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 Derivations
<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

Start with the start symbol:
<Sum> =>

BNF Derivations
<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

Pick a non-terminal:
<Sum> =>

BNF Derivations
<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

Pick a rule and substitute:
<Sum> ::= <Sum> + <Sum>
<Sum> => <Sum> + <Sum >

BNF Derivations
<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

Pick a non-terminal:
<Sum> => <Sum> + <Sum >

BNF Derivations
<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)

Pick a rule and substitute:
<Sum> ::= ( <Sum> )
<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>
BNF Derivations

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

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

BNF Derivations

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

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

BNF Derivations

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

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

BNF Derivations

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

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

BNF Derivations

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

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

- Pick a rule and substitute
  - \(<\text{Sum}> \::= 0\) \\
  - \(<\text{Sum}> \::= 1\) \\
  - \(<\text{Sum}> + <\text{Sum}> \) \\
  - \((<\text{Sum}> + <\text{Sum}> ) + <\text{Sum}> \) \\
  - \((<\text{Sum}> + 1) + <\text{Sum}> \) \\
  - \((<\text{Sum}> + 1)0\) \\
  - \((0 + 1) + 0\)

BNF Derivations

- \(<\text{Sum}> \::= 0 | 1 | <\text{Sum}> + <\text{Sum}> | (<\text{Sum}> )\)
- \((0 + 1) + 0\) is generated by grammar

- \(<\text{Sum}> \::= <\text{Sum}> + <\text{Sum}>\) \\
- \((<\text{Sum}> ) + <\text{Sum}>\) \\
- \((<\text{Sum}> + <\text{Sum}> ) + <\text{Sum}>\) \\
- \((<\text{Sum}> + 1) + <\text{Sum}>\) \\
- \((<\text{Sum}> + 1) + 0\) \\
- \((0 + 1) + 0\)

Regular Grammars

- Subclass of BNF
- Only rules of form \\
  - \(<\text{nonterminal}> ::= <\text{terminal}> <\text{nonterminal}>\) or \\
  - \(<\text{nonterminal}> ::= <\text{terminal}>\) or \\
  - \(<\text{nonterminal}> ::= \varepsilon\)
- Defines same class of languages as regular expressions
- Important for writing lexers (programs that convert strings of characters into strings of tokens)

Extended BNF Grammars

- Alternatives: allow rules of form \(X::=y | z\)
  - 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 non-terminal \(V\) and rules \\
  - \(X::=yz, X::=vVz, V::=v, V::=vV\)

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:
  - \(<\text{exp}> ::= <\text{factor}>\) \\
  - \(<\text{factor}> ::= <\text{bin}>\) \\
  - \(<\text{bin}> ::= 0 | 1\)
- Problem: Build parse tree for \(1 * 1 + 0\) as an \(<\text{exp}>\)
Example cont.

- $1 * 1 + 0$: $<\text{exp}>$

$<\text{exp}>$ is the start symbol for this parse tree

Example cont.

- $1 * 1 + 0$: $<\text{exp}>

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

Example cont.

- $1 * 1 + 0$: $<\text{exp}>

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

Example cont.

- $1 * 1 + 0$: $<\text{exp}>

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

Example cont.

- $1 * 1 + 0$: $<\text{exp}>

\[
\text{Use rules: } <\text{bin}> ::= 1 \text{ or } 0
\]
Example cont.

- $1 \times 1 + 0: \quad \langle \text{exp} \rangle$

  - $\langle \text{factor} \rangle$
  - $\langle \text{bin} \rangle \times \langle \text{exp} \rangle$
    - $1$
    - $\langle \text{factor} \rangle \times \langle \text{factor} \rangle$
      - $\langle \text{bin} \rangle$
      - $\langle \text{bin} \rangle$
        - $1$
        - $0$

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:
  
  \[
  \langle \text{exp} \rangle ::= \langle \text{factor} \rangle | \langle \text{factor} \rangle + \langle \text{factor} \rangle \\
  \langle \text{factor} \rangle ::= \langle \text{bin} \rangle | \langle \text{bin} \rangle \times \langle \text{exp} \rangle \\
  \langle \text{bin} \rangle ::= 0 | 1
  \]

- Represent as Abstract Data Types:
  
  - type \text{exp} = \text{Factor2Exp} of \text{factor}
  - and \text{factor} = \text{Bin2Factor} of \text{bin}
  - and \text{bin} = \text{Zero} | \text{One}

Example cont.

