Put table of reg exp and corresponding actions (written in ocaml) into a file `<filename>.mll`

Call

```
ocamllex <filename>.mll
```

Produces Ocaml code for a lexical analyzer in file `<filename>.ml`

---

**Sample Input**

```
rule main = parse
  ['0'..'9']+ { print_string "Int
"}
| ['0'..'9']+'.'['0'..'9']+ { print_string "Float
"}
| ['a'..'z']+ { print_string "String
"}
| _ { main lexbuf }
{
  let newlexbuf = (Lexing.from_channel stdin) in
  main newlexbuf
}
```

---

**General Input**

```
{ header }
let ident = regexp ...
rule entrypoint [arg1... argn] = parse
  regexp { action }
| ...
| regexp { action }
and entrypoint [arg1... argn] = parse ...
and ...
{ trailer }
```

---

**Ocamllex Input**

- *header* and *trailer* contain arbitrary ocaml code put at top an bottom of `<filename>.ml`

- let ident = regexp ... Introduces ident for use in later regular expressions

---

**Ocamllex Input**

- `<filename>.ml` contains one lexing function per *entrypoint*

  - Name of function is name given for *entrypoint*

  - Each entry point becomes an Ocaml function that takes n+1 arguments, the extra implicit last argument being of type `Lexing.lexbuf`

  - *arg1... argn* are for use in *action*
Ocamlllex Regular Expression

- Single quoted characters for letters: ‘a’
- _: (underscore) matches any letter
- Eof: special “end_of_file” marker
- Concatenation same as usual
- “string”: concatenation of sequence of characters
- e₁ | e₂: choice - what was e₁ ∨ e₂

Ocamlllex Regular Expression

- [c₁ - c₂]: choice of any character between first and second inclusive, as determined by character codes
- [^c₁ - c₂]: choice of any character NOT in set
- e*: same as before
- e+: same as e e*
- e?: option - was e ∨ ε
- (e): same as e

Ocamlllex Manual

- More details can be found at
- Version for ocaml 4.07: https://v2.ocaml.org/releases/4.07/htmlman/lexyacc.html
  (same, except formatting, I think)

Example : test.mll

{ type result = Int of int | Float of float | String of string }

let digit = ['0'-'9']
let digits = digit +
let lower_case = ['a'-'z']
let upper_case = ['A'-'Z']
let letter = upper_case | lower_case
let letters = letter +

Example : test.mll

rule main = parse
  (digits)'.'digits as f { Float (float_of_string f) }
| digits as n              { Int (int_of_string n) }
| letters as s             { String s}
| _ { main lexbuf }
{ let newlexbuf = (Lexing.from_channel stdin) in
  print_newline ();
  main newlexbuf }

4/7/24
```ocaml
# use "test.ml";;
...
val main : Lexing.lexbuf -> result = <fun>
val __ocaml_lex_main_rec : Lexing.lexbuf -> int -> result = <fun>
hi there 234 5.2
  : result = String "hi"
```
Dealing with comments

```plaintext
| open_comment         { comment lexbuf}
| eof                  { [] }
| _ { main lexbuf }
and comment = parse
close_comment        { main lexbuf }
| _                   { comment lexbuf }
```

Dealing with nested comments

```plaintext
rule main = parse ...
| open_comment         { comment 1 lexbuf}
| eof                  { [] }
| _ { main lexbuf }
and comment depth = parse
    open_comment        { comment (depth+1) lexbuf
| close_comment       { if depth = 1
    then main lexbuf
        else comment (depth - 1) lexbuf }
| _                   { comment depth lexbuf }
```

Dealing with nested comments

```plaintext
rule main = parse
    (digits) '.' digits as f { Float (float_of_string f) :: main lexbuf}
| digits as n          { Int (int_of_string n) :: main lexbuf }
| letters as s         { String s :: main lexbuf}
| open_comment         { (comment 1 lexbuf}
| eof                  { [] }
| _ { main lexbuf }
```

