

# Programming Language Implementation VIII

In this lecture we will finish looking at **syntax analysis**. i.e. **parsing**.

- **Generic bottom-up parsing.**
- **Parser Generators (ML-YACC).**

Material for ML-YACC can be found in Appel, Chapter 3 and at <http://www.cs.princeton.edu/~appel/modern/ml/>.

## Generic Bottom-Up Parsing

There are several generic methods for parsing with context free grammars. One method based on **dynamic programming** is due to Cocke & Younger and Kasami (Often called the **CKY algorithm**).

The first step is to transform the grammar into **Chomsky normal form (CNF)**. This requires the grammar to be

- $\epsilon$ -free
- and each (non  $\epsilon$  production) is of the form  $A \rightarrow a$  or  $A \rightarrow BC$  where  $a$  is a terminal and  $A, B, C$  are non-terminals.

Consider the input string  $w = a_1a_2 \cdots a_n$ .

We construct a  $n \times n$  table  $T$  so that

$$T[i, j] = \{A | A \Rightarrow^* a_i a_{i+1} \cdots a_j\}.$$

- Initially for  $i = 1, \dots, n$ ,  $T[i, i]$  is  $\{A | A \Rightarrow a_i\}$ .  
From now on we are only concerned with productions of the form  $A \rightarrow BC$ .
- Then we iteratively compute column  $j$  in terms of lower columns. For each production  $A \rightarrow BC$ , if for some  $i, k < j$  we have  $B \in T[i, k]$  and  $C \in T[k + 1, j]$  then add  $A$  to  $T[i, j]$ .
- $w$  is in the language iff  $S \in T[1, n]$ .

## Converting to Chomsky Normal Form

To convert a grammar into chomsky normal form we need to perform two steps:

- make the grammar  $\epsilon$ -free
- convert all productions into one of the forms  $A \rightarrow a$  or  $A \rightarrow BC$

Reminder: to make a grammar  $\epsilon$ -free

- determine all non-terminals  $X$  for which  $X \Rightarrow^* \epsilon$ . We call these non-terminals *nullable*.
- replace each production  $p$  of the form  $A \rightarrow B_1, B_2, \dots, B_n$  by a set of productions that is composed of copies of  $p$  with each possible combination of nullable non-terminals removed on the LHS.

## Converting Productions to $A \rightarrow a$ or $A \rightarrow BC$

This can be simply done by adding new non-terminals.

Repeat the following process exhaustively:

Each production  $P$  that is not in the required form must be of the either of the forms  $A \rightarrow a\alpha$  or  $A \rightarrow B\alpha$  where  $a$  is a terminal,  $B$  a non-terminal and  $\alpha$  a string over terminal and non-terminals.

- If  $P$  is of the form  $A \rightarrow a\alpha$  replace  $P$  by  $A \rightarrow XY$ , where  $X, Y$  are fresh non-terminals. Add the production  $X \rightarrow a$ .
- If  $P$  is of the form  $A \rightarrow B\alpha$  replace  $P$  by  $A \rightarrow BY$ , where  $Y$  is a fresh non-terminal.
- Add the production  $Y \rightarrow \alpha$ .

We have ignored the case  $A \rightarrow B$ . How is this handled?

## Simple Implementation of CYK

```
for j := 1 to n do
  T[j,j] := {A|A ⇒ ai}
  for i := j-1 to 1 do
    for k := i to j do
      if B ∈ T[i, k] and C ∈ T[k + 1, j] and
        A → BC is a production in G then
        add A to T[i, j]
      end
    end
  end
end
accept if S ∈ T[1, n].
end.
```

## Example of CYK Parsing

Consider the grammar

$$exp \rightarrow exp + exp \mid \mathbf{int}$$

Give a Chomsky normal form grammar for it:

Now parse the string:  $\mathbf{int}_1 + \mathbf{int}_2 + \mathbf{int}_3$ .

