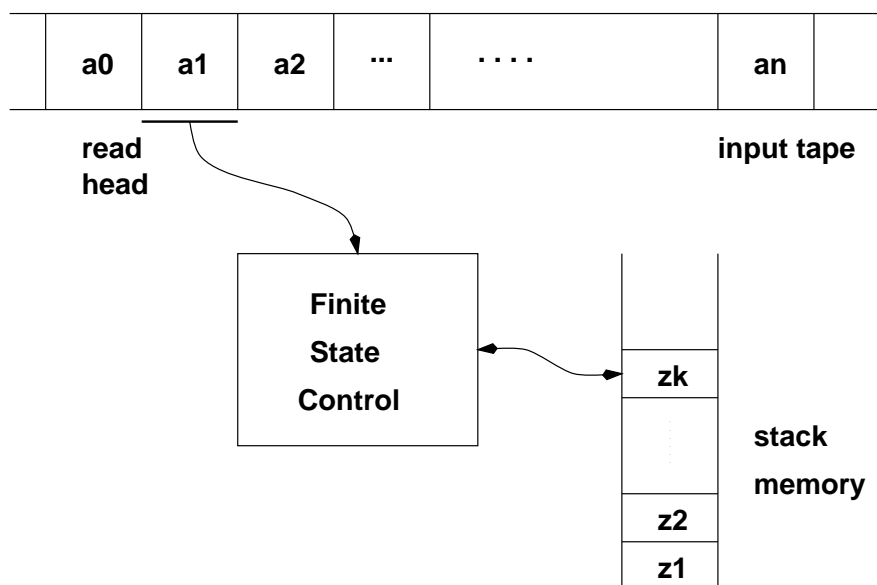
Chapter 2

The Theory of Bottom-Up Parsing

This section of notes looks at the theory and algorithms underlying parser generators like Bison and YACC. We start with push-down automata (slightly extended) and then consider how to make them into *deterministic* parsers. This leads to the idea of an *LR-parser* of which the parsers generated by Bison are examples.

2.1 Push-down Automata

By adding a *stack memory* to Finite Automata, its complexity and power is increased. We call these machines *Push-down Automata (PDA)*.



Each action of the machine may involve:

- change to the FSC state
- pushing or popping the stack

- advance to the next input symbol

The action of the machine may **depend on**:

- the current FSC state
- the current input symbol
- the current top-of-stack symbol(s)

For the machine to **accept** an input string as a sentence of the language, it must reach a specified goal state, with the input exhausted and the stack empty.

Example 2.1.1

$\{0^n 1^n \mid n \geq 1\}$

This language cannot be recognised by a *FA* since *FAs* can't keep count beyond a pre-determined limit. ($2n + 1$ states would be required, but n is not known. That is, unless n is bounded, the machine is not finite. Recall COMP2600.)

Here is an *ad hoc* design of a *PDA* for this language:

- first phase : stack 0s (state q_1)
- second phase : pop 0s, match with 1s on input (state q_2)
- if the stack is empty and the input is exhausted in the goal state, accept the string.

Theorem 2.1.2

The class of languages recognised by *PDA*s is exactly the class of *context-free* languages.

Since *PDA* transitions can modify the stack as well as change the FSC state, it is convenient to write transitions as a function δ of type:

$$\delta : (\text{state}, \text{input}, \text{tos}) \rightarrow (\text{state}, \text{string})$$

The string in the result is the symbols with which to *replace* the top-of-stack symbol. This is just a notational device makes it simple to specify pushes and pops in a uniform way. To simplify the notation for testing for empty stack, we use marker symbol Z as the base element of the stack.

Continuing the example:

$$\begin{aligned} \delta(q_0, 0, Z) &= q_1/Z0 && \text{— push first 0} \\ \delta(q_1, 0, 0) &= q_1/00 && \text{— push 0s} \\ \delta(q_1, 1, 0) &= q_2/\epsilon && \text{— start popping 0s} \\ \delta(q_2, 1, 0) &= q_2/\epsilon && \text{— pop 0s} \\ \delta(q_2, _, Z) &= q_3/\epsilon && \text{— accept} \end{aligned}$$

PDA configurations are written as triples: (state, remaining input, stack) Note: when working with *EPDA*s it is notationally convenient to draw stacks with the top to the *right*.

Tracing the parse of a sample string:

$$\begin{aligned}
(q_0, 000111, Z) &\Rightarrow (q_1, 00111, Z0) \\
&\Rightarrow (q_1, 0111, Z00) \\
&\Rightarrow (q_1, 111, Z000) \\
&\Rightarrow (q_2, 11, Z00) \\
&\Rightarrow (q_2, 1, Z0) \\
&\Rightarrow (q_2, _, Z) \\
&\Rightarrow (q_3, _, _)
\end{aligned}$$

A trace where the string is rejected:

$$\begin{aligned}
(q_0, 0010, Z) &\Rightarrow (q_1, 010, Z0) \\
&\Rightarrow (q_1, 10, Z00) \\
&\Rightarrow (q_2, 0, Z0) \\
&\Rightarrow ???
\end{aligned}$$

No transition applies and the machine is “stuck” without reaching the goal state.

2.2 Extended Push-Down Automata

We can extend the *PDA* control such that transitions depend on the *top of stack string*, rather than just one symbol. Why bother? Because *EPDAs* form the basis of bottom-up parsing algorithms.

From Context-Free Grammars to Extended Push-Down Automata

The translation is as follows. We use only two states: q_0 (which is both the initial state and the processing state), and q_1 (the goal state).

- For all terminal symbols t , add the transition:

$$\delta(q_0, t, _) = q_0/t$$

That is, terminal symbols may be pushed from input onto the stack. This is known as a *shift* action.

- For all productions $A \rightarrow \alpha$, add the transition:

$$\delta(q_0, _, \alpha) = q_0/A$$

That is, if the top-of-stack string matches the right-hand side of a production, replace it by the left-hand side (a single non-terminal). This is known as a *reduce* action.

- For the start symbol S , add the transition:

$$\delta(q_0, _, ZS) = q_1/\epsilon$$

That is, if the stack contains only the start symbol, go to the goal state (accept the string).

If the start symbol is on the top of stack then it indicates that the input matched so far can be derived from it. This reflects the fact that a bottom-up parse is taking place. Observe that the translation guarantees that we will always obtain a *non-deterministic EPDA*.