- Can be represented as

  \[
  \text{Factor2Exp} (\text{Mult}(\text{One}, \\
  \quad \text{Plus}(\text{Bin2Factor One}, \\
  \quad \text{Bin2Factor Zero})))
  \]
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 BNFs for a language are ambiguous then the language is *inherently ambiguous*.

Example: Ambiguous Grammar

```
0 + 1 + 0
<Sum>                 <Sum>
  <Sum> + <Sum>  <Sum> + <Sum>
    0         0
    1                          1
```

Example

- What is the result for:
  \[ 3 + 4 \times 5 + 6 \]

Possible answers:

- \[41 = ((3 + 4) \times 5) + 6\]
- \[47 = 3 + (4 \times (5 + 6))\]
- \[29 = (3 + (4 \times 5)) + 6 = 3 + ((4 \times 5) + 6)\]
- \[77 = (3 + 4) \times (5 + 6)\]

Example

- What is the value of:
  \[ 7 - 5 - 2 \]

Possible answers:

- In Pascal, C++, SML assoc. left:
  \[7 - 5 - 2 = (7 - 5) - 2 = 0\]
- In APL, associate to right:
  \[7 - 5 - 2 = 7 - (5 - 2) = 4\]
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 language of G = language of G’
- Not always possible
- No algorithm in general

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)
- Replace old rules to use new non-terminals
- Rinse and repeat

Example

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

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

- \(<\text{Sum}> ::= 0 \mid 1 \mid <\text{Sum}> + <\text{Sum}> \mid (<\text{Sum}>)
- Becomes
  - \(<\text{Sum}> ::= <\text{Num}> \mid <\text{Num}> + <\text{Sum}>\)
  - \(<\text{Num}> ::= 0 \mid 1 \mid (<\text{Sum}>)

Operator Precedence

- Operators of highest precedence evaluated first (bind more tightly).
  - For instance multiplication (*) has higher precedence than addition (+)

Needs to be reflected in grammar

Precedence in Grammar

- Higher precedence translates to longer derivation chain
- Example:
  - \(<\text{exp}> ::= 0 \mid 1 \mid <\text{exp}> + <\text{exp}> \mid <\text{exp}> * <\text{exp}>
- Becomes
  - \(<\text{exp}> ::= <\text{mult}\_\text{exp}> \mid <\text{exp}> + <\text{mult}\_\text{exp}>\)
  - \(<\text{mult}\_\text{exp}> ::= <\text{id}> \mid <\text{mult}\_\text{exp}> * <\text{id}>\)
  - \(<\text{id}> ::= 0 \mid 1\)

Disambiguating a Grammar

- \(<\text{exp}> ::= 0\mid1\mid b<\text{exp}> \mid <\text{exp}>a \mid <\text{exp}>m<\text{exp}>
  - Want \textit{a} to have higher precedence than \textit{b}, which in turn has higher precedence than \textit{m}, and such that \textit{m} associates to the left.