## Parser Generators – YACC

Creating LR parsers by hand is quite tedious and error prone, because the table construction is too complicated. Instead we can use a parser generator. The best known parser generator is the unix program `yacc` (“Yet Another Compiler Compiler”).

- **bison**, is the yacc implementation as part of the Open Software Foundation’s GNU system. It interfaces with **FLEX** and “C” code.
- **ML-YACC** is an ML implementation of YACC (with slight modifications). It interfaces with ML code and ML-Lex.

A parser generator takes

- an attributed LR grammar
- additional code annotations for semantic actions
- (optional) precedence rules to resolve shift/reduce conflicts
- (optional) additional error handling code

It produces a complete LR parser for the given language.

## ML-YACC specification

An ML-YACC specification has the form

```
{user declarations}  
%%  
{ML-Yacc declarations}  
%%  
{rules}
```

As in ML-Lex, the “user declarations” contain arbitrary ML code that can later be used in the semantic actions of the grammar.

The “ML-Yacc declarations” contain in particular the declaration of terminal and non-terminals symbols and types. Additional precedence declarations and error handling declarations also go into this part.

Finally, the “rules” form the core of the parser specification. Each rule is a production of the form

```
NT : LHS1 ... LHSn {semantic action code}
```

where **NT** is the RHS non-terminal, **LHS<sub>i</sub>** are the terminals and non-terminals on the left-hand side and the semantic action is given as ML-Code. The code for this action will be executed immediately when this production is used for a reduction.

Alternative right-hand sides are separated by bars.

For example, our usual expression production is

```
exp: exp PLUS exp | exp TIMES exp
```

## Semantic Actions

Each non-terminal symbol can have a value associated with it. This value is set by the semantic action of the production, ie. the value to which the semantic action evaluates is assigned to the value of this non-terminal. Of course, the types must be in agreement.

Whether a grammar symbol has a value is derived from its declaration in the “ML-YACC declarations” section, for example

```
%nonterm EXP of int
```

declares the non-terminal *EXP* to have a value of type integer.

**Note that you can only use non-polymorphic types.**

This means, of course, that every symbol can only have a single value and that *multiple attributes for a symbol must be simulated with a tuple or other datatype.*

If a symbol has no value, its actions are still evaluated for **side-effects** and the returned values are ignored.

The values of the non-terminals on the left-hand side are accessible as *symbol $n$*  where  $n$  is the occurrence count of the symbol on the LHS. For example, **exp1** is the value for the first *exp* symbol on the LHS.

## Synthesized versus Inherited Attributes

Recall that a *synthesized attribute* is an assignment to an attribute of the LHS symbol (computed from RHS attributes).

An *inherited attribute* is an assignment to one of the RHS symbols (computed from LHS attributes and other RHS attributes).

ML-Yacc only uses a synthesized attribute schema, as for example in

```
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)
```

This means that *inherited attributes have to be simulated* by returning a function instead of a value from the derived symbol.

This function takes the value of the intended inherited attribute and computed the value for the synthesized attribute.

## Converting Inherited Attributes

Consider this fragment of our expression attribute grammar:

$$\begin{aligned} term &\rightarrow factor\ term' \\ &\quad term'.v1 := factor.v; term.v := term'.v; \\ term' &\rightarrow *term \\ &\quad term'.v := term'.v1 * term.v; \\ term' &\rightarrow \epsilon \\ &\quad term'.v := term'.v1; \\ factor &\rightarrow \mathbf{int} \\ &\quad factor.v := int.v; \end{aligned}$$

To simulate its function with synthesized (function) attributes only convert it to:

$$\begin{aligned} term &\rightarrow factor\ term' \\ &\quad term.val := term'.fn(factor.val); \\ term' &\rightarrow *term \\ &\quad term'.fn := (fn\ t => t * term.val); \\ term' &\rightarrow \epsilon \\ &\quad term'.v := fn\ t => t; \\ factor &\rightarrow \mathbf{int} \\ &\quad factor.val := int.val; \end{aligned}$$

where  $term'.fn$  is a synthesized function attribute.