Example 2.2.1 (Deriving an EPDA from a CFG)

Derive an EPDA for the following CFG:

$$\begin{aligned} E &\rightarrow T \mid E + T \\ T &\rightarrow F \mid T * F \\ F &\rightarrow \mathbf{id} \mid (E) \end{aligned}$$

Shift actions:

$$\begin{aligned} \delta(q_0, +, _) &= q_0/+ \\ \delta(q_0, *, _) &= q_0/* \\ \delta(q_0, \mathbf{id}, _) &= q_0/id \\ \delta(q_0, (, _) &= q_0/(\\ \delta(q_0,), _) &= q_0/ \end{aligned}$$

Reduce actions:

$$\begin{aligned} \delta(q_0, _, T) &= q_0/E \\ \delta(q_0, _, E + T) &= q_0/E \\ \delta(q_0, _, F) &= q_0/T \\ \delta(q_0, _, T * F) &= q_0/T \\ \delta(q_0, _, id) &= q_0/F \\ \delta(q_0, _, (E)) &= q_0/F \end{aligned}$$

Accept:

$$\delta(q_0, _, ZE) = q_1/\epsilon$$

Example 2.2.2 (Parsing a sentence of the language)

(Oracle-guided through the non-determinism . . .)

$$\begin{aligned} (q_0, \mathbf{id} * \mathbf{id}, Z) &\Rightarrow (q_0, * \mathbf{id}, Z \mathbf{id}) \\ &\Rightarrow (q_0, * \mathbf{id}, ZF) \\ &\Rightarrow (q_0, * \mathbf{id}, ZT) \\ &\Rightarrow (q_0, \mathbf{id}, ZT*) \\ &\Rightarrow (q_0, \mathbf{id}, ZT * \mathbf{id}) \\ &\Rightarrow (q_0, _, ZT * F) \\ &\Rightarrow (q_0, _, ZT) \\ &\Rightarrow (q_0, _, ZE) \\ &\Rightarrow (q_1, _, _) \\ &\Rightarrow \text{accept} \end{aligned}$$

Note how the stack contains a *sentential form representing the part of the input that has been matched so far*. Note how the parse follows the *reverse of a right-most derivation*.

Exercise 2.2.3

Work through traces for some more parses of both valid and invalid strings. Explore some blind alleys.

The *EPDA* model presented (as derived from *CFGs*) is non-deterministic in the sense that choices may be possible at various steps in the parse. Clearly this is unsatisfactory as the basis of a translator — backtracking is potentially very inefficient. Furthermore, semantic processing and translation may be interleaved with the parsing, so any backtracking will also require the undoing of such processing — a great (and unnecessary) complication. The key to achieving determinism (if that is possible — the grammar may be ambiguous, for example) is once again by considering the remaining input.

For example, by taking into account that the next symbol is $*$, we can determine that the transition:

$$(q_0, *id, ZF) \Rightarrow (q_0, *id, ZT)$$

is the right choice rather than

$$(q_0, *id, ZF) \Rightarrow (q_0, id, ZF*)$$

since this must lead to a blind alley — there is no right hand side of a production that can match $F*$.

Using this idea, it is possible to modify the above *EPDA* to make it deterministic. The following table gives the basic idea:

$$\begin{aligned} \delta(q_0, +, E) &= q_0/E+ \\ \delta(q_0, +, T) &= q_0/E \\ \delta(q_0, +, T * F) &= q_0/T \\ \delta(q_0, *, T) &= q_0/T* \\ \delta(q_0, *, T * F) &= q_0/T \\ \delta(q_0, id, +) &= q_0/+ id \\ \delta(q_0, id, *) &= q_0/* id \\ \delta(q_0, id, () &= q_0/(id \\ \delta(q_0, id, Z) &= q_0/Zid \\ \delta(q_0, (, +) &= q_0/+ (\\ \delta(q_0, (, *) &= q_0/* (\\ \delta(q_0, (, Z) &= q_0/Z(\\ \delta(q_0,), E) &= q_0/E) \\ \delta(q_0,), T * F) &= q_0/T) \end{aligned}$$

etc. . . .

The uniform computation of such tables is the basis of *LR* parser generation.

2.3 Deterministic Bottom-Up Parsing

The approach we will concentrate on is *LR(1)* — **L**eft to right scan of input; **R**everse of a **R**ightmost derivation; **1** symbol lookahead. There are variations on this approach. *SLR(1)* (Simple *LR(1)*) and *LALR(1)* (LaLonde *LR(1)*) which is the one used in parser generators such as Bison. The *LR* parsers are modifications of the *EDPA* model to eliminate nondeterminism by taking account of the lookahead token *and* by using stack symbols that represent more information than simple grammar symbols.

Recall the mapping from context-free grammars to EPDAs. Nondeterministic parses arise in the choice between whether to:

1. *shift* the next input symbol onto the stack; *OR*
2. *reduce* the top of stack string to a single non-terminal; *AND*
3. *which production* to use for the reduction.

LR parsers are not guaranteed to avoid all of these problems — unresolved choices between (1) and (2) are *shift/reduce* conflicts, unresolved choices within (3) are *reduce/reduce* conflicts.

Suppose we have the following *EPDA* configuration:

$$(q_0, +\mathbf{id} \neg, ZT * F)$$

- Should we shift $+$ onto the stack?
- Should we reduce by $T \rightarrow F$?
- Should we reduce by $T \rightarrow T * F$?

A deterministic choice between these options can be achieved by considering the *lookahead symbol* and by keeping *more information on the stack* about the progress of the parse. Roughly speaking:

- **shift** if the next input symbol occurs in the production such that the top of stack string is the part of the production preceding that symbol. In the example above we would not shift because $+$ is incompatible with F as there is no derivable sentential form containing $F+$. We should only shift $+$ if E is on top of stack since the only occurrence of $+$ is in the production $E \rightarrow E + T$.
- **reduce** if the top of stack string is a complete right-hand side of some production and the lookahead symbol is a follower of the left-hand side non-terminal of that production (in that context). In the example above we should reduce since the top of stack string is the right-hand side of $T \rightarrow T * F$ and $+$ can follow that production. However, there is still an outstanding issue: another available action is to reduce by $T \rightarrow F$, but that would leave $T + T$ on the stack which would be incorrect. How can we determine which reduce action to choose?
- We can determine **which production** by which to reduce, by making the stack symbols represent terminals and non-terminals *in a particular production*, thus distinguishing between the F in the two different productions for T .

In fact, rather than stack symbols being particular symbols in productions, we refer to *positions between symbols in a production*. A production annotated with a position is called an *LR-item*.

For example

$$\begin{aligned} E &\rightarrow \bullet E + T \\ E &\rightarrow E \bullet + T \\ E &\rightarrow E + \bullet T \\ E &\rightarrow E + T \bullet \end{aligned}$$

are all LR-items.

How does this help to make the parser deterministic? Rather than just a sequence of grammar symbols T , $+$, F on top of the parser stack, we would have the positions:

$T \rightarrow T * F \bullet$
$T \rightarrow T * \bullet F$
$T \rightarrow T \bullet * F$
$T \rightarrow \bullet T * F$
...

thus eliminating the possibility of reducing with the $T \rightarrow F$ production.

Closures

- If the stack symbol indicates the position $E \rightarrow E \bullet + T$ has been reached (by recognising an E) then if $+$ is the lookahead, the parser will shift position $E \rightarrow E + \bullet T$ onto the stack.
- If the stack indicates that position $E \rightarrow E + \bullet T$ has been reached, we now want to recognise a T . Since T is a non-terminal, recognising it must eventually involve some *reduction*. That is, at some stage we must match F or $T * F$ so that $T \rightarrow F$ or $T \rightarrow T * F$ can be used to reduce. Furthermore, this will require that we first match an **id** or (E) so that $F \rightarrow \mathbf{id}$ or $F \rightarrow (E)$ can be used to reduce.

In other words, if we are in position like $E \rightarrow E + \bullet T$ then we are also in a number of possible *consistent positions* — the beginning of all productions for T :

$$\begin{aligned} T &\rightarrow \bullet T * F \\ T &\rightarrow \bullet F \end{aligned}$$

The LR-item $T \rightarrow \bullet F$ indicates that we must also be at the beginning of a production for F .

