# Outline

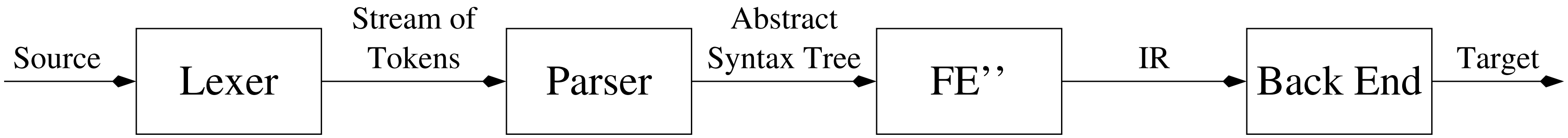
- ML-Yacc
- Read Chap 3 (Especially 3.4)

# Syntax Analysis

Front End:

- Lexical Analysis - Break source into *tokens*.
- Syntax Analysis - Parse phrase structure.
- Semantic Analysis - Calculate meaning.

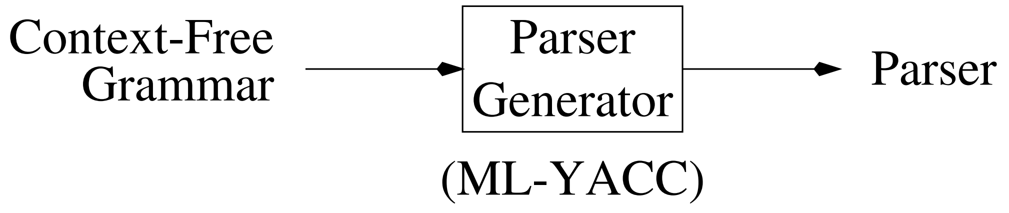
Our Compiler:

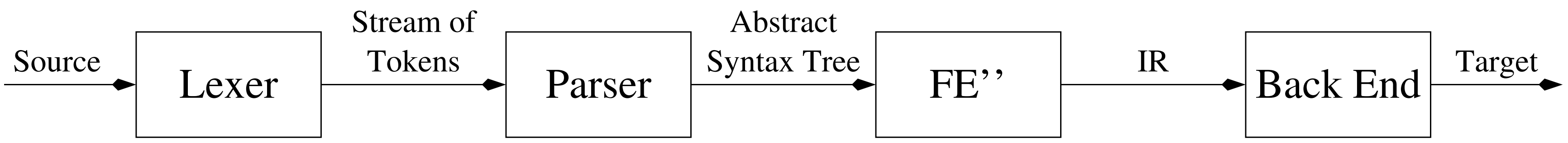



Parser Functions:

- Verify that token stream is valid.
- If it is not valid, report syntax error and recover.
- Build Abstract Syntax Tree (AST).

# ML-YACC ("Yet Another Compiler-Compiler")




- Input to **ml-yacc** is a context-free grammar specification.
- Output from **ml-yacc** is a shift-reduce parser in ML.

# Context-Free Grammar Specification

- CFG specification consists of 3 parts:

```
User Declarations
%%
ML-YACC Definitions
%%
Rules
```

- **User Declarlarions:** define various values that are available to *rules* section.
- **ML-YACC Definitions:** declare terminal and non-terminal symbols; declare precedences for terminals that help resolve shift-reduce conflicts.
- **Rules:** specify productions of grammar and *semantic actions* associated with productions.

# ML-YACC Declarations

- Need to specify type associated with positions of tokens in input file

```
%pos int
```

- Need to specify terminal and non-terminal symbols (no symbols can be in both lists)

```
%term IF | THEN | ELSE |...
%nonterm prog | stmt | expr |...
```

- Optionally specify end-of-parse symbol - terminals which may follow start symbol

```
%eop EOF
```

- Optionally specify start symbol - otherwise, LHS non-terminal of first rule is taken as start symbol

```
%start prog
```

# Attribute Grammar

- ML-YACC employs *attribute grammar* scheme
  - Each terminal or non-terminal symbol may have associated attribute/value.
  - When parser reduces using production $A \rightarrow \alpha$, semantic action associated with production is exectued in order to compute value for A based on the values of symbols in $\alpha$.
  - Parser returns value associated with start symbol. (If no attribute, `()` is returned.)
- Can specify *types* of attributes associated with symbols.

```
%term     ID of string | NUM of int | IF | THEN | ...
%nonterm  prgm | stmt | expr of int | ...
```

# Rules

$$symbol_0 \ : \ symbol_1 \ symbol_2 \ ... \ symbol_n \ (\textit{semantic\_action})$$

- Semantic action typically builds piece of AST corresponding to derived string
- Can access attribute/value of RHS symbol X using X<n>, where n specifies a particular occurrence of X on RHS.

```
%term    PLUS | MINUS | NUM of int | ...
%nonterm exp of int | ...

exp: exp PLUS exp     (exp1 + exp2)
|    exp MINUS exp    (exp1 - exp2)
|    NUM              (NUM)
```

- Type of value computed by semantic action must match type of value associated with LHS non-terminal.