## A first Example

The following code is a complete parser (and evaluator) for simple arithmetic expressions. It parses appropriately lex'ed arithmetic expressions and returns their value. If an expression is prefixed with a `PRINT` token (statement) its value will additionally be printed to the console.

```
fun lookup "x" = 5
  | lookup "y" = 6
  | lookup "z" = 7
  | lookup s = 0

%%

%eop EOF SEMI
%pos int

%left SUB PLUS
%left TIMES DIV
%right CARAT

%term ID of string | NUM of int | PLUS | TIMES | PRINT |
      SEMI | EOF | CARAT | DIV | SUB
%nonterm EXP of int | START of int option

%name Calc

%subst PRINT for ID
%prefer PLUS TIMES DIV SUB
```

```

%keyword PRINT SEMI

%noshift EOF
%value ID ("bogus")
%nodefault
%verbose
%%

START : PRINT EXP (print (Int.toString EXP);
                  print "\n";
                  TextIO.flushOut TextIO.stdout;
        SOME EXP)
      | EXP (SOME EXP)
      | (NONE)
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)
      | EXP CARAT EXP (let fun e (m,0) = 1
                        | e (m,1) = m*e(m,1-1)
                        in e (EXP1,EXP2)
                        end)

```

You will find this code in the ML-YACC example directory.

Remember that *SOME* and *NONE* are the constructors of the *option* datatype.

## Declarations in Example (cont.)

The declarations of terminals and non-terminals and the grammar rules should be easy enough to understand. What are the remaining declarations.

First note that we have simply defined a *lookup* function to implement a variable valuation, ie. a trivial *symbol table*. In reality this would, of course, not be hard-coded in the grammar.

**name** gives the parser a name that we need later to couple it with the lexer.

**%eop EOF SEMI** declares that the tokens **EOF** and **SEMI** (semicolon) may follow the start symbol, i.e. end the parse. ML-YACC cannot recognise the end-of-input, so the lexer must insert an appropriate token.

**%noshift EOF** declares that **EOF** may not be shifted. This may seem obvious but must be declared so that the parser does not attempt to shift it in cases of error recovery.

**%pos** defines the (non-polymorphic) ML type for the position attribute of tokens. The position values for the tokens can be used for error messages and can be accessed as *symbolnamen + 1left* and *symbolnamen + 1right*. The position values for terminals must be set by the lexer (se below).

**%left,%right** declare the associativity and order of precedence of symbols in increasing order. Symbols in the same declaration have the same precedence. This is used for conflict resolution (see below).

`%subst`, `%prefer`, `%keyword`, `%value` declarations are used in error recovery and will be explained later.

`%nodefault`, `%verbose` are used for debugging purposes only.



The parsing table for this grammar produces a conflict. If ML-YACC is used with the `%verbose` declaration we will can inspect the state table (produced as output).

We find that in one of the states we have **shift/reduce conflict** with *shift(ELSE)* or reducing with the first “IF” rule.

ML-YACC will by default choose *shift(ELSE)*.

As a consequence the *ELSE* will be bound to the innermost open *THEN*. If this is the intended effect the conflict is acceptable.

## Precedence in ML-YACC

In general the grammar should be re-written such that conflicts are avoided.

In ML-YACC we can also achieve conflict resolution by using **precedence declarations**.

Remember: `%left,%right` declare the associativity and order of precedence of symbols in increasing order. Symbols in the same declaration have the same precedence.

We know that our naive expression grammar is ambiguous:

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

We have previously resolved this by operator precedence parsing (or by rewriting the grammar).

If we declare the proper precedences/associativities in ML-YACC with the declarations

```
%left SUB PLUS
%left TIMES DIV
%right CARAT
```

the conflicts will be resolved appropriately.