A maximal collection of consistent LR-items is called a **closure**. Closures are given names (like s_1 , s_2 etc.) and are usually called **states** (which can be slightly confusing¹). For example:

$$\{ E \rightarrow E + \bullet T,$$

¹The closures themselves are only used in the process of generating the parser. When the parser is operational, only the state *identifiers* need appear on the stack.

$$\begin{aligned} T &\rightarrow \bullet T * F, \\ T &\rightarrow \bullet F, \\ F &\rightarrow \bullet \mathbf{id}, \\ F &\rightarrow \bullet (E) \} \end{aligned}$$

It is easy to see from this closure that the lookahead symbol must be either **id** or '(' if the string is valid. If the lookahead happens to be '(', then the symbol pushed onto the stack will be the closure of $F \rightarrow (\bullet E)$. That is:

$$\begin{aligned} \{ &F \rightarrow (\bullet E), \\ &E \rightarrow \bullet T, \\ &E \rightarrow \bullet E + T, \\ &T \rightarrow \bullet F, \\ &T \rightarrow \bullet T * F, \\ &F \rightarrow \bullet \mathbf{id}, \\ &F \rightarrow \bullet (E) \} \end{aligned}$$

Now suppose position $F \rightarrow (E)\bullet$ has been reached. We want to reduce by this production but only for some particular lookahead symbols (otherwise the string is invalid). It is the choice of these **follower sets** that distinguishes *LR*, *SLR* and *LALR*:

- *SLR* simply calculates the follower symbols as we did for top-down parsers in the previous section.
- *LR* and *LALR* take account of the *context* of the parse, for example distinguishing between the followers of E within parentheses and E at a top-level.

The *LR* Parsing Algorithm

LR parsers are based on a uniform driver program. Different grammars produce different *parsing tables* which are used to direct the parsing algorithm. The main job of Bison is to construct parsing tables from grammars. Parsing tables consist of two parts:

- an *action* function
- a *goto* function

The *action* function depends on the state on top of stack s_m and the current lookahead symbol a_i . The value of $action[s_m, a_i]$ can be one of the following:

- *shift* s , where s is a state
- *reduce* by a production $A \rightarrow \alpha$
- *accept*
- *error*

The *goto* function depends on a state and a *non-terminal* symbol, and returns a state. It cooperates with the reduce actions. The *configuration* of an *LR* parser is the same as an *EPDA* but we can leave out the unchanging and hence uninteresting FSC q_0 .

The stack contains a representation of the part of the sentential form matched so far, following the reverse of a rightmost derivation. However, rather than grammar symbols, the stack contains *states* representing item closures.

Suppose the current configuration is:

$$(a_i a_{i+1} \dots a_k \dashv, \quad s_0 s_1 \dots s_m)$$

The next move is given by looking up the parsing table to find $action[s_m, a_i]$:

- If $action[s_m, a_i] = shift\ s$, the parser configuration becomes:

$$(a_{i+1} \dots a_k \dashv, \quad s_0 s_1 \dots s_m s)$$

- If $action[s_m, a_i] = reduce\ A \rightarrow \alpha$, the parser configuration becomes:

$$(a_i a_{i+1} \dots a_k \dashv, \quad s_0 s_1 \dots s_{m-r} s)$$

where $goto[s_{m-r}, A] = s$ and r is the length of α .

- If $action[s_m, a_i] = accept$, the string is accepted.
- Otherwise, the string is rejected.

The *LR* parser's *shift* action is just like the *EPDA* shift. Consider an *EPDA* for the expression grammar. Suppose E is on top of stack and $+$ is on input. A shift action would push $+$ and advance the read head.

In an *LR* parser, rather than E on top of stack it would be a state representing a set of items including $E \rightarrow E \bullet + T$. The shift action advances the read head and pushes a state representing a set of items including $E \rightarrow E + \bullet T$.

The *EPDA* *reduce* action replaces some top of stack sequence matching a production's right-hand side with the production's left-hand side non-terminal. The *LR* action is just the same except that we need to push a *state* rather than just a non-terminal.

In more detail, the effect of a reduce action in an *LR* parser is as follows. Suppose the production by which we are reducing is:

$$A \rightarrow x_1 x_2 \dots x_k$$

Then the top of stack state must represent a closure including an item:

$$A \rightarrow x_1 x_2 \dots x_k \bullet$$

Furthermore, the top k states will represent closures including the following sequence of items:

$$A \rightarrow x_1 x_2 \dots x_k \bullet$$

$$A \rightarrow x_1 x_2 \dots \bullet x_k$$

$$\begin{aligned} & \vdots \\ & A \rightarrow x_1 \bullet x_2 \dots x_k \end{aligned}$$

The first step in the reduce action is to pop those k states from the stack. The new top of stack will represent a closure containing:

$$A \rightarrow \bullet x_1 x_2 \dots x_k$$

and something like:

$$B \rightarrow \dots \bullet A \dots$$

In a sense we now want to “shift” an A . Since the k states just popped indicate that we have matched an A , such a shift is valid.

The second step is therefore to push a state representing a closure containing the item:

$$B \rightarrow \dots A \bullet \dots$$

In the parse table this state is given by the *goto* function. If the state representing $B \rightarrow \dots \bullet A \dots$ is called s_m and the state representing $B \rightarrow \dots A \bullet \dots$ is s then:

$$goto[s_m, A] = s$$

Example 2.3.1 (Parsing table for the expression grammar)

1. $E \rightarrow T$
2. $E \rightarrow E + T$
3. $T \rightarrow F$
4. $T \rightarrow T * F$
5. $F \rightarrow \mathbf{id}$
6. $F \rightarrow (E)$

state	action					goto			
	+	*	id	()	⊖	<i>E</i>	<i>T</i>	<i>F</i>
1			s5	s6			s2	s3	s4
2	s7					acc			
3	r1	s8			r1	r1			
4	r3	r3			r3	r3			
5	r5	r5			r5	r5			
6			s5	s6			s9	s3	s4
7			s5	s6				s10	s4
8			s5	s6					s11
9	s7				s12				
10	r2	s8			r2	r2			
11	r4	r4				r4			
12	r6	r6			r6	r6			

(*si* means shift state i . rk means reduce with production k . *acc* means accept. *blank* entries mean error, so errors are detected as early as possible.)

