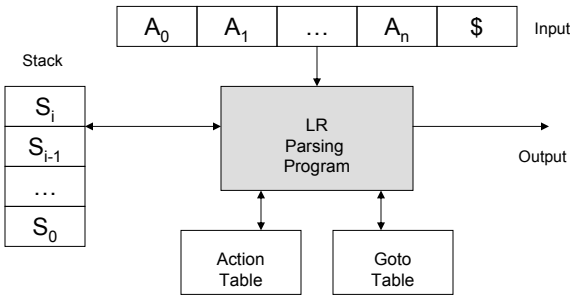
LR(k) Parsing

- LR Parsing is an efficient bottom-up parsing technique that can be used to parse a large class of context-free grammars.
- LR means that the input is scanned from left-to-right building a rightmost derivation in reverse.
- k represents the number of lookahead symbols required to make a parsing decision.
- If k is omitted it is assumed to be 1. Many grammars in compiling fall into the LR(1) class of parsers.
- The main advantage of LR parsers is that they be made to recognise virtually any language for which a context-free grammar exists.
- The main drawback however is that they tend to be very complicated to code by hand, however may generators exist that take a context-free grammar as input and produce a parser for it.

Design (1)

- An LR shift-reduce parser consists of an input, an output, a stack, a parsing program and two parsing tables (action and goto):



Design (2)

- The stack is used to store parsing states.
- The state on the top of the stack combined with the next input token are used by the parsing program to deduce whether it has a handle to reduce or whether it should shift a new state on top of the stack and read the next input token.
- Each entry in the action table contains the four actions for any combination of top stack symbol and next token Si, Aj:
  - Shift,
  - Reduce,
  - Accept,
  - Error.

Design (3)

- The goto table is used whenever the action is a reduction. After a reduction $X \rightarrow \alpha$, the states corresponding to the handle α are popped from the stack to expose the new topmost state s' and the entry for goto[s',X] becomes the new state on top.

Algorithm (1)

- The parser starts with an initial state s_0 on the stack. At some point through a parse the stack will contain $s_0 s_1 s_2 \ldots s_i$.
- Given the next input token a, the parser will proceed as follows:
  - If action[s_i,a] = shift s_{i+1}, the new state is put on top of the stack to become: $s_0 s_1 s_2 \ldots s_i s_{i+1}$, and the new token is read.
  - If action[s_i,a] = reduce $Y \rightarrow X_1 \ldots X_k$, then the k states $s_{i-k+1} \ldots s_i$ are popped off the stack leaving s_{i-k} on top. Now, goto[s_{i-k},Y] is consulted to find a new topmost state s_{i-k+1} which is put on top of the stack to become: $s_0 s_1 s_2 \ldots s_{i-k} s_{i-k+1}$.

Algorithm (2)

- If action [s_i,a] = accept, then the parsing is complete – the whole input tokens have been consumed and reduced to the sentence symbol.
- If action [s_i,a] = error, then a syntax error has been detected.

Parsing Program (1)

```
Set pointer ip to point to the input string
Repeat forever
  Let s = state on top of stack
  Let a = symbol pointed to by ip

  if action[s,a] = shift s' then
    push s' on top of stack
    increment ip to next symbol

  else if action[s,a] = reduce A→B then
    for i = 1 to length(B)
      pop state from stack

    Let s' be the new state on top of stack
    Let s'' = goto[s',A]
    Push s'' on top of stack
```


Parsing Program (2)

```
  else if action[s,a] = accept then
    exit from infinite loop

  else if action[s,a] = error then
    report error
End Repeat
```

LR Parsing Example (1)

- Consider the expression `id * id + id`.
- Grammar:

1) `E → E + T`
2) `E → T`
3) `T → T * F`
4) `T → F`
5) `F → (E)`
6) `F → id`

LR Parsing Example (2)

State	Action Table						Goto Table		
	id	+	*	(	)	EOF	E	T	F
0	S5			S4			1	2	3
1		S6				Acpt			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	R7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

LR Parsing Example (3)

- Notes:
  - s_n means shift state n and read the new token.
  - r_k means reduce by the production k.
  - Encountering blank entries in the tables signify an error.
  - The value goto[s,a] for a TERMINAL a, is found in the action field action[s,a]. The goto table, therefore contains values for goto[s,a] where a is a NON-TERMINAL.