Use ML-YACC to produce the parse table for the naive grammar. One of the conflicts will be found in a state with lookahead “+” which has the items

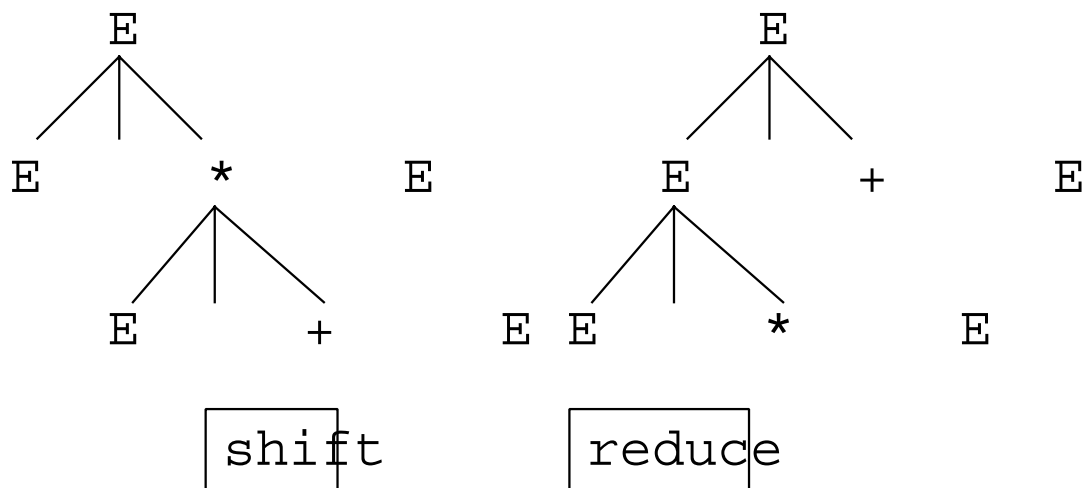
$$\begin{aligned} E &\rightarrow E * E. \quad (\text{lookahead } +) \\ E &\rightarrow E. + E \end{aligned}$$

In this state, the top of the stack will contain  $E * E$ .

- Shifting would lead to  $E * E +$ , then to  $E * E + E$  and subsequently by reduction of  $E + E$  to  $E$  again to  $E * E$ .
- Reducing would lead to  $E$  and subsequently to  $E +$ ,  $E + E$  and then again to  $E$  by reduction of  $E + E$  to  $E$ .

## Conflict Resolution by Precedence

The parse trees for these alternatives are given below



We should therefore **reduce** in favour of the operator with higher precedence.

The case for the conflict with items

$$\begin{array}{l}
 E \rightarrow E + E. \quad (\text{lookahead } +) \\
 E \rightarrow E. + E
 \end{array}$$

can be analysed in a similar fashion. *Shifting* will make “+” right-associative, *reducing* will make it left-associative.

To decide which *action* takes precedence in a *shift/reduce* conflict we need to look at the items:

$$\begin{array}{l}
 E \rightarrow E * E. \quad (\text{lookahead } +) \\
 E \rightarrow E. + E
 \end{array}$$

*The precedence of a reduce action is given by the precedence of the last token on the RHS of the item, the precedence of a shift action is given by the item to be shifted.*

Thus the precedences are:

- $E \rightarrow E * E$ . (action: reduce) has the precedence of “\*”,
- $E \rightarrow E. + E$  (action: shift) has the precedence of “+”

the rule with the action precedence will be executed (here: reduce).

Accordingly, if the precedence does not resolve the conflict (i.e. is the same), *left associativity* will favour reducing, *right associativity* will favour shifting.

## Error Correction

Sensible error handling is extremely important: imagine your compiler would abort the compilation of a faulty program with just the message “error - compilation aborted”!

Four levels of behaviour are possible when an error is encountered:

1. locate & report (without further analysis),
2. diagnose (report the type of error),
3. recover (i.e. skip over the error and continue parsing afterwards),
4. correct.

Any good compiler implements some level of recovery. Correction is the “holy grail” of parsing, but only possible in special cases.

The stack in the table driven recursive parser makes explicit the terminals and non-terminals it expects to match with the rest of the input. This can be used to guide error recovery.