Example 2.3.2 (A sample parse using the table above)Parsing $(\mathbf{id} + \mathbf{id}) \dagger$ trace:

```

(id + id)  $\dagger$ , 1
id + id)  $\dagger$ , 1 6
  +id)  $\dagger$ , 1 6 5
  +id)  $\dagger$ , 1 6 4
  +id)  $\dagger$ , 1 6 3
  +id)  $\dagger$ , 1 6 9
    id)  $\dagger$ , 1 6 9 7
      )  $\dagger$ , 1 6 9 7 5
      )  $\dagger$ , 1 6 9 7 4
      )  $\dagger$ , 1 6 9 7 10
      )  $\dagger$ , 1 6 9
         $\dagger$ , 1 6 9 12
         $\dagger$ , 1 4
         $\dagger$ , 1 3
         $\dagger$ , 1 2
          accept

```

Example 2.3.3 (Corresponding *EPDA* trace)

```

(id + id)  $\dagger$ , Z
id + id)  $\dagger$ , Z (
  +id)  $\dagger$ , Z (id
  +id)  $\dagger$ , Z (F
  +id)  $\dagger$ , Z (T
  +id)  $\dagger$ , Z (E
    id)  $\dagger$ , Z (E+
      )  $\dagger$ , Z (E + id
      )  $\dagger$ , Z (E + F
      )  $\dagger$ , Z (E + T
      )  $\dagger$ , Z (E
         $\dagger$ , Z (E)
         $\dagger$ , Z F
         $\dagger$ , Z T
         $\dagger$ , Z E
          accept

```

For example, *state 6* represents the set of items:

$$\{ F \rightarrow (\bullet E), \quad E \rightarrow \bullet T, \quad E \rightarrow \bullet E + T, \quad T \rightarrow \bullet F, \quad T \rightarrow \bullet T * F, \quad F \rightarrow \bullet \text{id}, \quad F \rightarrow \bullet (E) \}$$

State 5 represents the set of items: $\{ F \rightarrow \text{id} \bullet \}$

State 4 represents the set of items: $\{ T \rightarrow F \bullet \}$

State 3 represents the set of items: $\{ E \rightarrow T \bullet, \quad T \rightarrow T \bullet * F \}$

State 9 represents the set of items: $\{ F \rightarrow (E \bullet), \quad E \rightarrow E \bullet + T \}$

2.4 Constructing *SLR* tables

There are two aspects to the construction of LR parsers:

- Identifying the **states**. That is, calculating the closures of item sets;
- Generating the **actions**. That is, the *shift* and *reduce* entries in the table

Building the states (i.e. the closed set of items) follows a symbolic trace of all possible parsing moves. We begin with s_1 being the closure of a position preceding the start symbol. If the parser reaches a position following the start symbol at the same time as being at the end of input, the string is accepted. Shift actions are generated by considering all valid lookaheads from s_1 . Each such action may lead to a new state, say s_m . Shift actions for all lookaheads from this new state are then generated, and so on.

Calculating Closures

First we require a routine for working out the *closure of sets of items*. Essentially, the closure function looks through a set of items for positions *immediately preceding a non-terminal*, e.g.:

$$A \rightarrow \dots \bullet B \dots$$

and adds into the set of items all the productions for B with the position at the beginning of the right hand side, i.e.:

$$B \rightarrow \bullet \alpha$$

The process is repeated on the extended set until no more items are added. Here is an algorithm:

```

fn closure(I: setOfItems) : setOfItems;
  setOfItems temp, res = I;
  repeat
    temp = res;
    for all  $A \rightarrow \dots \bullet B \dots \in$  temp
      for all  $B \rightarrow \bullet \alpha$  in grammar
        res = res  $\cup$   $B \rightarrow \bullet \alpha$ ;
  until res = temp; // i.e. no new items added
  return res;

```

The grammar is augmented with a new start symbol and production: $S' \rightarrow S$ where S is the original start symbol. The initial state s_1 represents the closure of the item $S' \rightarrow \bullet S$ and the input string is accepted when the parser is about to reduce by $S' \rightarrow S \bullet$. The only valid follower of S' is \perp , end of input.

The process of constructing closures and generating actions commences by supposing that s_1 is on top of the parse stack and considering each possible lookahead symbol. Each lookahead will lead to a *shift* action and possibly a new state, or a *reduce* action if the position is at the end of the production and the lookahead is a valid follower of the left-hand side non-terminal. For *SLR* parsers, the followers are calculated in the manner described in the lecture notes on top-down parsers. The process continues by supposing that each of those new states is on top of the parse stack, considering all possible lookaheads and so on, until all states have been treated. Note that we only need the closures during *construction* of the parse table. In the table itself we simply use a *name* s_k to represent closure I_k .

Algorithm to construct *SLR* parse tables:

```

i = 0; p = 1; I1 = closure({S' → •S});
while (i < p)
  i++;
  for all B → α• ∈ Ii
    if B → α• is S' → S•
      action[i, ⊥] = accept
    else for all x ∈ follower(B)
      action[i, x] = reduce by B → α
  for all terminal symbols x
    temp = closure({A → αx•β | A → α•xβ ∈ Ii});
    if temp = Ik where k ≤ p
      action[i, x] = shift k;
    else
      p++;
      Ip = temp;
      action[i, x] = shift p;
  for all non-terminal symbols X
    temp = closure({A → αX•β | A → α•Xβ ∈ Ii});
    if temp = Ik where k ≤ p
      goto[i, X] = shift k;
    else
      p++;
      Ip = temp;
      goto[i, X] = shift p;

```

Example 2.4.1 (Very simple)

Trace the *SLR* table construction algorithm for the simple grammar:

1. $E \rightarrow E + \text{num}$
2. $E \rightarrow \text{num}$

First we augment the grammar with a new start symbol:

$$G \rightarrow E$$

Calculate the follower sets:

$$G: \quad \vdash$$

$$E: \quad +, \vdash$$

Trace the algorithm:

$$\begin{aligned}
 & i = 0 \\
 & p = 0 \\
 & I_1 = \{G \rightarrow \bullet E, E \rightarrow \bullet E + \mathbf{num}, E \rightarrow \bullet \mathbf{num}\} \\
 & \text{---} \\
 & i = 1 \\
 & x = \mathbf{num} \\
 & I_2 = \{E \rightarrow \mathbf{num} \bullet\} \\
 & p = 2 \\
 & \bullet \text{ action}[1, \mathbf{num}] = \text{shift2} \\
 & X = E \\
 & I_3 = \{G \rightarrow E \bullet, E \rightarrow E \bullet + \mathbf{num}\} \\
 & p = 3 \\
 & \bullet \text{ goto}[1, E] = \text{shift3} \\
 & \text{---} \\
 & i = 2 \\
 & \bullet \text{ action}[2, \vdash] = \text{reduce2} \\
 & \bullet \text{ action}[2, +] = \text{reduce2} \\
 & \text{---} \\
 & i = 3 \\
 & \bullet \text{ action}[3, \vdash] = \text{accept} \\
 & x = + \\
 & I_4 = \{E \rightarrow E + \bullet \mathbf{num}\} \\
 & p = 4 \\
 & \bullet \text{ action}[3, +] = \text{shift4} \\
 & \text{---} \\
 & i = 4 \\
 & x = \mathbf{num} \\
 & I_5 = \{E \rightarrow E + \mathbf{num} \bullet\} \\
 & p = 5 \\
 & \bullet \text{ action}[4, \mathbf{num}] = \text{shift5}
 \end{aligned}$$

—
 $i = 5$

- $action[5, -] = reduce1$
- $action[5, +] = reduce1$

The *SLR* parse table is therefore:

	+	num	-	E
1		s2		s3
2	r2		r2	
3	s4		acc	
4		s5		
5	r1		r1	

Example 2.4.2 (Less simple)

Trace the *SLR* table construction algorithm for the grammar:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow \mathbf{id}$
6. $F \rightarrow (E)$
0. $G \rightarrow E$

$$I_1 = \{G \rightarrow \bullet E, E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet \mathbf{id}, F \rightarrow \bullet (E)\}$$

Suppose I_1 is on top of stack.