Dealing with nested comments

```plaintext
rule main = parse
    (digits) '.' digits as f { Float (float_of_string f) :: main lexbuf}
| digits as n          { Int (int_of_string n) :: main lexbuf }
| letters as s         { String s :: main lexbuf}
| open_comment         { (comment 1 lexbuf}
| eof                  { [] }
| _ { main lexbuf }
```

Types of Formal Language Descriptions

- Regular expressions, regular grammars
- Context-free grammars, BNF grammars, syntax diagrams
- Finite state automata
- Pushdown automata
- Whole family more of grammars and automata – covered in automata theory

BNF Grammars

- Start with a set of characters, \( a,b,c,... \)
  - We call these **terminals**
- Add a set of different characters, \( X,Y,Z,... \)
  - We call these **nonterminals**
- One special nonterminal \( S \) called **start symbol**
BNF Grammars

- BNF rules (aka *productions*) have form
  \[ X \ ::= \ y \]
  where \( X \) is any nonterminal and \( y \) is a string of terminals and nonterminals
- BNF grammar is a set of BNF rules such that every nonterminal appears on the left of some rule

Sample Grammar

- Terminals: 0 1 + ( )
- Nonterminals: \(<\text{Sum}>\)
- Start symbol = \(<\text{Sum}>\)
- \(<\text{Sum}>\) ::= 0
- \(<\text{Sum}>\) ::= 1
- \(<\text{Sum}>\) ::= \(<\text{Sum}>\) + \(<\text{Sum}>\)
- \(<\text{Sum}>\) ::= (\(<\text{Sum}>\))
- Can be abbreviated as
  \(<\text{Sum}>\) ::= 0 | 1
  | \(<\text{Sum}>\) + \(<\text{Sum}>\) | (\(<\text{Sum}>\))

BNF Derivations

- Given rules
  \[ X ::= yZw \text{ and } Z ::= v \]
  we may replace \( Z \) by \( v \) to say
  \[ X \Rightarrow yZw \Rightarrow yyw \]
- Sequence of such replacements called *derivation*
- Derivation called *right-most* if always replace the right-most non-terminal

BNF Semantics

- The meaning of a BNF grammar is the set of all strings consisting only of terminals that can be derived from the Start symbol

BNF Derivations

- Start with the start symbol:
  \[ <\text{Sum}> => \]

BNF Derivations

- Pick a non-terminal
  \[ <\text{Sum}> => \]
BNF Derivations

- Pick a rule and substitute:
  - $<\text{Sum}> ::= <\text{Sum}> + <\text{Sum}>$
  $<\text{Sum}> => <\text{Sum}> + <\text{Sum}>$

BNF Derivations

- Pick a non-terminal:
  - $<\text{Sum}> => <\text{Sum}> + <\text{Sum}>$

BNF Derivations

- Pick a rule and substitute:
  - $<\text{Sum}> ::= ( <\text{Sum}> )$
  $<\text{Sum}> => <\text{Sum}> + <\text{Sum}>$
  $=> ( <\text{Sum}> ) + <\text{Sum}>$

BNF Derivations

- Pick a non-terminal:
  - $<\text{Sum}> => <\text{Sum}> + <\text{Sum}>$
  $=> ( <\text{Sum}> ) + <\text{Sum}>$

BNF Derivations

- Pick a rule and substitute:
  - $<\text{Sum}> ::= <\text{Sum}> + <\text{Sum}>$
  $<\text{Sum}> => <\text{Sum}> + <\text{Sum}>$
  $=> ( <\text{Sum}> ) + <\text{Sum}>$
  $=> ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}>$

BNF Derivations

- Pick a non-terminal:
  - $<\text{Sum}> => <\text{Sum}> + <\text{Sum}>$
  $=> ( <\text{Sum}> ) + <\text{Sum}>$
  $=> ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}>$
BNF Derivations