A **syntax error** is detected when

- (a) the terminal on top of the stack does not match the input token,
- or
- (b) the non-terminal symbol on top of the stack  $A$  has an empty entry  $M[A, a]$  for the current input symbol  $a$ .

**Panic-mode** error recovery is based on the idea of skipping input symbols until a token in a selected set of **synchronizing** tokens appears.

For LL parsing, one heuristic is to place all of the symbols in  $FOLLOW(A)$  into the synchronizing set for non-terminal  $A$ .

We then skip input until a member of  $FOLLOW(A)$  is encountered, and pop  $A$  from the stack.

Another is to place all of the symbols in  $FIRST(A)$  into the synchronizing set for non-terminal  $A$ .

We then skip input until a member of  $FIRST(A)$  is encountered.

See Aho et al Section 4.4 for more details.

## Error Correction – Example

Recall

$$\begin{aligned}
 exp &\rightarrow term\ exp' && (P1) \\
 exp' &\rightarrow +exp && (P2) \\
 exp' &\rightarrow \epsilon && (P3) \\
 term &\rightarrow factor\ term' && (P4) \\
 term' &\rightarrow *term && (P5) \\
 term' &\rightarrow \epsilon && (P6) \\
 factor &\rightarrow \mathbf{int} && (P7) \\
 factor &\rightarrow (exp) && (P8)
 \end{aligned}$$

The LL parsing table with error correction for this grammar is

	<b>int</b>	+	*	(	)	\$	<i>first</i>	<i>follow</i>
<i>exp</i>	<i>P1</i>	<i>skip</i>	<i>skip</i>	<i>P1</i>	<i>synch</i>	<i>synch</i>	( <i>int</i>	) \$
<i>exp'</i>	<i>skip</i>	<i>P2</i>	<i>skip</i>	<i>skip</i>	<i>P3</i>	<i>P3</i>	+	) \$
<i>term</i>	<i>P4</i>	<i>synch</i>	<i>skip</i>	<i>P4</i>	<i>synch</i>	<i>synch</i>	( <i>int</i>	+ ) \$
<i>term'</i>	<i>skip</i>	<i>P6</i>	<i>P5</i>	<i>skip</i>	<i>P6</i>	<i>P6</i>	*	+ ) \$
<i>factor</i>	<i>P7</i>	<i>synch</i>	<i>synch</i>	<i>P8</i>	<i>synch</i>	<i>synch</i>	( <i>int</i>	+ * ) \$

A *skip* means to skip the current input terminal and continue.

A *synch* does the following where *A* is the non-terminal on top of the stack:

- skip input terminals until a symbol in  $FIRST(A)$  or  $FOLLOW(A)$  is encountered
- if the symbol is in  $FOLLOW(A)$  pop *A* off the stack
- continue.

## Error Correction in ML-YACC

Instead of this type of *local recovery* ML-YACC uses a *global recovery* (and more costly) technique called **Burke-Fisher Parsing**.

Imagine the correct ML definition

```
val inc = fn x => 1 + x;
```

Consider the damaged version

```
val inc = x => 1 + x;
```

The error would only be encountered when the => token is read and could therefore not be corrected appropriately by skipping input.

Burke-Fisher Parsing proceeds in the following way: If an error is encountered

- Perform a correction attempt at the current input position and each of the preceding 15 input positions (15 is a heuristic choice from experience).
- At each correction position try to
  - delete the token,
  - substitute the token by some other token,
  - insert an additional token
- Try to continue parsing with this change past the error position.

- Among all possible corrections choose the one that allows to continue parsing furthest past the error point.

Obviously the parser must be able to back up over the last 15 shift operations. To do this ML-YACC maintains two stacks: the *current* stack and the *previous* stack that are joined by a queue.

## Semantic Actions and Error Correction

It is obvious that semantic actions may only be executed during the error-recovery attempts if they are **side-effect free** as their side-effects could not be undone if the recovery attempt is unsuccessful.

ML-Yacc uses higher-order functions to defer the evaluation of all user semantic actions. The semantic actions will only be executed once the corresponding parser action are committed (i.e. reach the *previous* stack).