Suppose lookahead is **id**.

$$I_2 = \{F \rightarrow \mathbf{id} \bullet\}$$

- $action[1, \mathbf{id}] = shift\ 2$

Suppose lookahead is (.

$$I_3 = \{F \rightarrow (\bullet E), E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet \mathbf{id}, F \rightarrow \bullet (E)\}$$

- $action[1, (] = shift\ 3$

Suppose an E has just been parsed.

$$I_4 = \{G \rightarrow E \bullet, E \rightarrow E \bullet + T\}$$

- $goto[1, E] = shift\ 4$

Suppose a T has just been parsed.

$$I_5 = \{E \rightarrow T \bullet, T \rightarrow T \bullet * F\}$$

- $goto[1, T] = shift\ 5$

Suppose an F has just been parsed.

$$I_6 = \{T \rightarrow F \bullet\}$$

- $goto[1, F] = shift\ 6$

I_2 tos.

- $action[2, -] = reduce\ 5$
- $action[2, +] = reduce\ 5$
- $action[2, *] = reduce\ 5$
- $action[2,)] = reduce\ 5$

I_3 tos.

Lookahead **id**.

$\{F \rightarrow \mathbf{id}\bullet\} = I_2$

- $action[3, \mathbf{id}] = shift\ 2$

Lookahead **(**.

$\{F \rightarrow (\bullet E), E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet \mathbf{id}, F \rightarrow \bullet (E)\} = I_3$

- $action[3, (] = shift\ 3$

Parsed E .

$I_7 = \{F \rightarrow (E\bullet), E \rightarrow E\bullet + T\}$

- $goto[3, E] = shift\ 7$

Parsed T .

$\{E \rightarrow T\bullet, T \rightarrow T\bullet * F\} = I_5$

- $goto[3, T] = shift\ 5$

Parsed F .

$\{T \rightarrow F\bullet\} = I_6$

- $goto[3, F] = shift\ 6$

I_4 tos.

- $action[4, -] = accept$

Lookahead **+**.

$I_8 = \{E \rightarrow E + \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet \mathbf{id}, F \rightarrow \bullet (E)\}$

- $action[4, +] = shift\ 8$

I_5 tos.

- $action[5, -] = reduce\ 2$
- $action[5, +] = reduce\ 2$
- $action[5,)] = reduce\ 2$

Lookahead *****.

$I_9 = \{T \rightarrow T * \bullet F, F \rightarrow \bullet \mathbf{id}, F \rightarrow \bullet (E)\}$

- $action[5, *] = shift\ 9$

I_6 tos.

- $action[6, -] = reduce\ 4$
- $action[6, +] = reduce\ 4$
- $action[6, *] = reduce\ 4$
- $action[6,)] = reduce\ 4$

I_7 tos.

Lookahead **)**.

$I_{10} = \{F \rightarrow (E)\bullet\}$

- $goto[7,)] = shift\ 10$

Lookahead **+**.

- $action[7, +] = shift\ 8$

I_8 tos.

Lookahead **id**.

- $action[8, \mathbf{id}] = shift\ 2$

Lookahead **(**.

- $action[8, (] = shift\ 3$

Parsed T .

$$I_{11} = \{E \rightarrow E + T \bullet, T \rightarrow T \bullet * F\}$$

- $goto[8, T] = shift\ 11$

Parsed F .

- $goto[8, F] = shift\ 6$

I_9 tos.

Lookahead **id**.

- $action[9, \mathbf{id}] = shift\ 2$

Lookahead **(**.

- $action[9, (] = shift\ 3$

Parsed F .

$$I_{12} = \{T \rightarrow T * F \bullet\}$$

- $goto[9, F] = shift\ 12$

I_{10} tos.

- $action[10, -] = reduce\ 6$
- $action[10, +] = reduce\ 6$
- $action[10, *] = reduce\ 6$
- $action[10,)] = reduce\ 6$

I_{11} tos.

- $action[11, -] = reduce\ 1$
- $action[11, +] = reduce\ 1$
- $action[11,)] = reduce\ 1$

Lookahead *****.

$$\{T \rightarrow T * \bullet F, \dots\} = I_9$$

- $goto[11, F] = shift\ 9$

I_{12} tos.

- $action[12, -] = reduce\ 3$
- $action[12, +] = reduce\ 3$
- $action[12, *] = reduce\ 3$
- $action[12,)] = reduce\ 3$

Collection of closures computed:

$$I_1 = \{G \rightarrow \bullet E, E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet \mathbf{id}, F \rightarrow \bullet (E)\}$$

$$I_2 = \{F \rightarrow \mathbf{id} \bullet\}$$

$$I_3 = \{F \rightarrow (\bullet E), E \rightarrow \bullet E + T, E \rightarrow \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet \mathbf{id}, F \rightarrow \bullet (E)\}$$

$$I_4 = \{G \rightarrow E \bullet, E \rightarrow E \bullet + T\}$$

$$I_5 = \{E \rightarrow T \bullet, T \rightarrow T \bullet * F\}$$

$$I_6 = \{T \rightarrow F \bullet\}$$

$$I_7 = \{F \rightarrow (E \bullet), E \rightarrow E \bullet + T\}$$

$$I_8 = \{E \rightarrow E + \bullet T, T \rightarrow \bullet T * F, T \rightarrow \bullet F, F \rightarrow \bullet \text{id}, F \rightarrow \bullet (E)\}$$

$$I_9 = \{T \rightarrow T * \bullet F, F \rightarrow \bullet \text{id}, F \rightarrow \bullet (E)\}$$

$$I_{10} = \{F \rightarrow (E) \bullet\}$$

$$I_{11} = \{E \rightarrow E + T \bullet, T \rightarrow T \bullet * F\}$$

$$I_{12} = \{T \rightarrow T * F \bullet\}$$

Conflicts in LR Parsers

Recall from §1.5 that non-*LR* grammars (e.g. ambiguous grammars) lead to **shift/reduce conflicts** and/or **reduce/reduce conflicts**. They arise when some position in the parse table has *two entries* — that is, for some state on top of stack and some lookahead symbol, the parser may have *more than one option*: either a shift and a reduce, or several reduces. In that case, the grammar should be modified to eliminate the conflict.

Shift/Reduce Conflicts

A classic example of a shift/reduce conflict is the *dangling-else* problem. (The grammar is ambiguous):

$$\begin{aligned} S &\rightarrow \text{if } B \text{ then } S \\ S &\rightarrow \text{if } B \text{ then } S \text{ else } S \end{aligned}$$

Following through an *LR* table construction algorithm starting from the usual position:

$$\begin{aligned} I_1 = \{ &S' \rightarrow \bullet S, \\ &S \rightarrow \bullet \text{if } B \text{ then } S \\ &S \rightarrow \bullet \text{if } B \text{ then } S \text{ else } S\} \end{aligned}$$

if the sequence **if *B* then *S*** is parsed then the top of stack state will represent an item:

$$\begin{aligned} I_n = \{ &S \rightarrow \text{if } B \text{ then } S \bullet \\ &S \rightarrow \text{if } B \text{ then } S \bullet \text{ else } S\} \end{aligned}$$

Clearly **else** is a follower of *S* (observe the second production above), so there will be a conflict:

$$\begin{aligned} \text{action}[n, \text{else}] &= \text{shift } m \\ \text{action}[n, \text{else}] &= \text{reduce } 1 \end{aligned}$$

Reduce/Reduce Conflicts

The reduce/reduce example from §1.5 was:

$$\begin{aligned} \text{expr} &\rightarrow \text{const} \mid \text{var} \\ \text{var} &\rightarrow \text{id} \mid \dots \end{aligned}$$

$$const \rightarrow \mathbf{id} \mid \dots$$

For simplicity, suppose $expr$ is the start symbol, so \vdash is a valid follower of $expr$.

