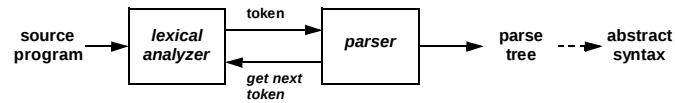


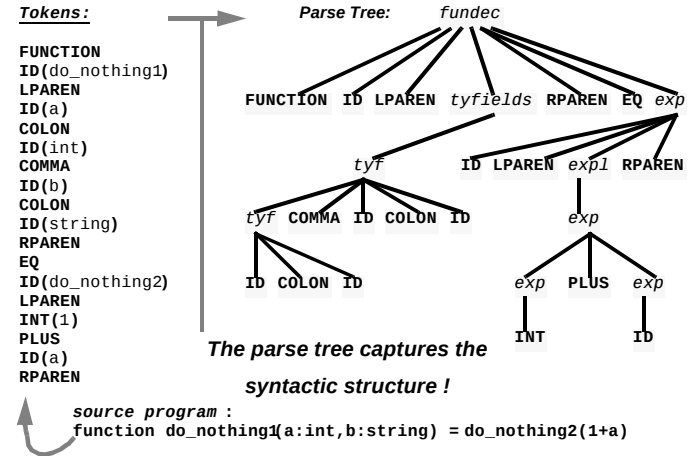
Syntax Analysis

- Convert the list of **tokens** into a **parse tree** ("hierarchical" analysis)



- The **syntactic structure** is specified using **context-free grammars**
[in lexical analysis, the lexical structure is specified using regular expressions]
- A **parse tree** (also called concrete syntax) is a graphic representation of a derivation that shows the hierarchical structure of the language
- Other secondary tasks: syntax error detection and recovery

Tokens ---> Parse Tree



Main Problems

- How to specify the **syntactic structure** of a programming language ?
by using **Context-Free Grammars (CFG) !**
- How to **parse** ? i.e., given a CFG and a stream of tokens, how to build its parse tree ?
 1. **bottom-up** parsing
 2. **top-down** parsing
- How to make sure that the parser generates a **unique** parse tree ? (the **ambiguity** problem)
- Given a CFG, how to build its **parser** quickly ?
using YACC ---- the parser generator
- How to detect, report, and recover syntax errors ?

Grammars

- A **grammar** is a precise, understandable specification of programming language syntax (but not semantics !)
- Grammar** is normally specified using **Backus-Naur Form (BNF)** ---
 - a set of **rewriting rules** (also called **productions**)


```

stmt -> if expr then stmt else stmt
expr -> expr + expr | expr * expr
      | ( expr ) | id
          
```
 - a set of **non-terminals** and a set of **terminals**

```

non-terminals ---- stmt, expr
terminals ----    if, then, else, +, *, (, ), id
          
```
 - lists are specified using recursion


```

stmt -> begin stmt-list end
stmt-list -> stmt | stmt ; stmt-list
          
```

Context-Free Grammars (CFG)

- A **context-free grammar** is defined by the following (T, N, P, S) :

T is vocabulary of **terminals**,

N is set of **non-terminals**,

P is set of **productions** (rewriting rules), and

S is the **start symbol** (also belong to N).

- Example: a context-free grammar $G=(T, N, P, S)$

$T = \{ +, *, (,), \text{id} \},$

$N = \{ E \},$

$P = \{ E \rightarrow E + E, E \rightarrow E * E, E \rightarrow (E), E \rightarrow \text{id} \},$

$S = E$

- Written in BNF: $E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$
- All regular expressions can also be described using CFG

Context-Free Languages (CFL)

- Each context-free grammar $G=(T, N, P, S)$ defines a **context-free language** $L = L(G)$
- The CFL $L(G)$ contains all sentences of **terminal** symbols (from T) --- **derived** by repeated application of **productions in P** , beginning at the **start symbol S** .
- Example the above CFG denotes the language $L =$

$$L(\{ +, *, (,), \text{id} \}, \{ E \}, \{ E \rightarrow E + E, E \rightarrow E * E, E \rightarrow (E), E \rightarrow \text{id} \}, E)$$

it contains sentences such as $\text{id}+\text{id}, \text{id}+(\text{id}*\text{id}), (\text{id}), \text{id}*\text{id}*\text{id}*\text{id}, \dots$

- Every regular language must also be a CFG ! (the reverse is not true)

Derivations

- derivation** is repeated application of productions to yield a sentence from the start symbol:

$E \Rightarrow E * E$	--- "E derives $E * E$ "
$\Rightarrow \text{id} * E$	--- "E derives id "
$\Rightarrow \text{id} * (E)$	--- "E derives (E) "
$\Rightarrow \text{id} * (E + E)$	
$\Rightarrow \text{id} * (\text{id} + E)$	
$\Rightarrow \text{id} * (\text{id} + \text{id})$	

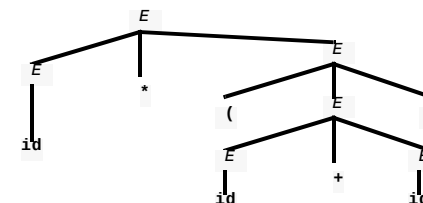
Summary: $E \Rightarrow^* \text{id} * (\text{id} + \text{id})$
 $\Rightarrow^*$: derives in 0 or more steps

- the intermediate forms always contain some non-terminal symbols
- leftmost derivation** : at each step, leftmost non-terminal is replaced;
e.g. $E \Rightarrow E * E \Rightarrow \text{id} * E \Rightarrow \text{id} * \text{id}$
- rightmost derivation** : at each step, rightmost non-terminal is replaced;
e.g. $E \Rightarrow E * E \Rightarrow E * \text{id} \Rightarrow \text{id} * \text{id}$

Parse Tree

- A parse tree is a graphical representation of a derivation that shows hierarchical structure of the language, independent of derivation order.
- Parse trees have leaves labeled with **terminals**; interior nodes labeled with **non-terminals**.

example: $E \Rightarrow^* \text{id} * (\text{id} + \text{id})$



- Every parse tree has unique leftmost (or rightmost) derivation !

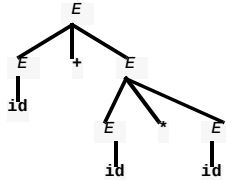
Ambiguity

- A language is **ambiguous** if a sentence has more than one parse tree, i.e., more than one leftmost (or rightmost) derivation

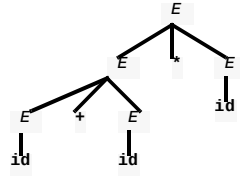
example: `id + id * id`

another leftmost derivation:

```
a) E => E + E => id + E
    => id + E * E
    => id + id * E
    => id + id * id
```



b) $E \Rightarrow E * E \Rightarrow E + E * E$
 $\Rightarrow \text{id} + E * E$
 $\Rightarrow \text{id} + \text{id} * E$
 $\Rightarrow \text{id} + \text{id} * \text{id}$



Resolving Ambiguity

- Solution #1 : using “**disambiguating rules**” such as **precedence** ...

e.g. let * has higher priority over + (favor derivation (a))

- *Solution #2 : rewriting grammar to be **unambiguous** !*

```

"dangling-else"    stmt -> if expr then stmt
                    | if expr then stmt else stmt
                    | .....

```

How to parse the following ?

if E_1 then if E_2 then S_1 else S_2

How to rewrite ?

Main Idea: build “precedence” into grammar with extra non-terminals !

Resolving Ambiguity (cont'd)

- solution: define “matched” and “unmatched” statements

$$stmt \rightarrow m-stmt \mid um-stmt$$

```
m-stmt -> if expr then m-stmt else m-stmt
        | .....
```

```
um-stmt -> if expr then stmt
           | if expr then m-stmt else um-stmt
```

Now how to parse the following ?

if E_1 then if E_2 then S_1 else S_2

Resolving Ambiguity (cont'd)

- *Another ambiguous grammar*

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid \text{id}$$

usual precedence:	highest	- (unary minus)
		* /
	lowest	+ -

- Build grammar from highest ---> lowest precedence

```

element -> ( expr ) | id
primary -> - primary | element
term    -> term * primary | term / primary | primary
expr    -> expr + term | expr - term | term

```

```
try the leftmost derivation for      - id + id * id
expr => expr + term => term + term => primary + term
=> - primary + term => - element + term => - id + term
=> - id + term * primary => ... => - id + id * id
```

Other Grammar Transformations

- **Elimination of Left Recursion** (useful for top-down parsing only)

replace productions of the form
with

$$\begin{aligned} A &\rightarrow A x \mid y \\ A' &\rightarrow y A' \\ A' &\rightarrow x A' \mid ? \end{aligned}$$