- Pick a rule and substitute:
  - `<Sum> ::= 1`
  - `<Sum> => <Sum> + <Sum>`
    - `=> ( <Sum> ) + <Sum>`
    - `=> ( <Sum> + <Sum> ) + <Sum>`
    - `=> ( <Sum> + 1 ) + <Sum>`

BNF Derivations

- Pick a non-terminal:
  - `<Sum> => <Sum> + <Sum>`
    - `=> ( <Sum> ) + <Sum>`
    - `=> ( <Sum> + <Sum> ) + <Sum>`
    - `=> ( <Sum> + 1 ) + <Sum>`

BNF Derivations

- Pick a rule and substitute:
  - `<Sum> ::= 0`
  - `<Sum> => <Sum> + <Sum>`
    - `=> ( <Sum> ) + <Sum>`
    - `=> ( <Sum> + <Sum> ) + <Sum>`
    - `=> ( <Sum> + 1 ) + <Sum>`
    - `=> ( <Sum> + 1 ) + 0`

BNF Derivations

- Pick a non-terminal:
  - `<Sum> => <Sum> + <Sum>`
    - `=> ( <Sum> ) + <Sum>`
    - `=> ( <Sum> + <Sum> ) + <Sum>`
    - `=> ( <Sum> + 1 ) + <Sum>`
    - `=> ( <Sum> + 1 ) + 0`

BNF Derivations

- Pick a rule and substitute:
  - `<Sum> ::= 0`
  - `<Sum> => <Sum> + <Sum>`
    - `=> ( <Sum> ) + <Sum>`
    - `=> ( <Sum> + <Sum> ) + <Sum>`
    - `=> ( <Sum> + 1 ) + <Sum>`
    - `=> ( <Sum> + 1 ) 0`
    - `=> ( 0 + 1 ) 0`

BNF Derivations

- (0 + 1) + 0 is generated by grammar
  - `<Sum> => <Sum> + <Sum>`
    - `=> ( <Sum> ) + <Sum>`
    - `=> ( <Sum> + <Sum> ) + <Sum>`
    - `=> ( <Sum> + 1 ) + <Sum>`
    - `=> ( <Sum> + 1 ) + 0`
    - `=> ( 0 + 1 ) + 0`
Extended BNF Grammars

- Alternatives: allow rules of from \( X::= y | z \)
  - Abbreviates \( X::= y, X::= z \)
- Options: \( X::= y[v]z \)
  - Abbreviates \( X::= yvz, X::= yz \)
- Repetition: \( X::= y(v)^*z \)
  - Can be eliminated by adding new nonterminal \( V \) and rules \( X::= yz, X::= wz, V::= v, V::= w \)

Parse Trees

- Graphical representation of derivation
- Each node labeled with either non-terminal or terminal
- If node is labeled with a terminal, then it is a leaf (no sub-trees)
- If node is labeled with a non-terminal, then it has one branch for each character in the right-hand side of rule used to substitute for it

Example

- Consider grammar:
  \[
  \begin{align*}
  <\text{exp}> &::= <\text{factor}> \\
  &\mid <\text{factor}> + <\text{factor}> \\
  <\text{factor}> &::= <\text{bin}> \\
  &\mid <\text{bin}> * <\text{exp}> \\
  <\text{bin}> &::= 0 \mid 1
  \end{align*}
  \]
- Problem: Build parse tree for \( 1 * 1 + 0 \) as an \( <\text{exp}> \)

Example cont.

- \( 1 * 1 + 0: <\text{exp}> \)
  \[
  \begin{array}{c}
  \text{factor} \\
  \text{bin} * \text{exp}
  \end{array}
  \]
  - \( <\text{exp}> \) is the start symbol for this parse tree

Example cont.

- Use rule: \( <\text{exp}> ::= <\text{factor}> \)

Example cont.

