

Lecture 7-8 — Context-free grammars and bottom-up parsing

- Grammars and languages
- Ambiguity and expression grammars
- Parser generators
 - Bottom-up (shift/reduce) parsing
 - `ocamlyacc` example
 - Handling ambiguity in `ocamlyacc`

Grammar for (almost) MiniJava

```
Program -> ClassDeclList
ClassDecl -> class id { VarDeclList MethodDeclList }
VarDecl -> Type id ;
MethodDecl -> Type id ( FormalList ) { VarDeclList StmtList return Exp ; }
Formal -> Type id
Type -> int [ ] | boolean | int | id
Stmt -> { StmtList } | if ( Exp ) Stmt else Stmt
      | while ( Exp ) Stmt | System.out.println ( Exp ) ;
      | id = Exp ; | id [ Exp ] = Exp ;
Exp -> Exp Op Exp | Exp [ Exp ] | Exp . length
      | Exp . id ( ExpList ) | integer | true | false | id
      | this | new int [ Exp ] | new id ( ) | ! Exp | ( Exp )
Op -> && | < | + | - | *
ExpList -> Exp ExpRest |
ExpRest -> , Exp ExpRest |
FormalList -> Type id FormalRest |
FormalRest -> , Type id FormalRest |
ClassDeclList = ClassDeclList VarDecl |
MethodDeclList = MethodDeclList MethodDecl |
VarDeclList = VarDeclList VarDecl |
StmtList = StmtList Stmt |
```

Grammars and languages

- E.g. **program** `class C {int f () { return 0; }}` has this parse tree:
- Parser converts source file to parse tree. AST is easily calculated from parse tree, or constructed while parsing (without explicitly constructing parse tree).

Context-free grammar (cfg) notation

- **CFG** is a set of productions $A \rightarrow X_1X_2 \dots X_n$ ($n \geq 0$). If $n = 0$, we may write either $A \rightarrow$ or $A \rightarrow \epsilon$.
- $A, B, C, \dots \in N$, the set of *non-terminals*. One non-terminal in the grammar is the *start symbol*.
- T is the set of *tokens*, aka *terminals*
- $X, Y, Z \in S = N \cup T$, the set of *grammar symbols*
- $u, v, w \in T^*$
- $\alpha, \beta, \gamma \in S^*$
- **Productions** $A \rightarrow \alpha, A \rightarrow \beta, \dots$, abbreviated as $A \rightarrow \alpha \mid \beta \mid \dots$
- **A parse tree, or (concrete) syntax tree, from A is a**

tree whose root is labelled A , and whose internal nodes are labelled with non-terminals such that if a node is labelled B , its children are labelled $X_1X_2 \dots X_n$, for some production $B \rightarrow X_1X_2 \dots X_n$. As a special case, if a node is labelled B , it may have a child labelled “•”, if B has an ϵ -production.

- An *A-form* of a grammar is any frontier of a parse tree (i.e. labels of the leaf nodes) from A , with •’s deleted. An *A-sentence* is an *A-form* in T^* .
- *Sentential form*, *sentence*, and *parse tree* refer to, respectively, an *S-form*, *S-sentence*, and parse tree from S , where S is the start symbol.
- $A \in N$ is *nullable* if ϵ is an *A-sentence*.
- An *ambiguous grammar* is one for which at least one sentence has more than one parse tree.

- A *recognizer* is a function of type $\text{string} \rightarrow \text{bool}$ that says whether a string is a sentence. A *parser* is a recognizer that also constructs a parse tree (or AST). (In practice, a recognizer is usually easy to convert to a parser, by adding tree-constructing operations at appropriate places.)
- An *extended cfg* is one whose right-hand sides can contain regular expression operations $*$, $+$, and $?$. These are used to abbreviate ordinary context-free rules:
 - β^* should be replaced by a new non-terminal, say B , and additional rules $B \rightarrow \epsilon \mid \beta B$ (or, equivalently, $B \rightarrow \epsilon \mid B\beta$)
 - β^+ should be replaced by a new non-terminal, say B , and additional rules $B \rightarrow \beta \mid \beta B$ (or, equivalently, $B \rightarrow \beta \mid B\beta$)
 - $\beta?$ should be replaced by a new non-terminal, say B , and additional rules $B \rightarrow \beta \mid \epsilon$.