(yields different parse trees but same language)

see Appel pp 51-52 for the general algorithm

- **Left Factoring** --- find out the common prefixes (see Appel pp 53)

change the production
to

$$\begin{aligned} A &\rightarrow x y \mid x z \\ A' &\rightarrow y \mid z \end{aligned}$$

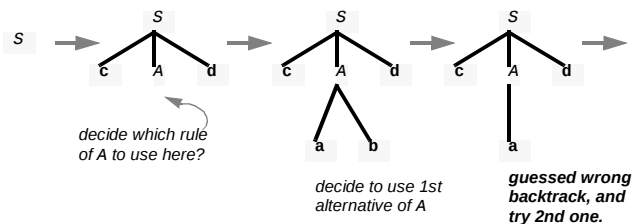
Parsing

- **parser** : a program that, given a sentence, reconstructs a derivation for that sentence ---- if done successfully, it "recognize" the sentence
- all parsers read their input left-to-right, but construct parse tree differently.
- **bottom-up parsers** --- construct the tree from leaves to root
shift-reduce, LR, SLR, LALR, operator precedence
- **top-down parsers** --- construct the tree from root to leaves
recursive descent, predictive parsing, LL(1)
- **parser generator** --- given BNF for grammar, produce parser
YACC --- a LALR(1) parser generator

Top-Down Parsing

- Construct parse tree by starting at the start symbol and "guessing" at derivation step. It often uses next input symbol to guide "guessing".

example: $S \rightarrow c A d$ input symbols: **cad**
 $A \rightarrow ab \mid a$



- Main algorithms : recursive descent, predictive parsing (details will be taught in the future)

Bottom-Up Parsing

- Construct parse tree "bottom-up" --- from leaves to the root
- Bottom-up parsing always constructs **right-most derivation**
- Important parsing algorithms: **shift-reduce**, **LR parsing**, ...
- **shift-reduce parsing** : given input string w , "reduce" it to the start symbol !

Main idea: look for substrings that match r.h.s of a production
Example:

Grammar		sentential form	reduction
		abbcde	
	right-most derivation in reverse	aAbcde	$A \rightarrow b$
$S \rightarrow aAcBe$		aAcde	$A \rightarrow Ab$
$A \rightarrow Ab \mid b$		aAcBe	$B \rightarrow d$
$B \rightarrow d$		S	$S \rightarrow aAcBe$

Handles

- **Handles** are substrings that can be replaced by l.h.s. of productions to lead to the start symbol.
- Not all possible replacements are handles --- some may not lead to the start symbol ... $abbcde \rightarrow aAbcde \rightarrow aAacde \rightarrow \text{stuck!}$
this b is not a handle !
- Definition : if γ can be derived from S via right-most derivation, then γ is called a **right-sentential form** of the grammar G (with S as the start symbol). Similar definition for **left-sentential form**.
- **handle** of a right-sentential form γ is $A \rightarrow \gamma$ if

$$S \Rightarrow_{rm}^* \gamma A \gamma' \Rightarrow_{rm} \gamma \gamma'' \gamma'''$$

and γ contains only terminals. E.g., $A \rightarrow Ab$ in $aAbcde$

Handle Pruning

- **Main idea:** start with terminal string w and "prune" handles by replacing them with l.h.s. of productions until we reach S :

$$S \Rightarrow_{rm} \gamma_1 \gamma_2 \Rightarrow_{rm} \gamma_3 \Rightarrow_{rm} \gamma_4 \Rightarrow_{rm} \gamma_5 \Rightarrow_{rm} \gamma_6 \Rightarrow_{rm} \gamma_7$$

(i.e., construct the rightmost derivation in reverse)

- **Example:** $E \rightarrow E + E \mid E * E \mid (E) \mid a \mid b \mid c$

right-sentential form	handle	reducing production
$a + b * c$	a	$E \rightarrow a$
$\bar{E} + b * c$	b	$E \rightarrow b$
$E + \bar{E} * c$	c	$E \rightarrow c$
$E + E * \bar{E}$	$E * E$	$E \rightarrow E * E$
$\bar{E} + \bar{E}$	$E + E$	$E \rightarrow E + E$
E		

Key of Bottom-Up Parsing: Identifying Handles

Shift-Reduce Parsing

- Using a stack, **shift** input symbols onto the stack until a handle is found; **reduce** handle by replacing grammar symbols by l.h.s. of productions; **accept** for successful completion of parsing; **error** for syntax errors.