- \( 1 * 1 + 0: <\text{exp}> <\text{factor}> \)
  \[
  \begin{array}{c}
  \text{bin} \\
  * \text{exp}
  \end{array}
  \]
  - Use rule: \( <\text{factor}> ::= <\text{bin}> * <\text{exp}> \)
Example cont.

1 * 1 + 0:  \[
<exp> \\
\text{\textbf{<factor>}} \\
\text{\textbf{<bin>}} \ast \text{\textbf{<exp>}} \\
1 \text{\textbf{<factor>}} + \text{\textbf{<factor>}}
\]

Use rules:  \[
<bin> ::= 1 \quad \text{and} \\
<exp> ::= \text{<factor>} + \text{<factor>}
\]

Example cont.

1 * 1 + 0:  \[
<exp> \\
\text{\textbf{<factor>}} \\
\text{\textbf{<bin>}} \ast \text{\textbf{<exp>}} \\
1 \text{\textbf{<factor>}} + \text{\textbf{<factor>}} \\
\text{\textbf{<bin>}} \text{\textbf{<bin>}}
\]

Use rule:  \[
<factor> ::= \text{<bin>}
\]

Example cont.

1 * 1 + 0:  \[
<exp> \\
\text{\textbf{<factor>}} \\
\text{\textbf{<bin>}} \ast \text{\textbf{<exp>}} \\
1 \text{\textbf{<factor>}} + \text{\textbf{<factor>}} \\
\text{\textbf{<bin>}} \text{\textbf{<bin>}} \text{\textbf{<bin>}}
\]

Use rules:  \[
<bin> ::= 1 \ | \ 0
\]

Example cont.

1 * 1 + 0:  \[
<exp> \\
\text{\textbf{<factor>}} \\
\text{\textbf{<bin>}} \ast \text{\textbf{<exp>}} \\
1 \text{\textbf{<factor>}} + \text{\textbf{<factor>}} \\
\text{\textbf{1}} \text{\textbf{<bin>}} \text{\textbf{<bin>}} \text{\textbf{1}} \text{\textbf{0}}
\]

Fringe of tree is string generated by grammar

Parse Tree Data Structures

Parse trees may be represented by OCaml datatypes  
One datatype for each nonterminal  
One constructor for each rule  
Defined as mutually recursive collection of datatype declarations

Example

Recall grammar:  
\[
<exp> ::= \text{<factor>} | \text{<factor>} + \text{<factor>} \\
<factor> ::= \text{<bin>} | \text{<bin>} \ast \text{<exp>} \\
<bin> ::= 0 | 1
\]

\text{type exp = Factor2Exp of factor}  
\text{ Plus of factor \ast factor}  
\text{and factor = Bin2Factor of bin}  
\text{ Mult of bin \ast exp}  
\text{and bin = Zero | One}
Example cont.

1 * 1 + 0:  
\[
\text{exp} \\
\text{factor} \\
\text{bin} \\
1 \\
\text{exp} \\
\text{factor} \\
\text{bin} \\
1 \\
\text{factor} \\
\text{bin} \\
0
\]

Can be represented as

\[
\text{Factor2Exp} \\
(\text{Mult(One,} \\
\text{Plus(Bin2Factor One,} \\
\text{Bin2Factor Zero))))
\]

Ambiguous Grammars and Languages

A BNF grammar is ambiguous if its language contains strings for which there is more than one parse tree

If all BNF’s for a language are ambiguous then the language is inherently ambiguous

Example: Ambiguous Grammar

\[
0 + 1 + 0 \\
\text{Sum} \\
\text{Sum} \\
\text{Sum} + \text{Sum} \\
\text{Sum} + \text{Sum} \\
\text{Sum} + \text{Sum} \\
0 \\
1 \\
0 \\
1 \\
0
\]