The *SLR* construction algorithm goes as follows (in part):

$$I_1 = \{G \rightarrow \bullet expr, \\ expr \rightarrow \bullet const, \\ expr \rightarrow \bullet var, \\ var \rightarrow \bullet \mathbf{id}, \\ const \rightarrow \bullet \mathbf{id}\}$$

I_1 tos.

Lookahead \mathbf{id} .

$$I_2 = \{var \rightarrow \mathbf{id}\bullet, con \rightarrow \mathbf{id}\bullet\}$$

$$action[1, \mathbf{id}] = shift\ 2$$

...

I_2 tos.

$$action[2, \vdash] = reduce\ by\ var \rightarrow \mathbf{id}$$

$$action[2, \vdash] = reduce\ by\ con \rightarrow \mathbf{id}$$

...

So there is a conflict between the choice of reduce-productions when s_2 is on top of stack and \vdash (at least) is the lookahead.

Chapter 3

Lexical Analysis and Finite State Automata

As you know, *Flex* takes a specification consisting of a sequence of regular expressions and produces a program that recognises strings that match those regular expressions. (It does some other things along the way, but this is the core behaviour.) How does it do it?

- Flex builds a *finite state automaton* (*FSA*) corresponding to the regular expressions;
- A specification of the *FSA*, e.g. a table, is passed to a standard routine (supplied by Flex) that imitates the behaviour of the *FSA* on the input data.

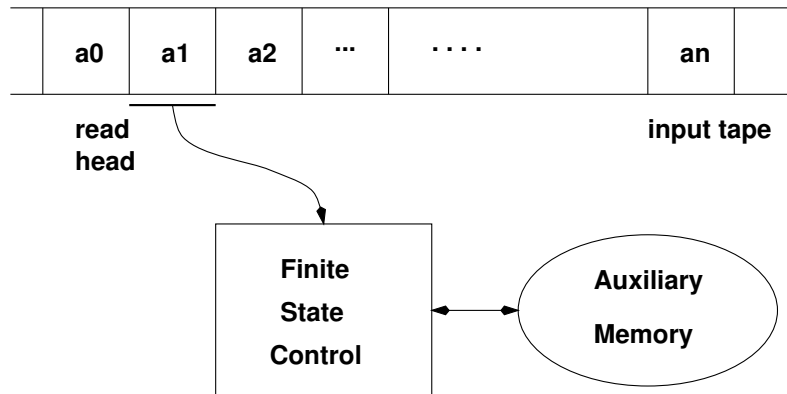
3.1 Languages and Automata

Recall that to define a language we can either:

1. Give a set of rules (i.e. a grammar) to produce all the legal strings (sentences) of the language.
2. Provide a machine (i.e. an algorithm) to recognise all the legal strings of the language.

There is a close relationship between the two approaches. Commonly we *define* a language by giving a grammar and then base *recognisers* on the corresponding machine.

Basic structure of formal machines



For the machine to accept an input string as a sentence of the language, it must reach a specified goal state, with the input exhausted. The kind of auxiliary memory in a machine determines the class of languages that the machine can recognise:

Language Class	Memory
regular	none
context-free	stack
context-sensitive	tape (bounded by input length)
unrestricted	unbounded tape

Finite State Automata

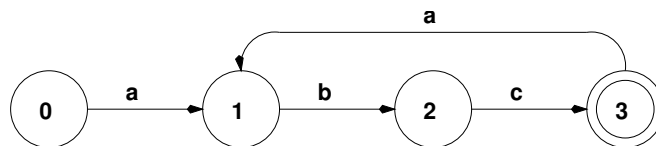
These are the *machines with no memory*, capable of recognising regular languages.

Example 3.1.1

The following machine recognises the language

$$\{(abc)^n \mid n \geq 1\}$$

(i.e. $abc, abcabc, abcabcabc, \dots$)



Note how the *states* indicate progress in string recognition:

q_1 : $(abc)^i a$ has been recognised ($i \geq 0$)

q_2 : $(abc)^i ab$ has been recognised ($i \geq 0$)

q_3 : $(abc)^i$ has been recognised ($i \geq 1$)

Theorem 3.1.2

The class of languages recognised by finite automata is *exactly* the class of regular languages (i.e., those that can be defined using regular expressions).

For our purposes, we're only interested in half this result: how to build a *FSA* from any given regular expression.

3.2 From Regular Expressions to Finite Automata

The *translation* of regular expressions to *NFA* (non-deterministic finite automata) is quite straightforward: *For each regular expression construct, there is a corresponding NFA construct.*

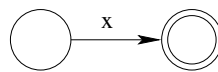
Here is a basic language of regular expressions:

- 'x' matches the character 'x'.
- 'r*' matches zero or more *r*'s where *r* is a regular expression.
- 'r+' matches one or more *r*'s.
- 'r?' matches zero or one *r*.
- 'rs' matches an *r* followed by an *s* where *r* and *s* are regular expressions.
- 'r|s' matches either an *r* or an *s*.
- '(r)' matches an *r*. Parentheses are used to override precedence.

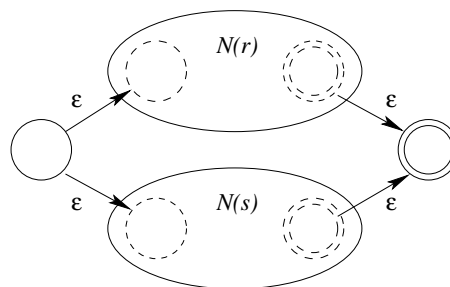
Corresponding NFAs

For each of the regexps above, there is a corresponding *NFA* construction as follows.

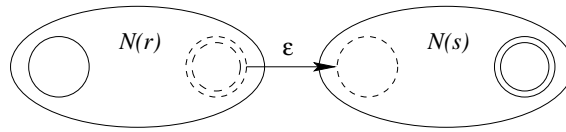
For regular expression *x*, construct the *NFA*:



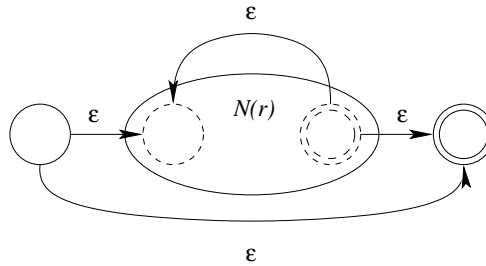
For the regular expression *r|s*, construct the *NFA*:



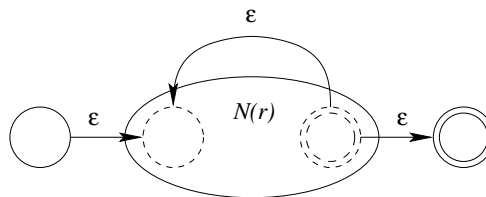
For the regular expression rs , construct the *NFA*:



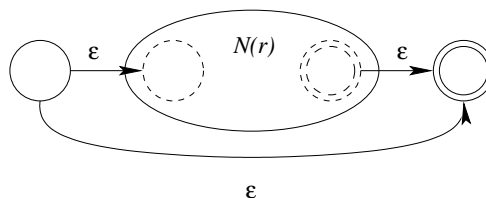
For the regular expression r^* , construct the *NFA*:



For the regular expression r^+ , construct the *NFA*:



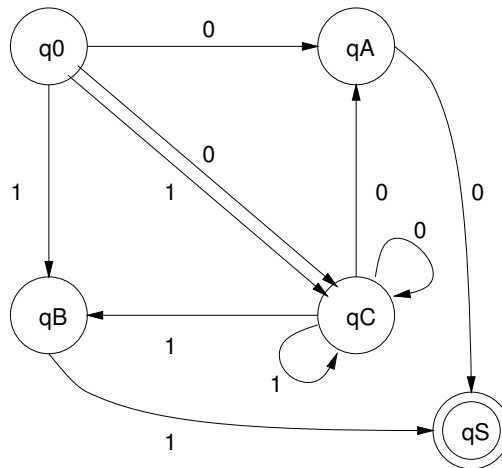
For the regular expression $r^?$, construct the *NFA*:



3.3 Making Them Deterministic

Unfortunately, the finite state automata thus generated from regular expressions may be *non-deterministic*. That's obvious from the common use of empty transitions, but in general it is possible to have two different transitions from a state for the same input symbol.

Here is an example of a *NFA*. (It was not generated using the algorithm above.) The language it recognises is sequences of 0s and 1s, ending with at least two 0s or two 1s. Note that there are arcs leaving nodes with the same labels:



The machine *can* accept a string if it is legal. That is, some transition sequence exists, ending in the goal state. For example, there is a transition sequence from $(q_0, 011)$ to $(\underline{q_S}, _)$ but there are dead-ends too, for example:

$$(q_0, 011) \Rightarrow (q_A, 11)$$

If we could look at input symbols further ahead we could decide whether the current symbol (0 or 1) was part of the end sequence (00 or 11), but this is not a general solution. Fortunately there *is* a good solution ...

Theorem 3.3.1

Every non-deterministic finite state automaton (*NFA*) can be mapped to an equivalent deterministic finite state automaton (*DFA*).

Equivalence is meant in the sense of the language recognised. The basic idea is to replace each *set* of states that are *reachable* in a non-deterministic machine, for the same input sequence, by one state. For the example above, if we coalesce states:

$$q_1 = \{q_A, q_C\}$$

$$q_2 = \{q_B, q_C\}$$

$$q_3 = \{q_A, q_C, q_S\}$$

$$q_4 = \{q_B, q_C, q_S\}$$

Observe the following intuitive correspondences:

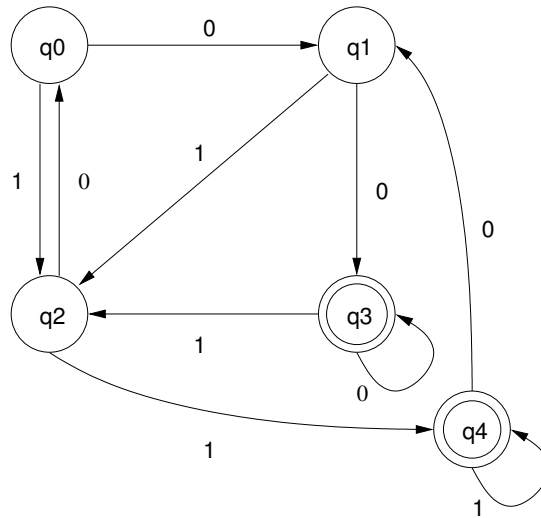
q_1 : one 0 recognised

q_2 : one 1 recognised

q_3 : several consecutive 0s recognised

q_4 : several consecutive 1s recognised

This leads to the following *DFA*:



Notice that there are now 2 goal states, since both q_3 and q_4 include q_S .

From *NFA* to *DFA*

In this section we outline an *algorithm* to build a *DFA* that recognises the same language as a given *NFA*. The basic idea is that the *DFA* we construct has states corresponding to *sets* of the *NFA* states that can be reached by scanning the same string, beginning in the start state q_0 .

Example 3.3.2

How did I construct the *DFA* above? Consider:

- What is the *set* of states we can reach from q_0 on input 0?
- What is the *set* of states we can reach from q_0 on input 1?
- What is the set of states we can reach from each state in those sets on input 0?
- etc.

$$\text{move}(q_0, 0) = \{q_A, q_C\}$$

$$\text{move}(q_0, 1) = \{q_B, q_C\}$$

$$\text{move}(\{q_A, q_C\}, 0) = \{q_A, q_C, q_S\}$$

$$\text{move}(\{q_A, q_C\}, 1) = \{q_B, q_C\}$$

$$\text{move}(\{q_B, q_C\}, 0) = \{q_B, q_C, q_S\}$$

$$\text{move}(\{q_B, q_C\}, 1) = \{q_A, q_C\}$$

$$\text{move}(\{q_A, q_C, q_S\}, 0) = \{q_A, q_C, q_S\}$$

$$\text{move}(\{q_A, q_C, q_S\}, 1) = \{q_B, q_C\}$$

$$\text{move}(\{q_B, q_C, q_S\}, 0) = \{q_A, q_C\}$$

$$\text{move}(\{q_B, q_C, q_S\}, 1) = \{q_B, q_C, q_S\}$$

The sets of states correspond to states q_1 to q_4 in the *DFA* informally described above and the transitions are given by *move*.

An algorithm to construct a *DFA* from a *NFA*

The algorithm consists of a process to construct the desired subsets of the *NFA* states, and the corresponding *DFA* transitions. An important sub-process is to compute sets of states that can be reached by ϵ -moves from some given state or states. (Note how the correspondence between regular expressions and *NFAs* given above introduces many ϵ -moves.

An ϵ -closure algorithm

For some subset T of the states of an *NFA*, the ϵ -closure of T is the set of states of the *NFA* that can be reached from one of the states in T by ϵ -moves alone.

Input: a set T of states of an *NFA*.

Output: the ϵ -closure of T (a set).

Algorithm:

```

push all states in  $T$  onto  $Stack$ ;
 $\epsilon$ -closure( $T$ ) =  $T$ ;
while  $Stack$  not empty do
     $t$  = pop( $Stack$ );
    for each  $u$  with an empty edge from  $t$  do
        add  $u$  to  $\epsilon$ -closure( $T$ );
        push  $u$  onto  $Stack$ ;

```

A subset construction algorithm

Let $move(T, a)$ be the set of states to which there is a transition on input a from one of the states in the set T . An *NFA* N with start state q_0 can be in any of the states in $T = \epsilon.closure(q_0)$ before scanning any input. If the input is a then N can move to any of $\epsilon.closure(move(T, a))$. And so on ...

The subset construction algorithm starts with $\epsilon.closure(q_0)$ as a *D*-state and generates other *D*-states by the process above. Once the *D*-states have been considered, they are *marked*.

Input: An *NFA* with an unmarked *D*-state = $\epsilon.closure(q_0)$.

Output: A *DFA* that recognises the same language as the *NFA*.