- **Example:** $E \rightarrow E + E \mid E * E \mid (E) \mid a \mid b \mid c$

stack	input	action
\$	$a+b*c\$$	shift
\$a	$+b*c\$$	reduce: $E \rightarrow a$
\$E	$+b*c\$$	shift
\$E+	$b*c\$$	shift
\$E+b	$*c\$$	reduce: $E \rightarrow b$
\$E+E	$*c\$$	shift (possible SR conflict)
\$E+E*	$c\$$	shift
\$E+E*c	\$	reduce: $E \rightarrow c$
\$E+E*c	\$	reduce: $E \rightarrow E*c$
\$E+E	\$	reduce: $E \rightarrow E+E$
\$E	\$	accept

handle is always at the top !

Conflicts

- **ambiguous grammars** lead to **parsing conflicts**; conflicts can be fixed by **rewriting** the grammar, or making a **decision during parsing**
- **shift / reduce (SR) conflicts** : choose between reduce and shift actions

$$S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \dots$$

stack	input	action
\$if E then S	else ...\$	reduce or shift?

- **reduce/reduce (RR) conflicts** : choose between two reductions

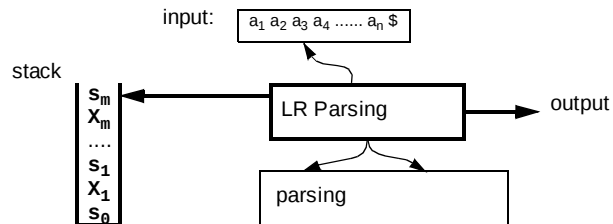
stmt $\rightarrow$ id (param) --- procedure call $a(i)$
 param $\rightarrow$ id
 E $\rightarrow$ id (E) | id --- array subscript $a(i)$

stack	input	action
\$id(id	) ...\$	id reduce to E or param ?

LR Parsing

today's most commonly-used parsing techniques !

- **LR(k) parsing** : the "L" is for left-to-right scanning of the input; the "R" for constructing a rightmost derivation in reverse, and the "k" for the number of input symbols of lookahead used in making parsing decisions. ($k=1$)
- **LR parser components**: input, stack (strings of grammar symbols and states), driver routine, parsing tables.



LR Parsing (cont'd)

- A sequence of new **state** symbols $s_0, s_1, s_2, \dots, s_m$ ----- each state summarize the information contained in the stack below it.
- **Parsing configurations**: (**stack, remaining input**) written as
 $(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} a_{i+2} \dots a_n \$)$
 next "move" is determined by s_m and a_i
- **Parsing tables**: **ACTION[s, a]** and **GOTO[s, X]**

Table A **ACTION[s, a]** --- s : state, a : terminal

its entries (1) shift s_k (2) reduce $A \rightarrow ?$
 (3) accept (4) error

Table G **GOTO[s, X]** --- s : state, X : non-terminal
 its entries are states

LR Parsing Driver Routine

Given the configuration:

$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} a_{i+2} \dots a_n \$)$

- (1) If **ACTION[s_m, a_i]** is "shift s", enter config

$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i s, a_{i+1} a_{i+2} \dots a_n \$)$

- (2) If **ACTION[s_m, a_i]** is "reduce A->", enter config

$(s_0 X_1 s_1 X_2 s_2 \dots X_{m-r} s_{m-r} A s, a_i a_{i+1} a_{i+2} \dots a_n \$)$

where $r = |A|$, and $s = \text{GOTO}[s_{m-r}, A]$
 (here ? should be $X_{m-r+1} X_{m-r+2} \dots X_m$)

- (3) If **ACTION[s_m, a_i]** is "accept", parsing completes

- (4) If **ACTION[s_m, a_i]** is "error", attempts error recovery.

Example: LR Parsing

- Grammar :

- | | |
|--------------------------------------|---------------------------|
| 1. $S \rightarrow S ; S$ | 6. $E \rightarrow E + E$ |
| 2. $S \rightarrow \text{id} := E$ | 7. $E \rightarrow (S, E)$ |
| 3. $S \rightarrow \text{print } (L)$ | 8. $L \rightarrow E$ |
| 4. $E \rightarrow \text{id}$ | 9. $L \rightarrow L, E$ |
| 5. $E \rightarrow \text{num}$ | |

- Tables :

sn + shift and put state n on the stack
 gn + go to state n
 rk + reduce by rule k
 a + accept and parsing completes
 - + error