Two Major Sources of Ambiguity

Lack of determination of operator precedence

Lack of determination of operator associativity

Not the only sources of ambiguity

Disambiguating a Grammar

Given ambiguous grammar G, with start symbol S, find a grammar G’ with same start symbol, such that

\[
\text{language of G} = \text{language of G'}
\]

Not always possible

No algorithm in general
Disambiguating a Grammar

- Idea: Each non-terminal represents all strings having some property
- Identify these properties (often in terms of things that can’t happen)
- Use these properties to inductively guarantee every string in language has a unique parse

Steps to Grammar Disambiguation

- Identify the rules and a smallest use that display ambiguity
- Decide which parse to keep; why should others be thrown out?
- What syntactic restrictions on subexpressions are needed to throw out the bad (while keeping the good)?
- Add a new non-terminal and rules to describe this set of restricted subexpressions (called stratifying, or refactoring)
- Characterize each non-terminal by a language invariant
- Replace old rules to use new non-terminals
- Rinse and repeat

Example

- Ambiguous grammar:
  \[ <exp> ::= 0 \mid 1 \mid <exp> + <exp> \mid <exp> * <exp> \]
- String with more than one parse:
  \[ 0 + 1 + 0 \]
  \[ 1 * 1 + 1 \]
- Source of ambiguity: associativity and precedence

Two Major Sources of Ambiguity

- Lack of determination of operator precedence
- Lack of determination of operator associativity
- Not the only sources of ambiguity

How to Enforce Associativity

- Have at most one recursive call per production
- When two or more recursive calls would be natural leave right-most one for right associativity, left-most one for left associativity

Example

- \[ <Sum> ::= 0 \mid 1 \mid <Sum> + <Sum> \mid (<Sum>) \]
- Becomes
  \[ <Sum> ::= <Num> \mid <Num> + <Sum> \]
  \[ <Num> ::= 0 \mid 1 \mid (<Sum>) \]
  \[ <Sum> + <Sum> + <Sum> \]
Operator Precedence

- Operators of highest precedence evaluated first (bind more tightly).
- Precedence for infix binary operators given in following table
- Needs to be reflected in grammar

Precedence Table - Sample

<table>
<thead>
<tr>
<th></th>
<th>Fortan</th>
<th>Pascal</th>
<th>C/C++</th>
<th>Ada</th>
<th>SML</th>
</tr>
</thead>
<tbody>
<tr>
<td>highest</td>
<td>**</td>
<td>* /, div, mod</td>
<td>++, --</td>
<td>**</td>
<td>div, mod, /, *</td>
</tr>
<tr>
<td></td>
<td>* /, mod</td>
<td>+, -</td>
<td>**</td>
<td>/, *</td>
<td>+, -</td>
</tr>
<tr>
<td></td>
<td>+, -</td>
<td>+, -</td>
<td>-</td>
<td>+, -</td>
<td>::</td>
</tr>
</tbody>
</table>

Infix Binary Operators Precedence

- Higher precedence translates to longer derivation chain
- Example:
  \[
  \text{<exp>} ::= 0 | 1 \mid \text{<exp>} + \text{<exp>}
  \mid \text{<exp>} \times \text{<exp>}
  \]
  - Becomes
  \[
  \text{<exp>} ::= \text{<mult_exp>}
  \mid \text{<exp>} + \text{<mult_exp>}
  \text{<mult_exp>} ::= \text{id} \mid \text{<mult_exp>} \times \text{id}
  \text{id} ::= 0 \mid 1
  \]
  
Predicate in Grammar

Parser Code

- `<grammar>.mly` defines one parsing function per entry point
- Parsing function takes a lexing function (lexer buffer to token) and a lexer buffer as arguments
- Returns semantic attribute of corresponding entry point

Ocamlyacc Input

- File format:
  ```
  %{  
  <header>
  %}
  <declarations>
  %
  <rules>
  %
  <trailer>
  ```