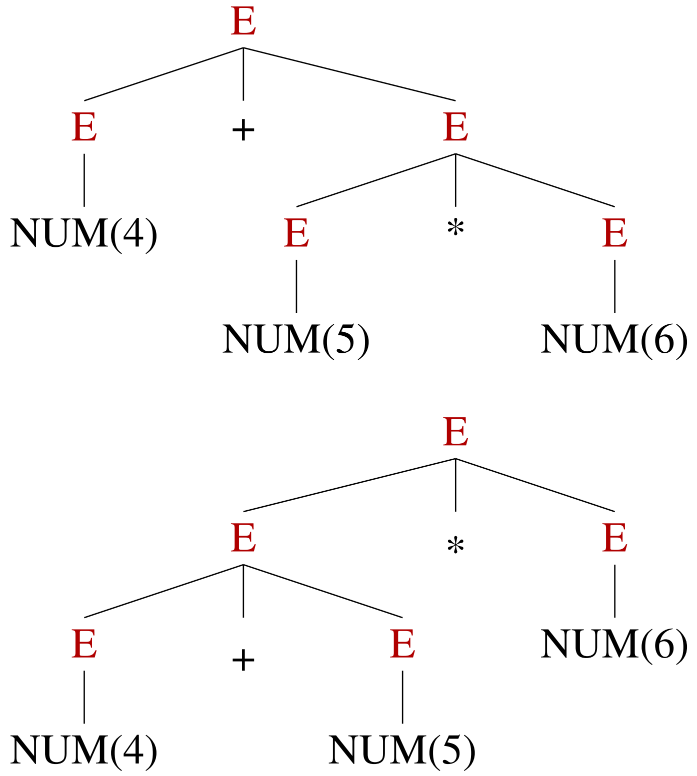
# Example

```
%%

%term     ID | NUM | PLUS | MINUS | MULT | DIV | EOF
%nonterm expr
%pos int
%start expr
%eop EOF
%verbose

%%

expr : ID                 ()
     | NUM                ()
     | expr PLUS expr     ()
     | expr MINUS expr    ()
     | expr MULT expr     ()
     | expr DIV expr      ()
```

# ML-YACC and Ambiguous Grammars

- A grammar is ambiguous if it can derive a string of tokens with two or more different parse trees.
- Consider: $4 + 5 * 6$, `NUM(4) PLUS NUM(5) MULT NUM(6)`

$$\begin{array}{lll} expr & \rightarrow & ID \\ expr & \rightarrow & NUM \\ expr & \rightarrow & expr + expr \\ expr & \rightarrow & expr * expr \end{array}$$




- We perfer to bind “*” tighter than “+”.

# ML-YACC and Ambiguous Grammars

- ML-YACC will report *shift-reduce* conflicts when parsing strings.
  - $4 + 5 * 6$, `NUM(4) PLUS NUM(5) MULT NUM(6)`
    - At some point, E + E will be on top of stack, "*" will be the current token-type in stream.
    - Parser can reduce by rule E $\rightarrow$ E + E, or shift. Prefer *shift*.

# ML-YACC and Ambiguous Grammars

- Similarly Consider: $4 - 5 - 6$, `NUM(4) MINUS NUM(5) MINUS NUM(6)`
- We perfer to bind left "-" first.
  - At some point, E - E will be on top of stack, "-" will be the current token-type in stream.
  - Parser can reduce by rule E $\rightarrow$ E - E, or shift. Prefer *reduce*.

# Directives

**Three Solutions:**

1. Let YACC complain, but demonstrate that its choice (to shift) was correct.
2. Rewrite grammar to eliminate ambiguity.
3. Keep grammar, but add *precedence directives* which enable conflicts to be resolved. Use `%left, %right, %nonassoc`

   - For this grammar:
     ```
     %left PLUS MINUS
     %left MULT DIV
     ```
   - PLUS, MINUS are left associative, bind equally tightly
   - MULT, DIV are left associative, bind equally tightly
   - MULT, DIV bind tighter than PLUS, MINUS (ie: MULT, DIV have higher precedence)

# Directives

- Given directives, ML-YACC assigns precedence to each *terminal* and *rule*
  - Precedence of terminal based on order in which associativity specified
  - Precedence of rule is the precedence of right-most terminal. For example, precedence(E $\rightarrow$ E + E) $=$ precedence(PLUS).
- Given shift-reduce conflict, ML-YACC performs the following:
  1. Find precedence of rule to be reduced, terminal to be shifted.
  2. prec(terminal) $>$ prec(rule) $\Rightarrow$ shift.
  3. prec(rule) $>$ prec(terminal) $\Rightarrow$ reduce.
  4. prec(terminal) $=$ prec(rule), then:
     - assoc(terminal) $=$ left $\Rightarrow$ reduce.
     - assoc(terminal) $=$ right $\Rightarrow$ shift.
     - assoc(terminal) $=$ nonassoc $\Rightarrow$ report as error.