**%pure** may be used as a declaration if all semantic actions in a grammar are side-effect free (and always terminate). With this declaration the parser will not defer their evaluation.

We can now also understand the other declarations that we have skipped above:

**%prefer** lists the keyword whose insertion should be attempted first.

**%subst** lists pairs *terminal* for *terminal* that specify a heuristics for preferred substitution attempts. The pairs on this list must be separated by bars (“—”).

Finally, each token that is inserted and has a semantic value must, of course, supply a default value. This is handled by the *value declaration*, in our example grammar:

```
%value ID ("dummy")
```

## Generating a Parser with ML Yacc

*(For details you can also read the ML-YACC documentation, from which the following is adapted, but please make sure to follow the instructions here, as there are some bugs in the official documentation).*

To generate a parser using ML-YACC you need to follow these steps:

1. Run ML-Lex to create the lexical analyzer
2. Run ML-Yacc on the specification file for a grammar
3. Load the ML-Yacc libraries
4. Load the .sig file that ML-Yacc produced
5. Load the lexer that ML-Lex produced
6. Load the .sml file that ML-Yacc produced
7. Join the parser structure by applying functors
8. Define functions that couple lexer and parser

## Combining ML-Lex and ML-YACC

For the example parser we first generate a lexer. Store the following code in a file named `lex-ex.lex`.

```
structure Tokens = Tokens

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

val pos = ref 0
val eof = fn () => Tokens.EOF(!pos,!pos)
val error = fn (e,l : int,_) =>
    TextIO.output(TextIO.stdOut,"line "
        ^ (Int.toString l) ^ ": " ^ e ^ "\n")
%%
%header (functor
%      CalcLexFun(structure Tokens: Calc_TOKENS));
alpha=[A-Za-z];
digit=[0-9];
ws = [\ \t];
%%
\n      => (pos := (!pos) + 1; lex());
{ws}+   => (lex());
{digit}+ => (Tokens.NUM
            (foldl (fn (a,r) =>
ord(a)-ord("#0")+10*r) 0
                (explode ytext) ,
```

```

                                !pos, !pos));
"+"      => (Tokens.PLUS(!pos, !pos));
"*"      => (Tokens.TIMES(!pos, !pos));
";"      => (Tokens.SEMI(!pos, !pos));
{alpha}+ => (if yytext="print"
            then Tokens.PRINT(!pos, !pos)
            else Tokens.ID(yytext, !pos, !pos)
            );
"_"      => (Tokens.SUB(!pos, !pos));
"^"      => (Tokens.CARAT(!pos, !pos));
"/"      => (Tokens.DIV(!pos, !pos));
"."      => (error ("ignoring bad character "^yytext,
                    !pos, !pos);
            lex());

```

Note that the lexer contains a declaration for a *pos* type (token position).

The parser generator will generate a structure *Tokens* that the lexer uses. The declarations

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

are mandatory and create the attribute types for the terminals.

The declaration

```
%header (functor
          CalcLexFun(structure Tokens: Calc_TOKENS));
```

is also mandatory. It causes ML-Lex to create a functor for a lexer. The string *Calc* in this declaration refers to the name of the parser (given in the **%name** declaration of the parser) and has to be substituted if your parser has any other name.

## Combining ML-Lex and ML-YACC (cont.)

After you have generated the file for the lexer, store the example ML-YACC grammar from above in a file called `yacc-ex.grm`.

Now create a file “sources.sml” (don’t change the name) with the following contents:

```
Group is
  ml-yacc-lib.cm
  lex-ex.lex
  yacc-ex.grm
```

This file will be used by the so-called compilation manager, the ML equivalent of the Unix command “make”. We will not go into details of CM, but we need it to invoke ML-YACC.