- \(<\text{exp}> ::= <\text{exp}> m <\text{not}\_m> \mid <\text{not}\_m> \mid <\text{not}\_b\_m> \mid <\text{not}\_b\_m> a \mid 0 \mid 1\)

Disambiguating a Grammar – Take 2

- \(<\text{exp}> ::= 0\mid1\mid b<\text{exp}> \mid <\text{exp}>a \mid <\text{exp}>m<\text{exp}>
  - Want \textit{b} to have higher precedence than \textit{m}, which in turn has higher precedence than \textit{a}, and such that \textit{m} associates to the right.
Disambiguating a Grammar – Take 2

- `<exp>::= 0|1| b<exp> | <exp>a
  | <exp>m<exp>
- Want `b` has higher precedence than `m`, which in turn has higher precedence than `a`, and such that `m` associates to the right.
- `<exp>::= <no_a_m> | <no_m> m <no_a> | <exp>a
  | <no_a> ::= <no_a_m> | <no_a_m> m <no_a>
  | <no_m> ::= <no_a_m> | <exp>a
  | <no_a_m> ::= b <no_a_m> | 0 | 1

Moving On With Richer Expressions

- How do we extend the grammar to support nested additions, e.g., `1 * (0 + 1)`
  - `<exp>::= <factor>
    | <factor> + <exp>
  - `<factor>::= <bin>
    | <bin> * <exp>
  - `<bin>::= 0 | 1 | ( <exp> )

Disambiguating a Grammar – Take 3

- `<exp>::= 0|1| b<exp> | <exp>a
  | <exp>m<exp>
- Want `a` has higher precedence than `m`, which in turn has higher precedence than `b`, and such that `m` associates to the right.
- For you...

Moving On With Richer Expressions

- How do we extend the grammar to support other operations, subtraction and division?
  - `<exp>::= <factor>
    | <factor> + <exp>
  - `<factor>::= <bin>
    | <bin> * <factor>
    | <bin> / <factor>
  - `<bin>::= 0 | 1 | ( <exp> )`
Disambiguation Grammars – Dangling Else

stmt ::= ...
  | if ( expr ) stmt
  | if ( expr ) stmt else stmt

How can we parse
  if (e1) if (e2) s1 else s2  ?

Disambiguation Grammars – Overlapping

seq = ε | may_word  | word seq
may_word = ε | “word”

How do you parse “word”? And ε?

How do you fix it?

How do you know you have ambiguity?

The Ocaml parser generator (ocamlyacc) will report ambiguity in the grammar as “conflicts”:

Shift/reduce: Usually caused by lack of associativity or precedence information in grammar

Reduce/reduce: can’t decide between two different rules to reduce by; Not always clear what the problem is, but often right-hand side of one production is the suffix of another

We will explain what these conflicts mean next time!

Parser Code

Ocamlyacc is a parser generator for Ocaml
  - Similar generators exist for other languages
  - Search under: Yacc, Bison, Menhir...
  - Another family: Antlr
Input: high level specification (<grammar>.mly file)
Output: tokens (<grammar>.ml) and generated parser (<grammar>.ml)
  - <grammar>.ml defines a 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

<grammar>.mly File format:
%{
  <header>
%
  <declarations>
%
  <rules>
%
  <trailer>
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
- <trailer> similar. Possibly used to call parser

Ocamlyacc Input

- <grammar>.mly File format:
  %{
    <header>
  %}
  <declarations>
  <rules>

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

Ocamlyacc Input

- <grammar>.mly File format:
  %{
    <header>
  %}
  <declarations>
  <rules>

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 ...
A slight variation of what we’ve seen earlier:

\[
\begin{align*}
\text{Expr} & ::= \text{Term} \mid \text{Term} + \text{Expr} \mid \text{Term} – \text{Expr} \\
\text{Term} & ::= \text{Factor} \mid \text{Factor} * \text{Term} \mid \text{Factor} / \text{Term} \\
\text{Factor} & ::= \text{Id} \mid (\text{Expr})
\end{align*}
\]