 4/7/24

Ocamlyacc `<header>`

- Contains arbitrary Ocaml code
- Typically used to give types and functions needed for the semantic actions of rules and to give specialized error recovery
- May be omitted
- `<footer>` similar. Possibly used to call parser

4/7/24
Ocamlyacc <declarations>

- `%token symbol ... symbol`
  Declare given symbols as tokens
- `%token <type> symbol ... symbol`
  Declare given symbols as token constructors, taking an argument of type `<type>`
- `%start symbol ... symbol`
  Declare given symbols as entry points; functions of same names in `<grammar>.ml`

Example - Base types

```ocaml
(* File: expr.ml *)
type expr =
  Term_as_Expr of term |
  Plus_Expr of (term * expr) |
  Minus_Expr of (term * expr)
and term =
  Factor_as_Term of factor |
  Mult_Term of (factor * term) |
  Div_Term of (factor * term)
and factor =
  Id_as_Factor of string |
  Parenthesized_Expr_as_Factor of expr
```

Ocamlyacc <rules>

- `nonterminal :` 
  - symbol ... symbol { semantic_action }
    | ... 
    | symbol ... symbol { semantic_action } ;
  - Semantic actions are arbitrary Ocaml expressions
  - Must be of same type as declared (or inferred) for nonterminal
  - Access semantic attributes (values) of symbols by position: $1$ for first symbol, $2$ to second ...

Example - Lexer (exprlex.mll)

```ocaml
{ (*open Exprparse*) }
let numeric = ['0' '-' '9']
let letter = ['a' '-' 'z' 'A' '-' 'Z']
rule token = parse
  | "+" {Plus_token}
  | ".n" {Minus_token}
  | "+x" {Times_token}
  | "+d" {Divide_token}
  | "+l" {Left_parenthesis}
  | "+r" {Right_parenthesis}
  | letter (letter|numeric|"_")* as id {Id_token id}
  | [
    " " 
    \n\n    ]* {token lexbuf}
  | eof {EOL}
```

Example - Parser (exprparse.mly)

```ocaml
{% open Expr%
%%
%token <string> Id_token
%token Left_parenthesis Right_parenthesis
%token Times_token Divide_token
%token Plus_token Minus_token
%token EOL
%start main
%type <expr> main
%%
```
Example - Parser (exprparse.mly)

expr:
  term
  { Term_as.Expr $1 }
  | term Plus_token expr
  { Plus.Expr ($1, $3) }
  | term Minus_token expr
  { Minus.Expr ($1, $3) }

Example - Using Parser

# #use "expr.ml";;
...
# #use "exprparse.ml";;
...
# #use "exprlex.ml";;
...
# let test s =
  let lexbuf = Lexing.from_string (s^"\n") in
  main token lexbuf;;

Example - Using Parser

# test "a + b";;
- : expr =
  Plus.Expr
  (Factor.as.Term (Id.as.Term "a"),
   Term.as.Expr (Factor.as.Term (Id.as.Term "b")))

LR Parsing

- Read tokens left to right (L)
- Create a rightmost derivation (R)
- How is this possible?
- Start at the bottom (left) and work your way up
- Last step has only one non-terminal to be replaced so is right-most
- Working backwards, replace mixed strings by non-terminals
- Always proceed so that there are no non-terminals to the right of the string to be replaced
Example: \(<\text{Sum}\> = 0 \mid 1 \mid (<\text{Sum}\>)\)

\(<\text{Sum}\> \Rightarrow \)

\[\begin{align*}
\text{shift} &\quad \Rightarrow (0 \oplus 1) + 0 \\
&\quad = (0 + 1) + 0 \\
&\quad = (0 + 1) + 0 \\
\end{align*}\]

4/7/24 102

Example: \(<\text{Sum}\> = 0 \mid 1 \mid (<\text{Sum}\>)\)

\(<\text{Sum}\> \Rightarrow \)

\[\begin{align*}
\text{shift} &\quad \Rightarrow (0 \oplus 1) + 0 \\
&\quad = (0 + 1) + 0 \\
&\quad = (0 + 1) + 0 \\
\end{align*}\]