Exercises on cfg notation

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * P \mid P$$

$$P \rightarrow id \mid int \mid (P)$$

- **Parse tree for: $x * 10 + y$:**

- **Give examples of each:**
 - **E-sentence:**
 - **T-sentence:**
 - **E-sentential form that is not a sentence:**

More exercises on cfg notation

$$E \rightarrow EA$$

$$A \rightarrow \epsilon \mid A + T$$

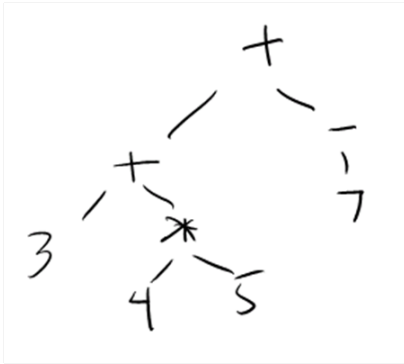
$$T \rightarrow T(*P)^*$$

$$P \rightarrow id \mid int \mid (P)$$

- Transform this grammar to remove the Kleene star.
- Show that both $E+T$ and $EA+T$ are sentential forms:

Expression grammars

- Recall that the expression $3 + 4 * 5 + -7$ has AST



The shape of this AST represents the precedence of multiplication and the left-associativity of addition. It ensures, for example, that `eval` would return the correct value.

- Parsing produces a parse tree which is translated to an AST. It simplifies this translation greatly if the shape of the concrete syntax tree correctly represents precedences and associativities of operators.

Some expression grammars

$G_A: E \rightarrow \text{id} \mid E - E \mid E * E$

$G_B: E \rightarrow \text{id} \mid \text{id} - E \mid \text{id} * E$

$G_C: E \rightarrow \text{id} \mid E - \text{id} \mid E * \text{id}$

$G_D: \begin{array}{l} E \rightarrow T - E \mid T \\ T \rightarrow \text{id} \mid \text{id} * T \end{array}$

$G_E: \begin{array}{l} E \rightarrow E - T \mid T \\ T \rightarrow \text{id} \mid T * \text{id} \end{array}$

$G_F: \begin{array}{l} E \rightarrow T E' \\ E' \rightarrow \epsilon \mid - E \\ T \rightarrow \text{id} T' \\ T' \rightarrow \epsilon \mid * T \end{array}$

● $G_A: E \rightarrow \text{id} \mid E - E \mid E * E$

● $x - y * z$

$x - y - z$

● **Ambiguous?**

Precedence?

Associativity?

● $G_B: E \rightarrow \text{id} \mid \text{id} - E \mid \text{id} * E$

● $x - y * z$

$x - y * z - w$

● $x * y - z$

● Ambiguous?

Precedence?

Associativity?

● $G_C: E \rightarrow \text{id} \mid E - \text{id} \mid E * \text{id}$

● $x - y * z$

$x - y * z - w$

● $x * y - z$

● Ambiguous?

Precedence?

Associativity?

● $G_D: E \rightarrow T - E \mid T$
 $T \rightarrow \text{id} \mid \text{id} * T$

● $x - y * z$

$x * y - z$

$x - y - z$

● **Ambiguous?**

Precedence?

Associativity?

- $G_E: E \rightarrow E - T \mid T$
 $T \rightarrow \text{id} \mid T * \text{id}$

- $x - y * z$

$$x * y - z$$

$$x - y - z$$

- **Ambiguous?**

Precedence?

Associativity?

● $G_F: E \rightarrow T E'$
 $E' \rightarrow \epsilon \mid - E$
 $T \rightarrow \text{id } T'$
 $T' \rightarrow \epsilon \mid * T$

● $x - y * z$

$x * y - z$

$x - y - z$

● **Ambiguous?**

Precedence?

Associativity?

Parser generators

- Like lexer generators, these are programs that input a specification — in the form of a context-free grammar, with an action associated with each production — and output a parser.
- The most famous of all parser generators is `yacc` — which, ironically, stands for “yet another compiler compiler.” Originally written to generate parsers in C, it has been copied in many other languages. We will use `ocamlyacc`.
- We start with a small but complete example.
- Next week, we will discuss how to write a parser by hand, using the method of *recursive descent*.