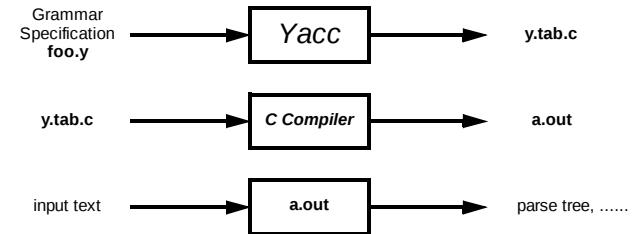
- Details see figure 3.18 and 3.19 in Appel pp.56-57

Summary: LR Parsing

- **LR Parsing is doing reverse right-most derivation !!!**
- If a grammar is ambiguous, some entries in its parsing table (**ACTION**) contain multiple actions : "shift-reduce" or "reduce-reduce" conflicts.
- **Two ways to resolve conflicts** ---- (1) rewrite the grammar to be unambiguous (2) making a decision in the parsing table (retaining only one action!)
- **LR(k) parsing:** parsing moves determined by state and next k input symbols; $k = 0, 1$ are most common.
- A grammar is an **LR(k) grammar**, if each entry in its LR(k)-parsing table is uniquely defined.
- **How to build LR parsing table?** ---- three famous varieties: SLR, LR(1), LALR(1) (detailed algorithms will be taught later !)

Yacc

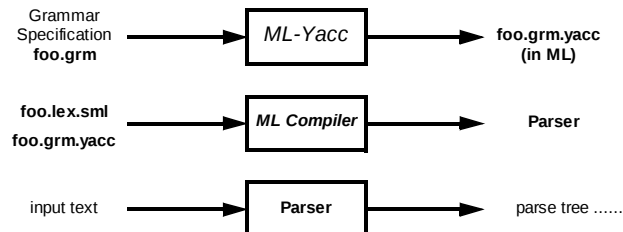
- **Yacc** is a program generator ----- it takes **grammar specification** as input, and produces an **LALR(1) parser** written in C.



- **Implementation of Yacc:**
Construct the LALR(1) parser table from the grammar specification

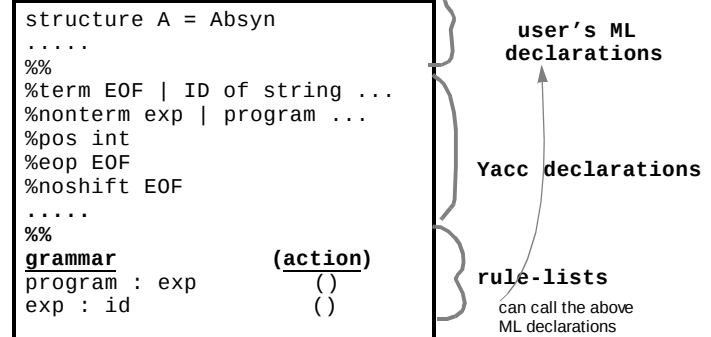
ML-Yacc

- **ML-Yacc** is like **Yacc** ----- it takes **grammar specification** as input, and produces a **LALR(1) parser** written in Standard ML.



- **Implementation of ML-Yacc** is similar to implementation of **Yacc**

ML-Yacc Specification



- **grammar** is specified as BNF production rules; **action** is a piece of ML program; when a grammar production rule is **reduced** during the parsing process, the corresponding **action** is executed.

ML-Yacc Rules

- *BNF production* $A \rightarrow ? \mid ? \mid \dots \mid ?$ is written as

```

A : ?           (action for A -> ?)
  | ?           (action for A -> ?)
  | ...
  | ?           (action for A -> ?)

```

- The **start symbol** is l.h.s. of the first production or symbol S in the Yacc declaration
`%start S`
- The **terminals or tokens** are defined by the Yacc declaration `%term`
`%term ID of string | NUM of int | PLUS | EOF | ...`
- The **non-terminals** are defined by the Yacc declaration `%nonterm`
`%nonterm EXP of int | START of int option`

Example: calc.grm

```

fun lookup "bogus" = 10000 | lookup s = 0

%%
%eop EOF SEMI
%pos int
%left SUB PLUS
%left TIMES DIV

%term ID of string | NUM of int | PLUS | TIMES | PRINT |
      SEMI | EOF | DIV | SUB
%nonterm EXP of int | START of int
%verbose
%name Calc
%%
START : PRINT EXP          (print EXP; print "\n"; EXP)
      | EXP                (EXP)

EXP : NUM                  (NUM)
    | ID                  (lookup ID)
    | EXP PLUS EXP        (EXP1+EXP2)
    | EXP TIMES EXP       (EXP1*EXP2)
    | EXP DIV EXP         (EXP1 div EXP2)
    | EXP SUB EXP         (EXP1-EXP2)