Example - Base types

\[
\begin{align*}
\text{type expr} &= \text{Term}\_\text{As}\_\text{Expr} \text{of term} \\
& \quad \mid \text{Plus}\_\text{Expr} \text{of} (\text{term} * \text{expr}) \\
& \quad \mid \text{Minus}\_\text{Expr} \text{of} (\text{term} * \text{expr}) \\
\text{and term} &= \text{Factor}\_\text{As}\_\text{Term} \text{of factor} \\
& \quad \mid \text{Mult}\_\text{Term} \text{of} (\text{factor} * \text{term}) \\
& \quad \mid \text{Div}\_\text{Term} \text{of} (\text{factor} * \text{term}) \\
& \quad \mid \text{Id}\_\text{As}\_\text{Factor} \text{of string} \\
& \quad \mid \text{Parenthesized}\_\text{Expr}\_\text{As}\_\text{Factor} \text{of expr}
\end{align*}
\]

Example - Lexer

\[
\begin{align*}
\text{let numeric} &= ['0' - '9'] \\
\text{let letter} &= ['a' - 'z', 'A' - 'Z'] \\
\text{rule token} &= \text{parse} \\
& | '+' \{\text{Plus}\_\text{token}\} \\
& | '-' \{\text{Minus}\_\text{token}\} \\
& | '*' \{\text{Times}\_\text{token}\} \\
& | '/' \{\text{Divide}\_\text{token}\} \\
& | '(' \{\text{Left}\_\text{parenthesis}\} \\
& | ')' \{\text{Right}\_\text{parenthesis}\} \\
& | \text{letter}(\text{letter}|\text{numeric}|\_\text{)* as id }\{\text{Id}\_\text{token} \text{id}\} \\
& | \text{' '}{\text{tab} \text{' '}}\text{'n'} \{\text{token lexbuf}\} \\
& | \text{eof} \{\text{EOL}\}
\end{align*}
\]

Example - Parser (exprparse.mly)

\[
\begin{align*}
\text{expr:} & \\
& | \text{term} \{\text{Term}\_\text{as}\_\text{Expr} \$1\} \\
& \mid \text{term Plus_token expr} \\
& \quad \{\text{Plus}\_\text{Expr} (\$1, \$3)\} \\
& \mid \text{term Minus_token expr} \\
& \quad \{\text{Minus}\_\text{Expr} (\$1, \$3)\}
\end{align*}
\]

Example - Base types

\[
\begin{align*}
\text{type expr} &= \text{Term}\_\text{As}\_\text{Expr} \text{of term} \\
& \quad \mid \text{Plus}\_\text{Expr} \text{of} (\text{term} * \text{expr}) \\
& \quad \mid \text{Minus}\_\text{Expr} \text{of} (\text{term} * \text{expr}) \\
\text{and term} &= \text{Factor}\_\text{As}\_\text{Term} \text{of factor} \\
& \quad \mid \text{Mult}\_\text{Term} \text{of} (\text{factor} * \text{term}) \\
& \quad \mid \text{Div}\_\text{Term} \text{of} (\text{factor} * \text{term}) \\
& \quad \mid \text{Id}\_\text{As}\_\text{Factor} \text{of string} \\
& \quad \mid \text{Parenthesized}\_\text{Expr}\_\text{As}\_\text{Factor} \text{of expr}
\end{align*}
\]

Example - Base types

\[
\begin{align*}
\text{type expr} &= \text{Term}\_\text{As}\_\text{Expr} \text{of term} \\
& \quad \mid \text{Plus}\_\text{Expr} \text{of} (\text{term} * \text{expr}) \\
& \quad \mid \text{Minus}\_\text{Expr} \text{of} (\text{term} * \text{expr}) \\
\text{and term} &= \text{Factor}\_\text{As}\_\text{Term} \text{of factor} \\
& \quad \mid \text{Mult}\_\text{Term} \text{of} (\text{factor} * \text{term}) \\
& \quad \mid \text{Div}\_\text{Term} \text{of} (\text{factor} * \text{term})
\end{align*}
\]