# Precedence Examples

```
 input: 4 + 5 * 6
               |
1
 stack: 4 + 5
action: prec(*) > prec(+) -> ?
```

---

```
 input: 4 * 5 + 6
               |
2
 stack: 4 * 5
action: prec(*) > prec(+) -> ?
```

---

```
 input: 4 + 5 + 6
               |
3
 stack: 4 + 5
action: assoc(+) = left -> ?
```

# Precedence Examples

**1**

```
 input: 4 + 5 * 6
              |
 stack: 4 + 5
action: prec(*) > prec(+) -> shift
```

---

**2**

```
 input: 4 * 5 + 6
              |
 stack: 4 * 5
action: prec(*) > prec(+) -> ?
```

---

**3**

```
 input: 4 + 5 + 6
              |
 stack: 4 + 5
action: assoc(+) = left -> ?
```

# Precedence Examples

**1**

```
  input: 4 + 5 * 6
               |
  stack: 4 + 5
 action: prec(*) > prec(+) -> shift
```

---

**2**

```
  input: 4 * 5 + 6
               |
  stack: 4 * 5
 action: prec(*) > prec(+) -> reduce
```

---

**3**

```
  input: 4 + 5 + 6
               |
  stack: 4 + 5
 action: assoc(+) = left -> ?
```

# Precedence Examples

**1**

```
 input: 4 + 5 * 6
              |
 stack: 4 + 5
action: prec(*) > prec(+) -> shift
```

---

**2**

```
 input: 4 * 5 + 6
              |
 stack: 4 * 5
action: prec(*) > prec(+) -> reduce
```

---

**3**

```
 input: 4 + 5 + 6
              |
 stack: 4 + 5
action: assoc(+) = left -> reduce
```

# One More Example

The unary minus problem:

```
%left PLUS MINUS
%left MULT DIV

exp : MINUS expr
    | expr PLUS expr ()
    | expr MULT expr ()...
```

Consider parsing:

```
 input: - 4 * 5
            |
 stack: - 4
action: prec(*) > prec(-) -> ?
```

# Direct Rule Presidence Specification

```
 input: - 4 * 5
              |
 stack: - 4
action: prec(*) > prec(-) -> shift
```

- We perfer to bind left unary minus ("-") tighter. Here, precedence of MINUS is lower than MULT, so we get $-(4 * 6)$, not $(-4) * 6$:
- Solution: Assign *specific* precedence to rule, rather than precedence of last terminal.
- Use the %prec directive.

```
%left PLUS MINUS
%left MULT DIV
%left UMINUS

exp : MINUS expr %prec UMINUS ()
    | expr PLUS expr ()...
```

# The Dangling Else Problem

- Valid Program: if a then if b then S1 else S2

  **1** S $\rightarrow$ *if* E *then* S *else* S
  **2** S $\rightarrow$ *if* E *then* S
  **3** S $\rightarrow$ OTHER

- 2 interpretations: if a then [if b then S1 else S2]
  if a then [if b then S1] else S2

- Want first behavoir, but parse will report *shift-reduce conflict* when S1 is on top stack.

- Eliminate Ambiguity by modifying grammar (matched/unmatched):

  **1** S $\rightarrow$ M
  **2** S $\rightarrow$ U
  **3** M $\rightarrow$ *if* E *then* M *else* M
  **4** M $\rightarrow$ OTHER
  **5** U $\rightarrow$ *if* E *then* S
  **6** U $\rightarrow$ *if* E *then* M *else* U

# Default Behavior

**What if directives not specified?**

- shift-reduce: report error, *shift* by default.
- reduce-reduce: report error, reduce by rule that occurs first.

**What to do:**

- shift-reduce: acceptable in well defined cases (dangling else).
- reduce-reduce: unacceptable. Rewrite grammar.

# Syntax vs. Semantics

**Consider language with two classes of expressions**

- *Arithmetic* expressions (ae)

```
ae : ae PLUS ae ()
   | ID         ()
```

- *Boolean* expressions (be)

```
be : be AND be ()
   | be OR be  ()
   | be EQ be  ()
   | ID        ()
```

- Consider: `a := b`, ID(a) ASSIGN ID(b):
  - Reduce-reduce conflict - parser can't choose between be $\rightarrow$ ID or ae $\rightarrow$ ID.
  - For now ae and be should be aliased - let semantic analysis (next phase) determine that `a & b + c` is a type error.
  - Type checking cannot be done easily in context free grammars.