4/7/24 103

Example: \(<\text{Sum}\> = 0 \mid 1 \mid (<\text{Sum}\>)\)

\(<\text{Sum}\> \Rightarrow \)

\[\begin{align*}
\text{shift} &\quad \Rightarrow (0 \oplus 1) + 0 \\
&\quad = (0 + 1) + 0 \\
&\quad = (0 + 1) + 0 \\
\end{align*}\]

4/7/24 104

Example: \(<\text{Sum}\> = 0 \mid 1 \mid (<\text{Sum}\>)\)

\(<\text{Sum}\> \Rightarrow \)

\[\begin{align*}
\text{shift} &\quad \Rightarrow (0 \oplus 1) + 0 \\
&\quad = (0 + 1) + 0 \\
&\quad = (0 + 1) + 0 \\
\end{align*}\]

4/7/24 105

Example: \(<\text{Sum}\> = 0 \mid 1 \mid (<\text{Sum}\>)\)

\(<\text{Sum}\> \Rightarrow \)

\[\begin{align*}
\text{shift} &\quad \Rightarrow (0 \oplus 1) + 0 \\
&\quad = (0 + 1) + 0 \\
&\quad = (0 + 1) + 0 \\
\end{align*}\]

4/7/24 106

Example: \(<\text{Sum}\> = 0 \mid 1 \mid (<\text{Sum}\>)\)

\(<\text{Sum}\> \Rightarrow \)

\[\begin{align*}
\text{shift} &\quad \Rightarrow (0 \oplus 1) + 0 \\
&\quad = (0 + 1) + 0 \\
&\quad = (0 + 1) + 0 \\
\end{align*}\]

4/7/24 107
Example: \( \text{<Sum>} = 0 \mid 1 \mid (\text{<Sum>}) \)
| \( \text{<Sum>} + \text{<Sum>} \)

\[
\begin{align*}
\text{<Sum>} & \Rightarrow \\
& \Rightarrow (\text{<Sum>} + \text{<Sum>} \bullet) + 0 \quad \text{reduce} \\
& \Rightarrow (\text{<Sum>} + 1 \bullet) + 0 \quad \text{reduce} \\
& = (\text{<Sum>} + \bullet 1) + 0 \quad \text{shift} \\
& = (\text{<Sum>} \bullet + 1) + 0 \quad \text{shift} \\
& \Rightarrow (0 \bullet + 1) + 0 \quad \text{reduce} \\
& = (0 + 1) + 0 \quad \text{shift} \\
& = \bullet (0 + 1) + 0 \quad \text{shift}
\end{align*}
\]
Example: $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$

$\langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$ reduce
$\Rightarrow \langle \text{Sum} \rangle + 0$ reduce
$= \langle \text{Sum} \rangle + 0$ shift
$= \langle \text{Sum} \rangle + 0$ shift
$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle) + 0$ reduce
$\Rightarrow (\langle \text{Sum} \rangle + 1) + 0$ reduce
$= (\langle \text{Sum} \rangle + 1) + 0$ shift
$= (\langle \text{Sum} \rangle + 1) + 0$ shift
$\Rightarrow (0 + 1) + 0$ reduce
$= (0 + 1) + 0$ shift
$= (0 + 1) + 0$ shift
$\Rightarrow (0 + 1) + 0$ reduce
$= (0 + 1) + 0$ shift
$= (0 + 1) + 0$ shift