Example - Parser (exprparse.mly)

\[
\begin{align*}
\text{term:} & \\
& | \text{factor} \{\text{Factor}\_\text{as}\_\text{Term} \$1\} \\
& \mid \text{factor Times_token term} \\
& \quad \{\text{Mult}\_\text{Term} (\$1, \$3)\} \\
& \mid \text{factor Divide_token term} \\
& \quad \{\text{Div}\_\text{Term} (\$1, \$3)\}
\end{align*}
\]
Example - Parser (exprparse.mly)

```ocaml
factor:
  | Id_token { Id_as_Factor $1 }
| Left_parenthesis expr Right_parenthesis {Parenthesized.Expr_as_Factor $2 }

main:
  | expr EOL { $1 }
```

Recall, we previously defined:

```ocaml
Example - Base types
(" File expr.ml *")
type expr =
  Term_as.Expr of term
| Plus.Expr of (term + expr) and term =
  Factor_as.Term of factor
| Mult.Term of (factor * term)
| (De.Term of (factor * term) and factor =
  Id_as.Factor of string
| Parenthesized.Expr as Factor of expr
```

Example - Using Parser

```ocaml
# 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

```ocaml
# test "a + b";;
- : expr =
  Plus.Expr
  (Factor_as.Term (Id_as.Factor "a"),
   Term_as.Expr
    (Factor_as.Term (Id_as.Factor "b")))
```

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

```
( 0 + 1 ) + 0
```

LR Parsing

General plan:

- 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:

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

Call:

```ocaml
$ ocamlyacc options exprparse.mly
```

Get:

- Tokens: exprparse.mli (can be used in lexer)
- Parser: exprparse.ml (included in the rest of code)
Example: `<Sum> ::= 0 | 1 | (<Sum>)
   | <Sum> + <Sum>`

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

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

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

LR(i) Parsing Algorithm

0. Insure token stream ends in special “end-of-tokens” symbol
1. Start in state 1 with an empty stack
2. Push $\text{state}(1)$ onto stack
3. Look at next $i$ tokens from token stream ($\text{toks}$) (don’t remove yet)
4. If top symbol on stack is $\text{state}(n)$, look up action in Action table at $(n, \text{toks})$
LR(i) Parsing Algorithm

5. If action = **shift** \( m \),
   a) Remove the top token from token stream and push it onto the stack
   b) Push \texttt{state}(m) onto stack
   c) Go to step 3

6. If action = **reduce** \( k \) where production \( k \) is 
   \( E ::= u \)
   a) Remove 2 \times \text{length}(u) symbols from stack (\( u \) and all the interleaved states)
   b) If new top symbol on stack is \texttt{state}(m), look up new state \( p \) in Goto(\( m \),E)
   c) Push \( E \) onto the stack, then push \texttt{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

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

\[ \Downarrow (0 + 1) + 0 \quad \text{shift} \]

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

\[ \Downarrow (0 + 1) + 0 \quad \text{shift} \]
LR(i) Parsing Algorithm

5. If action = \textbf{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

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

\( \text{<Sum>} \Rightarrow \)
\[ \Rightarrow (0 \ast + 1) + 0 \quad \text{reduce} \]
\[ = (\ast 0 + 1) + 0 \quad \text{shift} \]
\[ = \ast (0 + 1) + 0 \quad \text{shift} \]

LR(i) Parsing Algorithm

6. If action = \textbf{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

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

\( \text{<Sum>} \Rightarrow \)
\[ \Rightarrow (\text{<Sum>} \ast + 1) + 0 \quad \text{shift} \]
\[ \Rightarrow (0 \ast + 1) + 0 \quad \text{reduce} \]
\[ = (\ast 0 + 1) + 0 \quad \text{shift} \]
\[ = \ast (0 + 1) + 0 \quad \text{shift} \]
Example: \(<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)
| <\text{Sum}> + <\text{Sum}>\)

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

\[= (\text{<Sum>} + 1 \bullet) + 0\] reduce
\[= (\text{<Sum>} + \bullet 1) + 0\] shift
\[= (\text{<Sum>} \bullet + 1) + 0\] shift
\[= (0 \bullet + 1) + 0\] reduce
\[= (0 + 1) + 0\] shift
\[= \bullet (0 + 1) + 0\] shift

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

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

\[= (\text{<Sum>} + \text{<Sum>}) \bullet + 0\] reduce
\[= (\text{<Sum>} + 1 \bullet) + 0\] reduce
\[= (\text{<Sum>} + \bullet 1) + 0\] reduce
\[= (\text{<Sum>} \bullet + 1) + 0\] reduce
\[= (\text{<Sum>} \bullet + 1) + 0\] reduce
\[= (\text{<Sum>} \bullet + 1) + 0\] reduce
\[= (\text{<Sum>} \bullet + 1) + 0\] reduce
\[= (\text{<Sum>} \bullet + 1) + 0\] reduce
\[= (\text{<Sum>} \bullet + 1) + 0\] reduce
\[= (\text{<Sum>} \bullet + 1) + 0\] reduce
\[= (\text{<Sum>} \bullet + 1) + 0\] reduce

LR(i) Parsing Algorithm

6. If action = reduce \(k\) where production \(k\) is \(E ::= u\)
   a) Remove \(2 \times\) length\((u)\) symbols from stack \((u\) and all the interleaved states\)
   b) If new top symbol on stack is \text{state}(m), look up new state \(p\) in \text{Goto}(m;E)
   c) Push \(E\) onto the stack, then push \text{state}(p) onto the stack
   d) Go to step 3
Example: \( \text{<Sum>} = 0 \mid 1 \mid (\text{<Sum>}) \) \\
| \( \text{<Sum>} + \text{<Sum>} \) \\
\[
\begin{align*}
\text{<Sum>} & \Rightarrow \\
& \Rightarrow \text{<Sum>} + \text{<Sum>} \quad \text{reduce} \\
& \Rightarrow \text{<Sum>} + 0 \quad \text{reduce} \\
& \Rightarrow \text{<Sum>} + 0 \quad \text{shift} \\
& \Rightarrow (\text{<Sum>} + 0) + 0 \quad \text{reduce} \\
& \Rightarrow (\text{<Sum>} + 1) + 0 \quad \text{reduce} \\
& \Rightarrow (\text{<Sum>} + 0 + 1) + 0 \quad \text{shift} \\
& \Rightarrow (0 + 0 + 1) + 0 \quad \text{reduce} \\
& \Rightarrow (0 + 1 + 0) + 0 \quad \text{reduce} \\
& \Rightarrow (0 + 1) + 0 \quad \text{reduce} \\
& \Rightarrow (0 + 1) + 0 \quad \text{shift} \\
& \Rightarrow (0 + 1) + 0 \quad \text{shift}
\end{align*}
\]

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

LR(i) Parsing Algorithm

7. If action = **accept**
   - Stop parsing, return success

8. If action = **error**,
   - Stop parsing, return failure

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
LR(i) Parsing Algorithm

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 / 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: $\langle \text{Sum} \rangle = 0 \ | \ 1 \ | \ (\langle \text{Sum} \rangle)$

<table>
<thead>
<tr>
<th>$\langle \text{Sum} \rangle + \langle \text{Sum} \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 + 1 + 0 shift</td>
</tr>
<tr>
<td>0 + 1 + 0 reduce</td>
</tr>
<tr>
<td>$\langle \text{Sum} \rangle$ + 1 + 0 shift</td>
</tr>
<tr>
<td>$\langle \text{Sum} \rangle$ + 1 + 0 reduce</td>
</tr>
<tr>
<td>$\langle \text{Sum} \rangle$ + $\langle \text{Sum} \rangle$ + 0 reduce</td>
</tr>
</tbody>
</table>

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

- $S ::= A \mid aB$
  - $A ::= abc$
  - $B ::= bc$

  | $abc$ shift |
  | $a \bullet bc$ shift |
  | $ab \bullet c$ shift |
  | $abc \bullet$ |

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