```

Yacc : Conflicts

- Yacc uses the LR parsing (i.e. LALR); if the grammar is ambiguous, the resulting parser table ACTION will contain **shift-reduce** or **reduce-reduce** conflicts.
- In Yacc, you resolve conflicts by (1) rewriting the grammar to be unambiguous (2) declaring precedence and associativity for terminals and rules.
- Consider the following grammar and input `ID PLUS ID PLUS ID`

```

E : E PLUS E      ( )
  | E TIMES E     ( )
  | ID            ( )

```

we can specify **TIMES** has higher precedence than **PLUS**; and also assume both **TIMES** and **PLUS** are left associative.
 (also read the examples on Appel pp73-74)

Precedence and Associativity

- To resolve conflicts in Yacc, you can define **precedence** and **associativity** for each **terminal**. The precedence of each **grammar rule** is the precedence of its **rightmost terminal** in r.h.s of the rule.

- On **shift / reduce** conflict:

```

if input terminal prec. > rule prec. then shift
if input terminal prec. < rule prec. then reduce
if input terminal prec. == rule prec. then {
    if terminal assoc. == left then reduce
    if terminal assoc. == right then shift
    if terminal assoc. == none then report error
}

```

if the input terminal or the rule has no prec. then shift & report error

- On **reduce / reduce** conflict: report error & rule listed first is chosen

Defining Prec. and Assoc.

- Defining precedence and associativity for **terminals**

```
%left OR
%left AND
%noassoc EQ NEQ GT LT GE LE
%left PLUS MINUS
%left TIMES DIV
```

lowest prec.
↓
highest prec.

- Defining precedence for **rules** using %prec

```
%%
.....
%left PLUS MINUS
%left TIMES DIV
%left UNARYMINUS
%%
Exp : Exp MINUS Exp      ( )
    | Exp TIMES exp      ( )
    | MINUS exp           ( )
    .....
%prec UNARYMINUS
```

Must define UNARYMINUS

Assuming unary minus has higher precedence than

Only specifies the prec. of this rule == prec. of UNARY-

Parser Description (.desc file)

- The Yacc declaration %verbose will produce a verbose description of the generated parser (i.e., the ".desc" file)

- A summary of errors found while generating the parser
- A detailed description of all errors
- The parsing engine --- describing the states and the parser table (see example 3.1 on pp15-18 in Appel's book)

```
state 0:

program : . exp          current states (characterized by grammar rules)
ID      shift 13          table ACTION
INT     shift 12
STRING  shift 11
LPAREN  shift 10
MINUS   shift 9
IF      shift 8

program goto 135          table GOTO
exp     goto 2
lvalue  goto 1

.      error
```

Tiger.Lex File “mumbo-jumbo”

You have to modify your “tiger.lex” file in assignment 2 by adding the following --- in order to generate the functor “TigerLexFun”

```
type svalue = Tokens.svalue
type pos = int
type ('a, 'b) token = ('a, 'b) Tokens.token
type lexresult = (svalue,pos) token

.....
.....

%%
%header (functor TigerLexFun(structure Tokens : Tiger_TOKENS));
.....
.....

%%
.....
```

Connecting Yacc and Lex

```
signature PARSE = sig val parse : string -> unit end

structure Parse : PARSE =
struct
  structure TigerLrVals = TigerLrValsFun(structure Token =
                                          LrParser.Token)

  structure Lex = ToyLexFun(structure Tokens = TigerLrVals.Tokens)
  structure TigerP =
    Join(structure ParserData = TigerLrVals.ParserData
          structure Lex=Lex
          structure LrParser = LrParser)

  fun parse filename =
    let val _ = (ErrorMsg.reset(); ErrorMsg.fileName := filename)
    val file = open_in filename
    fun parseerror(s,p1,p2) = ErrorMsg.error p1 s
    val lexer = LrParser.Stream.streamify
                  (Lex.makeLexer (fn _ => TextIO.input file))
    val (absyn, _) = TigerP.parse(30,lexer,parseerror,())
    in close_in file;
    absyn
    end handle LrParser.ParseError => raise ErrorMsg.Error
end
```