Example: $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle)$

$\langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$ reduce
$\Rightarrow \langle \text{Sum} \rangle + 0$ reduce
$= \langle \text{Sum} \rangle + 0$ shift
$= \langle \text{Sum} \rangle + 0$ shift
$\Rightarrow (\langle \text{Sum} \rangle) + 0$ reduce
$\Rightarrow (\langle \text{Sum} \rangle + 0) + 0$ reduce
$= (\langle \text{Sum} \rangle + 0 + 0)$ shift
$= (\langle \text{Sum} \rangle + 0 + 0)$ shift
$\Rightarrow (0 + 0 + 0)$ reduce
$= (0 + 0 + 0)$ shift
$= (0 + 0 + 0)$ shift

Example

$(0 + 1) + 0$

Example

$(0 + 1) + 0$
LR Parsing Tables

- Build a pair of tables, Action and Goto, from the grammar
  - This is the hardest part, we omit here
  - Rows labeled by states
  - For Action, columns labeled by terminals and “end-of-tokens” marker
    - (more generally strings of terminals of fixed length)
  - For Goto, columns labeled by non-terminals
Action and Goto Tables

- Given a state and the next input, Action table says either
  - **shift** and go to state \( n \), or
  - **reduce** by production \( k \) (explained in a bit)
  - **accept** or **error**

- Given a state and a non-terminal, Goto table says
  - go to state \( m \)

---

LR(i) Parsing Algorithm

- **Based on push-down automata**
- **Uses states and transitions** (as recorded in Action and Goto tables)
- **Uses** a stack containing states, terminals and non-terminals

---

0. Insure token stream ends in special “end-of-tokens” symbol
1. Start in state 1 with an empty stack
2. Push state(1) onto stack
3. Look at next \( i \) tokens from token stream \( (toks) \) (don’t remove yet)
4. If top symbol on stack is state(\( n \)), look up action in Action table at \( (n, toks) \)
5. If action = **shift** \( m \),
   a) Remove the top token from token stream and push it onto the stack
   b) Push state(\( m \)) onto stack
   c) Go to step 3
6. If action = **reduce** \( k \) where production \( k \) is \( E ::= u \)
   a) Remove 2 * length(\( u \)) symbols from stack (\( u \) and all the interleaved states)
   b) If new top symbol on stack is state(\( m \)), look up new state \( p \) in Goto(\( m,E \))
   c) Push \( E \) onto the stack, then push state(\( p \)) onto the stack
   d) Go to step 3
7. If action = **accept**
   - Stop parsing, return success
8. If action = **error**
   - Stop parsing, return failure
Adding Synthesized Attributes

- Add to each `reduce` a rule for calculating the new synthesized attribute from the component attributes.
- Add to each non-terminal pushed onto the stack, the attribute calculated for it.
- When performing a `reduce`,
  - gather the recorded attributes from each non-terminal popped from stack
  - Compute new attribute for non-terminal pushed onto stack.

Shift-Reduce Conflicts

- **Problem**: can’t decide whether the action for a state and input character should be `shift` or `reduce`.
- Caused by ambiguity in grammar.
- Usually caused by lack of associativity or precedence information in grammar.

Example: `<Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>`

- `0 + 1 + 0` shift
- `0 + 1 + 0` reduce
- `<Sum> 1 + 0` shift
- `<Sum> + 1 + 0` shift
- `<Sum> + 1 + 0` reduce
- `<Sum> + <Sum> + 0` shift

Example - cont

- **Problem**: `shift` or `reduce`?
- You can `shift-shift-reduce-reduce` or `reduce-shift-shift-reduce`.
- Shift first - right associative.
- Reduce first - left associative.

Reduce - Reduce Conflicts

- **Problem**: can’t decide between two different rules to reduce by.
- Again caused by ambiguity in grammar.
- **Symptom**: RHS of one production suffix of another.
- Requires examining grammar and rewriting it.
- Harder to solve than `shift-reduce` errors.

Example

- `S ::= A | aB`  `A ::= abc`  `B ::= bc`
- `abc` shift
- `a bc` shift
- `ab c` shift
- `abc` shift

- **Problem**: reduce by `B ::= bc` then by `S ::= aB`, or by `A ::= abc` then `S::A`?