LR Parsing Example (4)

	Stack	Next Token	Action
1	S_0	id	Shift S_5
2	S_0S_5	*	Reduce F → id
3	S_0S_3	*	Reduce T → F
4	S_0S_2	*	Shift S_7
5	$S_0S_2S_7$	id	Shift S_5
6	$S_0S_2S_7S_5$	+	Reduce F → id
7	$S_0S_2S_7S_{10}$	+	Reduce T → T * F
8	S_0S_2	+	Reduce E → T
9	S_0S_1	+	Shift S_6
10	$S_0S_1S_6$	id	Shift S_5
11	$S_0S_1S_6S_5$	EOF	Reduce F → id
12	$S_0S_1S_6S_3$	EOF	Reduce T → F
13	$S_0S_1S_6S_9$	EOF	Reduce E → E + T
14	S_0S_1	EOF	Accept

LR Parsing Example (5)

- At the beginning the parser is in state `0` with `id` as the first input token.
- Therefore `action[0,id]` is taken giving the state s_5 which is pushed on the stack and the new input token is read.
- `'*'` is now the input symbol and the `action[5,*]` is to reduce by rule `6`. One state is popped off (one state on right hand side) exposing state `0`. The value for `goto[0,F]` is `3` meaning that state `3` must be popped onto the stack.
- `'*'` is still the input symbol and `action[3,*]` is to reduce by rule `4`. One state is popped off (one state on right hand side) exposing state `0`. The value for `goto[0,T]` is `2` meaning that state `2` must be popped onto the stack.
- And so on…

Constructing The Parsing Tables

- There are 3 widely used LR parsing techniques:
  - Canonical LR(k) or LR(k) is the most general form of LR parsing methods and is the most powerful. Such parsers usually have many thousands of states for a programming languages and are VERY difficult to hand code.
  - Simple LR(k) or SLR(k) is a variant of LR(k) parsing and usually involves a few hundred states. SLR parsers are the weakest in terms of grammars it can handle but serves as a good starting point to other LR parsing methods.
  - Lookahead LR(k) or LALR(k) is somewhat in the middle in terms of the grammars it can handle. LALR parsers have the same number of states as the equivalent SLR parser but are more difficult to construct. Popular parser generators use this technique to automate parser generation.

FLEX

- FLEX is a popular program that generates lexical analysers.
- FLEX accepts as an input a description of the scanner it has to generate and produces a C source file called `'lexyy.c'` containing the scanning code.
- By convention, FLEX input files have the extension `'.l'`.

The FLEX Input File

- The general format of a FLEX source file is:

```
Definitions
%%
Rules
%%
User Subroutines
```

- The definitions and user subroutines setions are optional as is the second set of `%%` delimiters. Note that the first set of delimiters is required to separate the definitions section from the rules.
- The absolute minimum FLEX program is:

```
%%
```

- Which copies the input program to the output unchanged.

FLEX Definitions (1)

- The definition sections contains declaration of language constructs to simplify the scanner specification.
- These declarations have the form:

```
name definition
```

- Where name is any alpha-numeric word starting with an underscore of a letter.
- For example:
  - `Digit  [0-9]`
  - `Ident  [a-z][a-z0-9]*`
- Where Digit defines a regular expression that recognises a simple one-character digit and Ident recognises a word starting with a letter followed by zero or more occurences of a letter or digit.

FLEX Definitions (2)

- Thus, a subsequent call to:

```
{digit}* "," {digit}+
```

- Is identical to

```
([0-9])* "." ([0-9])+
```

- In the definitions section, any indented text or text enclosed within `%{` and `%}` is copied to the output as it is with the `%{` and `%}` removed.
- The text lines within are usually:
  - Compiler directives such as `#include`'s or `#define`'s.
  - Declarations of variables that are used by other sections of the FLEX input file.

FLEX Rules

- The rules section of a FLEX program contain a series of `pattern action` statements.
- For example:

```
Integer puts("I found an integer")
```

- Would print a message each time an Integer (defined in the definitions section) is found.
- Any indented or '`%{…%}`' code appearing before the first rule in this section is local to the main scanning routine generated by FLEX and is executed each time the routine is called.

FLEX User Subroutines

- The user subroutines section is copied excatly to the output source produced by FLEX.
- When building FLEX scanners that are not interfaced by external programs the C `main` function is defined and programmed here.

Regular Expressions in FLEX (1)

`x`	Matches the character x.
`.`	Any character except the newline.
`[xyz]`	A ‘character class’ – in this case it matches an ‘x’ a ‘y’ or a ‘z’
`[abj-oZ]`	A character class with a range in it – matches an ‘a’, a ‘b’, any letter from ‘j’ to ‘o’ or ‘Z’.
`[^A-Z]`	A negated character class.
`[^A-Z\n]`	A negated character class with an escape character (newline).
`r*`	Zero or more r’s where r is a regular expression.
`r+`	One or more r’s.
`r?`	An optional r (zero or one)
`r{2,5}`	Anything between 2 and 5 r’s.
`r{2,}`	2 or more r’s.

