Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

---

**Parser Code**

- `<grammar>.ml` 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>`

---

**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

---

**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 `<declarations>`**

- `%type <type> symbol ... symbol`
  - Specify type of attributes for given symbols. Mandatory for start symbols
- `%left symbol ... symbol`
- `%right symbol ... symbol`
- `%nonassoc symbol ... symbol`
  - Associate precedence and associativity to given symbols. Same line, same precedence; earlier line, lower precedence (broadest scope)
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 - Base types

(* File: expr.ml *)

```ocaml
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
```

Example - Lexer (exprlex.mll)

```ocaml
{"open Exprparse*} }
let numeric = ['0' - '9']
let letter =['a' - 'z' 'A' - 'Z']
rule token = parse
  | "+" {Plus_token}
  | "+" {Minus_token}
  | "*" {Times_token}
  | "/" {Divide_token}
  | "$" {Left_parenthesis}
  | "}" {Right_parenthesis}
  | letter (letter|numeric|"_")* as id {Id_token id}
  | [" 	"
  | eol {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)

```ocaml
expr:
  term
  { Term_as_Expr $1 }
  | term Plus_token expr
  { Plus_Expr ($1, $3) }
  | term Minus_token expr
  { Minus_Expr ($1, $3) }
```

Example - Parser (exprparse.mly)

```ocaml
term:
  factor
  { Factor_as_Term $1 }
  | factor Times_token term
  { Mult_Term ($1, $3) }
  | factor Divide_token term
  { Div_Term ($1, $3) }
```
Example - Parser (exprparse.mly)

factor:
  Id_token
  { Id_as_Factor $1 }
| Left_parenthesis expr Right_parenthesis
  {Parenthesized_Expr_as_Factor $2 }
main:
| expr EOL
  { $1 }

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_Factor "a"),
     Term_as_Expr (Factor_as_Term (Id_as_Factor "b")))

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

More Disambiguating Grammars

M ::= M * M | ( M ) | M ++ | 6

Ambiguous because of associativity of *
because of conflict between * and ++:

\[ 6 \times 6 ++ \quad 6 \times 6 ++ \]
\[ M \quad M \quad M \* M \]
\[ M \* M \quad 6 \quad M ++ \]
\[ 6 \quad 6 \quad 6 \]
How to disambiguate?
Choose associativity for *
Choose precedence between * and ++
Four possibilities
Four different approaches
Some easier than others
Will do --- You choose
\[ M ::= M \ast M \mid (M) \mid M \text{++} \mid 6 \]

* higher prec than ++
* Left assoc
M ::= M++ | StarExp | (M) | 6
StarExp ::= M * NoStarNoPlusPlus
NoStarNoPlusPlus ::= (M) | 6
But we have (M) | 6 twice, and it’s the same language each time. Let’s have one.

\[ M ::= M \ast M \mid (M) \mid M \text{++} \mid 6 \]

* higher prec than ++
* Left assoc
M ::= M++ | StarExp | NoStarNoPlusPlus
StarExp ::= M * NoStarNoPlusPlus
NoStarNoPlusPlus ::= (M) | 6

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

\[
<\text{Sum}> &= \]
\[
= \circ (0 + 1) + 0 \text{ shift}
\]
Example: \(<\text{Sum}\> = 0 | 1 | (<\text{Sum}\>)
| <\text{Sum}\> + <\text{Sum}\>)

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

\( \rightarrow (0 + 1) + 0 \) reduce
\( = (0 + 1) + 0 \) shift
\( = (0 + 1) + 0 \) shift

11/1/22
Example: $<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)$

$<\text{Sum}> =;$

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

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

$<\text{Sum}> =;$

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

( 0 + 1 ) + 0

Example

<Sum>

( 0 + 1 ) + 0

Example

<Sum>

( 0 + 1 ) + 0

Example

<Sum>

( 0 + 1 ) + 0
Example

\[
\left( \begin{array}{cc}
0 & \text{Sum} \\
\text{Sum} & 1
\end{array} \right) + 0
\]

Example

\[
\left( \begin{array}{cc}
0 & \text{Sum} \\
\text{Sum} & 1
\end{array} \right) + 0
\]

Example

\[
\left( \begin{array}{cc}
0 & \text{Sum} \\
\text{Sum} & 1
\end{array} \right) + 0
\]

Example

\[
\left( \begin{array}{cc}
0 & \text{Sum} \\
\text{Sum} & 1
\end{array} \right) + 0
\]

Example

\[
\left( \begin{array}{cc}
0 & \text{Sum} \\
\text{Sum} & 1
\end{array} \right) + 0
\]
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
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 $i$ 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: \(<\text{Sum}> = 0 | 1 | (\langle\text{Sum}\rangle)\)  
| \(<\text{Sum}> + \langle\text{Sum}\rangle\)  
- 0 + 1 + 0 shift  
- \(\langle\text{Sum}\rangle 0 + 1 + 0\) reduce  
- \(\langle\text{Sum}\rangle 0 + 1 + 0\) shift  
- \(\langle\text{Sum}\rangle + 1 0 + 0\) reduce  
- \(\langle\text{Sum}\rangle + \langle\text{Sum}\rangle 0 + 0\)

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

- \(S ::= A | aB\)  
- \(A ::= abc\)  
- \(B ::= bc\)

\[
\begin{align*}
\text{abc} & \quad \text{shift} \\
\text{a} \quad \text{bc} & \quad \text{shift} \\
\text{ab} \quad \text{c} & \quad \text{shift} \\
\text{abc} & \\
\end{align*}
\]

- Problem: reduce by \(B ::= bc\) then by \(S ::= aB\), or by \(A ::= abc\) then \(S ::= A\)?