Start SML and invoke the compilation manager:

```
bruce_39% sml
Standard ML of New Jersey, Version 110.0.7, ...
- CM.make();
[starting dependency analysis]
[scanning sources.cm]
...
```

This will load the YACC libraries and invoke the ML-LEX and ML-YACC on your specifications. Afterwards you will find the compiled `.sig` and `.sml` files in your directories (plus a `.desc` file with the verbose output of ML-YACC).

## Using the Parser

Now restart SML and load all the libraries for the parser as well as the files that were generated by ML-LEX and ML-YACC.

```
Standard ML of New Jersey, Version 110.0.7, ...
- use "/local/lib/sml/src/ml-yacc/lib/base.sig";
...
- use "/local/lib/sml/src/ml-yacc/lib/join.sml";
...
- use "/local/lib/sml/src/ml-yacc/lib/lrtable.sml";
...
- use "/local/lib/sml/src/ml-yacc/lib/stream.sml";
...
- use "/local/lib/sml/src/ml-yacc/lib/parser2.sml";
...
- use "yacc-ex.grm.sig";
...
- use "lex-ex.lex.sml";
...
- use "yacc-ex.grm.sml";
```

## Using the Parser (cont.)

Next, you must create the signature for the parser:

```
- structure CalcLrVals =
= CalcLrValsFun(structure Token = LrParser.Token);
structure CalcLrVals :
  sig
    structure ParserData : <sig>
    structure Tokens : <sig>
  end
- structure CalcLex =
= CalcLexFun(structure Tokens = CalcLrVals.Tokens);
structure CalcLex :
  sig
    structure Internal : <sig>
    structure UserDeclarations : <sig>
    exception LexError
    val makeLexer : (int -> string)
                    -> unit -> Internal.result
  end
- structure CalcParser=
= Join(structure ParserData = CalcLrVals.ParserData
= structure Lex = CalcLex
= structure LrParser = LrParser );
structure CalcParser : PARSE
```

This ultimately generates a structure *PARSER* that contains the finished Parser. You now need to define functions that invoke this parser and couple it with the lexer.

## Using the Parser (cont.)

We define a function *invoke* that calls the parser with a function for error output.

```
fun invoke lexstream =
  let fun print_error (s,i:int,_) =
        TextIO.output(TextIO.stdOut,
                      "Error, line " ^
                      (Int.toString i) ^ ", "
                      ^ s ^ "\n")
      in CalcParser.parse(0,lexstream,print_error,())
    end
```

## Using the Parser (cont.)

Finally, the function *parse* couples the lexer and the parser:

```
fun parse () =
  let val lexer = CalcParser.makeLexer
        (fn _ =>
          TextIO.inputLine TextIO.stdIn)
      val dummyEOF = CalcLrVals.Tokens.EOF(0,0)
      val dummySEMI = CalcLrVals.Tokens.SEMI(0,0)
      fun loop lexer =
          let val (result,lexer) = invoke lexer
              val (nextToken,lexer) =
CalcParser.Stream.get lexer
              in case result
                of SOME r =>
                     TextIO.output(TextIO.stdOut,
                                     "result = "
                                     ^ (Int.toString r) ^ "\n")
                 | NONE => ();
              if CalcParser.sameToken(nextToken,
                                     dummyEOF) then ()
              else loop lexer
          end
      in loop lexer
  end
```

Note that the lexer now returns a stream instead of a sequence of tokens.

## Using the Parser (cont.)

The parser is now ready to be used

```
- parse ();  
3+4*5;  
result = 23  
print 8+16*2;  
40  
result = 40
```

## Summary

We have looked at:

- Generic bottom-up parsing (CYK).
- The ML-YACC parser generator.

## Homework

- Read Section 4.4 of Aho et al.
- Read Section 3.4 and 3.5 of Appel
- Study the ML-YACC documentation at <http://www.cs.princeton.edu/~appel/modern/ml/ml-yacc/>. (but be aware that there are mistakes in it!)
- Modify the expression parser from the ML-YACC given in the lecture such that it accepts and evaluates function symbols (such as  $\sin(X)$ ) and unary minus without brackets, i.e. expressions of the form  $PRINT3 + 4 * -5$ .