Regular Expressions in FLEX (2)

`r{4}`	Exactly 4 r’s.
`{name}`	The expansion of a name definition.
`"[xyz]"`	The literal ‘[xyz]’
`\X`	If X is an a,b,f,n,r,t or v, the ANSI C interpretation of \X otherwise the literal X – for example \”
`\123`	A character with the octal value of 123.
`\x2a`	A character with the hexadecimal value 2a.
`(r)`	Match an r – parenthesis are used to emphasis precedence.
`rs`	Concatenation of regular expression r and s.
`r|s`	Either an r or an s.
`^r`	An r but only ath the beginning of a line.
`r$`	An r but only at the end of a line – equivalent to r\n.
`<<EOF>>`	The end of file.

Regular Expressions in FLEX (3)

- If there is more than one rule matching the input, the one matching the most text characters is chosen. If the matches have the same length, the file listed first is chosen.
- Once a match is made, the text corresponding the match is put in a special character pointer (C string) variable called `yytext` (`char *yytext`) and its length is in `yyleng`.
  - `Integer printf("I found an integer %s.", yytext);`

Actions (1)

- Each pattern in a rule has a corresponding action that may be any arbitrary C statement.
- The pattern ends at the first non escaped whitespace character. The rest of the line is the action statement.
- If the action is left empty, the token found is discarded.
- The following FLEX program deletes all occurrences of the word 'username' from the input an keeps the rest:

```
%%
"username"
```

- The following program compreses multiple spaces and tabs into one space charater and removes trailing spaces too:

```
[ \t]+   putchar(' ');
[ \t]+$  /* ignore trailing blanks */
```

Actions (2)

- An action consisting of only the vertical bar '|' means "the same action as the one for the next rule. If the action contains a '{', then the action spans until the next balancing '}'. For example:

```
IF  |
if  {
                puts{"Keyword IF found.");
                return IFWORD;
        }
```

- Actions contain arbitrary C code, including return statements to return values to whatever external routine called `yylex()` – the token parser.
- Each time `yylex()` is called, it continues processing from where it last left off until it reaches an `EOF` or meets a return statement.
- Once `yylex()` reaches the end of file, however, any subsequent call to `yylex()` will immediately return unless `yyrestart()` is called.
- Note any actions are not allowed to modify `yytext` or `yyleng`.

Special Routines and Directives (1)

- `ECHO`: copies `yytext` to the scanners output.
- `yymore()`: tells the scanner that the next time it matches a rule, the corresponding token should be appended onto the current value of `yytext` rather than replacing it. For example:

```
%%
a-      ECHO; yymore();
b       ECHO;
```

- The first '`a-`' is matched and echoed to the output. Then '`b`' is matched by the previous '`a-`' is still in `yytext` so the echo for '`b`' will include the previous '`a-`'s

Special Routines and Directives (2)

- `yyless(n)`: redirects all but the first `n` characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match. `yytext` and `yyleng` are adjusted appropriately. Note that a call to `yyless(0)` will cause the entire input string to be scanned again and would result in an infinite loop unless care is taken.
- unput(c): puts the character '`c`' back onto the input stream which will then be the next character scanned. For example the following action will take the current token and cause it to be rescanned enclosed in parenthesis:

```
{
   int i;
   unput(')');
   for (i = yyleng -1; i > 0; --i)
     unput( yytext[i] );
   unput('{');
}
```

- Note that all characters are pushed to the BEGINNING of the input string so the original characters are put in reverse.

Special Routines and Directives (3)

- `input()`: reads the next character from the input stream. (or `yyinput()` if used with C++)
- `yyterminate()`: can be used instead of a return statement. It aborts the action returning 0. subsequent calls to `yylex()` immediately return unless `yyrestart()` is called. (usually called on encountering the EOF)
- `yyrestart()`: tells FLEX to start scanning from a new (maybe the same) input file. Takes a single `file *` pointer.

The Generated Scanner (1)

- Whenever `yylex()` is called, it scans tokens from a global input file denoted by `yyin` which by default points to standard input unless specified using the C function `fopen` (file open).
- For example, to open `example.txt` for reading in text mode:

```
yyin = fopen("example.txt","r");
```

- `yylex()` continues reading from `yyin` until it reaches EOF. In this case the function will return immediately unless yyrestart() is called and yyin is set to point to a new file.

The Generated Scanner (2)

- Likewise, the scanner produces output to `yyout` which, again, by default, points to standard output. As with `yyin`, `yyout` can be changed by assigning it another `FILE` pointer:
  - `yyout = fopen("output.txt", "w");`

Interfacing with Parser Generators

- One of the main uses of FLEX is interfacing it with an external parser generator like Bison. Bison parsers expect to call a function called `yylex()` to find the next input token.
- `yylex()`, is expected to return the type of the token found (implemented as a constant, maybe) and putting any associated value in `yylval`.
- To use Bison in association with FLEX, it is called with the `-d` option to instruct is to generate an header file containing all the token definitions. This header is then used in FLEX. To include the header:

```
%{
#include "generated_header.h"
%}
%%
[0-9]+    yylval = atoi(yytext); return NUM;
```