Example - expression grammar

- In this example, we will use ocaml yacc to create a parser for this grammar:

$$\begin{aligned} M &\rightarrow E \text{ eof} \\ E &\rightarrow T \mid E + T \mid E - T \\ T &\rightarrow P \mid T * P \mid T / P \\ P &\rightarrow \text{id} \mid (E) \end{aligned}$$

- It will product ASTs of type exp:

```
(* File: exp.ml *)
type exp =
  Plus of exp * exp
  | Minus of exp * exp
  | Mult of exp * exp
  | Div of exp * exp
  | Id of string
```

Example - exprlex.mll

```
{ type token = PlusT | MinusT | TimesT | DivideT
      | OParenT | CParenT | IdT of string | EOF }

let numeric = ['0' - '9']
let letter = ['a' - 'z' 'A' - 'Z']

rule tokenize = parse
  | "+"          {PlusT}
  | "-"          {MinusT}
  | "*"          {TimesT}
  | "/"          {DivideT}
  | "("          {OParenT}
  | ")"          {CParenT}
  | letter (letter | numeric | "_")* as id      {IdT id}
  | [', ' '\t' '\n']      {tokenize lexbuf}
  | eof              {EOF}
```

Example - exprparse.mly

```
%token <string> IdT
%token OParenT CParenT TimesT DivideT PlusT MinusT EOF
%start main
%type <exp> main
%%
expr:
    term                                {$1}
  | expr PlusT term                    {Plus($1,$3)}
  | expr MinusT term                   {Minus($1,$3)}

term:
    factor                             {$1}
  | term TimesT factor                {Mult($1,$3)}
  | term DivideT factor               {Div($1,$3)}

factor:
    IdT                               {Id $1}
  | OParenT expr CParenT              {$2}

main:
    expr EOF {$1}
```

Shift-reduce parsing

- `ocamlyacc` uses a method of parsing known as *shift/reduce*, a.k.a. *bottom-up parsing*. Here's how it works:
 - Keep a stack of grammar symbols (initially empty). Based on this stack, and the next input token (“lookahead symbol”), take one of these actions:
 - Shift: Move lookahead symbol to stack
 - Reduce $A \rightarrow \alpha$: Symbols on top of stack are α ; replace them by A . (*If you create a tree node here, you can construct the parse tree while parsing.*)
 - Accept: When stack consists of just the start symbol, and input is exhausted
 - Reject

Shift-reduce example 1

- $L \rightarrow L ; E \mid E$
 $E \rightarrow id$

Input: x; y

Shift-reduce example 2

- $E \rightarrow E + T \mid T$
 $T \rightarrow T * P \mid P$
 $P \rightarrow id \mid int$

Input: $x + 10 * y$

Parsing conflicts

- Parsers for programming languages must be very efficient, but no efficient method for parsing arbitrary cfg's is known. So all parser generators accept only certain grammars. The hardest part of using any parser generator is getting the grammar into a form the parser generator will accept.
- yacc (and ocaml yacc) accept a class of grammars known as *LALR(1) grammars*. We will not be able to describe exactly what distinguishes this class; that is done in CS 426.
- If you present a non-LALR(1) grammar to ocaml yacc, it will report an error message, called a *conflict*.
- To understand how to deal with conflicts, we need to look at shift-reduce parsing a little closer.

S/R parsing $\equiv$ parse tree

- Each shift-reduce parse — i.e. each sequence of s/r actions — produces a unique parse tree.
- Every parse tree is built by a unique s/r parse:
 - Traverse the tree in post-order. A leaf node corresponds to a shift action, and an internal node to a reduce action.
- An LALR(1) parser generator will accept a grammar only if it can determine, based on the stack and lookahead symbol, the unique correct action.
- It follows that ambiguous grammars can never be LALR(1).

Shift-reduce example 3

- **Grammar:** $E \rightarrow E + E \mid E * E \mid id$
Input: $x + y + z$

Show a parse tree, and corresponding s/r parse, that represents left-associativity of addition.

Shift-reduce example 3 (cont.)

- **Grammar:** $E \rightarrow E + E \mid E * E \mid id$
Input: $x + y + z$

Show a parse tree, and corresponding s/r parse, that represents right-associativity of addition.