Algorithm:

```

while there is an unmarked state  $T$  in  $Dstates$  do
    mark  $T$ ;
    for each alphabet symbol  $a$  do
         $U = \epsilon.closure(move(T, a))$ ;
        if  $U$  not in  $Dstates$  then
            add  $U$  to  $Dstates$  as unmarked;
         $Dtran[T, a] = U$ ;

```

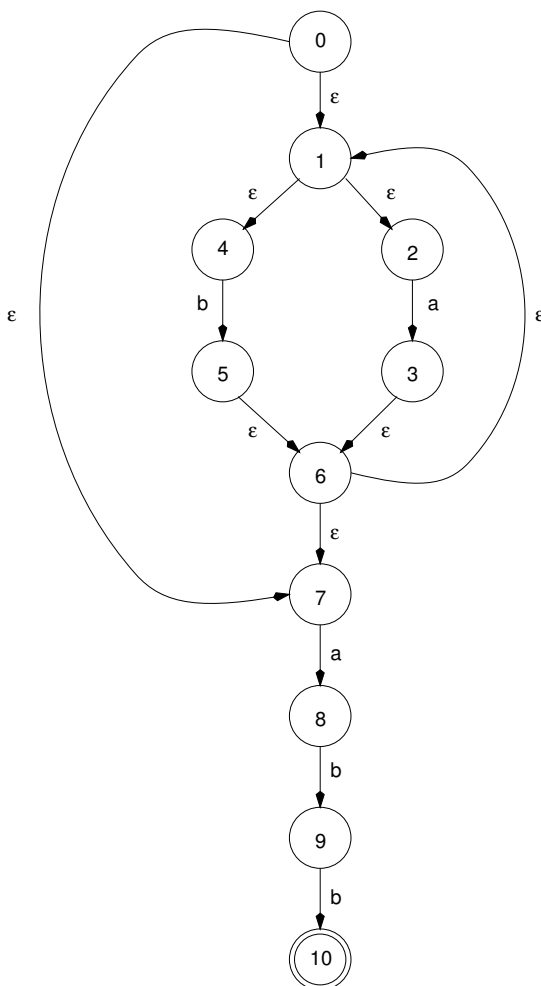
Now $Dtran$ is the transition table for D . The start state for D is $\epsilon.closure(q_0)$. The accept states for D are all those that contain an accept state of N . If the *NFA* has n states then the constructed *DFA* has at most $2^n - 1$ states, but usually far fewer.

Example 3.3.3

Using the correspondence outlined in §3.2, the regular expression:

$$(a|b)^*abb$$

may be translated to the following *NFA*:



To derive an equivalent *DFA*, apply the algorithm:

$$\epsilon.\text{closure}() = \{0, 1, 2, 4, 7\} = A$$

$$T_A^a = \{3, 8\}$$

$$\epsilon.\text{closure}(T_A^a) = \{1, 2, 3, 4, 6, 7, 8\} = B$$

$$DTran[A, a] = B$$

$$T_A^b = \{5\}$$

$$\epsilon.\text{closure}(T_A^b) = \{1, 2, 4, 5, 6, 7\} = C$$

$$DTran[A, b] = C$$

$$T_B^a = \{3, 8\}$$

$$\epsilon.\text{closure}(T_B^a) = B$$

$$DTran[B, a] = B$$

$$T_B^b = \{5, 9\}$$

$$\epsilon.\text{closure}(T_B^b) = \{1, 2, 4, 5, 6, 7, 9\} = D$$

$$DTran[B, b] = D$$

$$T_C^a = \{3, 8\}$$

$$\epsilon.\text{closure}(T_C^a) = B$$

$$DTran[C, a] = B$$

$$T_C^b = \{5\}$$

$$\epsilon.\text{closure}(T_C^b) = C$$

$$DTran[C, b] = C$$

$$T_D^a = \{3, 8\}$$

$$\epsilon.\text{closure}(T_D^a) = B$$

$$DTran[D, a] = B$$

$$T_D^b = \{5, \underline{10}\}$$

$$\epsilon.\text{closure}(T_D^b) = \{1, 2, 4, 5, 6, 7, \underline{10}\} = E$$

$$DTran[D, b] = E$$

$$T_E^a = \{3, 8\}$$

$$\epsilon.\text{closure}(T_E^a) = B$$

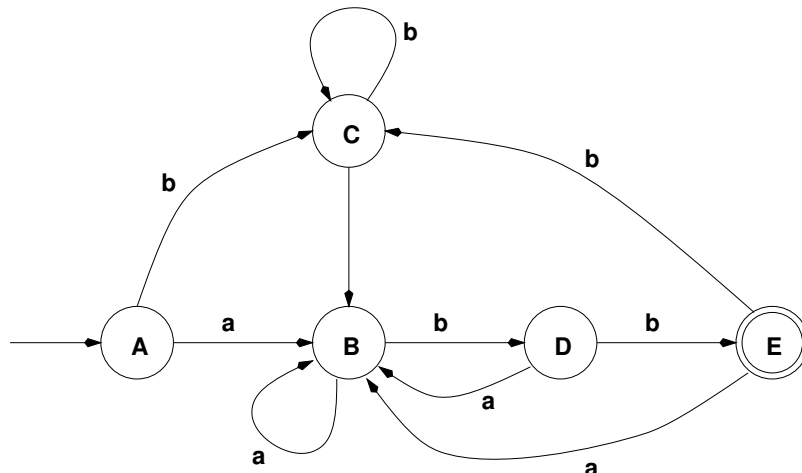
$$DTran[E, a] = B$$

$$T_E^b = \{5\}$$

$$\epsilon.\text{closure}(T_E^b) = C$$

$$DTran[E, b] = C$$

Here is a diagram of the derived *DFA*:



Exercise 3.3.4

Derive a *DFA* from the *NFA* example given a few pages ago (i.e. $(0|1)^*(00|11)$).

Lexical Analysis using Finite Automata

Since the lexical analysis phase of a compiler deals with a regular language fragment, we should expect that *DFA*s provide a *table-driven* approach for a lexical analyser generator such as Flex). Lexer generators such as Flex take as input a specification of the tokens, using regular expressions. This can be automatically converted into a transition table for a *DFA* and the lexer proceeds by simulating a *DFA* for that table. The following table is a scanner for the tokens +, *, :, :=, numerals, identifiers, (,). White space and comments (* ... *) are skipped.

(This is only an example — I'm not suggesting that it would be generated by Flex.)

		cr	+	*	:	=	0	...	9	A	...	Z	()
q ₀	0	0	1	2	3	6	7	...	7	9	...	9	11	15
q ₁	return plussymbol													
q ₂	return multsymbol													
q ₃	4	4	4	4	4	5	4	...	4	4	...	4	4	4
q ₄	return colonsymbol													
q ₅	return assignsymbol													
q ₆	return equalsymbol													
q ₇	8	8	8	8	8	8	7	...	7	8	...	8	8	8
q ₈	return numeral													
q ₉	10	10	10	10	10	10	9	...	9	9	...	9	10	10
q ₁₀	return identifier													
q ₁₁	12	12	12	13	12	12	12	...	12	12	...	12	12	12
q ₁₂	return lparen													
q ₁₃	13	13	13	14	13	13	13	...	13	13	...	13	13	13
q ₁₄	13	13	13	14	13	13	13	...	13	13	...	13	13	0
q ₁₅	return rparen													