Dealing with ambiguity

- **yacc has a special trick for dealing with ambiguity: annotations telling the parser explicitly what to do in some cases.**
- **For the previous grammar, there are four interesting inputs: $x+y+z$, $x*y*z$, $x+y*z$, $x*y+z$.**
- **Consider $x+y+z$. It has two parse trees. For both, the stack looks the same until the second $+$ is the lookahead symbol.**
- **What is the right decision?**

Dealing with ambiguity (cont.)

- For $x*y*z$, consider where the two stack configurations that can occur for the two parse trees differ. What is the correct decision?
- Do the same for $x+y*z$:
- and for $x*y+z$:

Precedence declarations

- Looking at these four cases, we can see that they will be parsed correctly if we follow these rules:
 - If the operator nearest the top of the stack and the lookahead symbol have the same precedence, then *shift* if the operator is right-associative, and *reduce* if it is left-associative.
 - If the operator nearest the top of the stack has *higher* precedence than the lookahead symbol, then *reduce*; otherwise, *shift*
- `ocaml yacc` will follow these rules if you tell it which operators have higher precedence, and which are left- or right-associative. Do that using *precedence declarations*...

Precedence declarations (cont.)

- Precedence declarations are added to the `ocaml yacc` specification after the `%token` declarations. Syntax:
 - `%left symbol ... symbol`
 - `%right symbol ... symbol`
 - `%nonassoc symbol ... symbol`
- Two symbols appearing in the same precedence declaration have the same precedence and the given associativity. (If they appear in a `%nonassoc` declaration, they cannot follow one another, e.g. $x < y < z$.)
- Two symbols appearing in different precedence declarations have different precedences: the one that comes earlier has lower precedence.

Precedence declaration example

- Recall this ocaml yacc specification:

```
expr:
    term                                {$1}
  | term PlusT expr                    {Plus($1,$3)}
  | term MinusT expr                   {Minus($1,$3)}

term:
    factor                             {$1}
  | factor TimesT term                 {Mult($1,$3)}
  | factor DivideT term                {Div($1,$3)}

factor:
    IdT                               {Id $1}
  | OParenT expr CParenT              {$2}

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

Precedence declarations (cont.)

- This can be simplified with precedence declarations (after the %token declarations):

```
%left PlusT MinusT
%left TimesT DivideT
%start main
%type <expr> main
%%
expr:
    IdT                {Id $1}
  | expr PlusT expr    {Plus($1,$3)}
  | expr MinusT expr   {Minus($1,$3)}
  | expr TimesT expr   {Plus($1,$3)}
  | expr DivideT expr  {Minus($1,$3)}
  | OParenT expr CParenT {$2}

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

Debugging ocaml yacc specifications

- In doing MP4, the main question will be: what operators are causing conflicts? Once you've identified them, you can add precedence declarations.
- When you run `ocaml yacc`, it will report the number of conflicts. Running with the `-v` option produces a file with the extension `.output`, containing details.
- E.g. grammar `Expr  $\rightarrow$  Expr + Expr | id` has a conflict. Search for “conflict” in the `.output` file:

```
6: shift/reduce conflict (shift 5, reduce 1) on plus
state 6
Expr : Expr . plus Expr (1)
Expr : Expr plus Expr . (1)
```

This says that there is a problem with the `plus` token.

Constructing AST's

- Precedence rules make it easier to construct AST's, because concrete syntax is closer to abstract syntax.
- Different non-terminals can produce different *types* of values. An important case is “list-like” syntax categories. E.g. consider this grammar:

$$\begin{aligned} funcall &\rightarrow id \ (\ arglist \) \\ arglist &\rightarrow funcall \ arglistrest \ | \\ arglistrest &\rightarrow , \ funcall \ arglistrest \ | \end{aligned}$$

- Suppose our abstract syntax is

```
type funcall = Funcall of string * (funcall list)
```

● Here is how to do this in ocaml yacc:

```
funcall:
    IdT OParenT arglist CParenT    { Funcall($1, $3) }
arglist:
    funcall arglistrest            { $1 :: $2 }
    |                              { [] }
arglistrest:
    CommaT funcall arglistrest     { $2 :: $3 }
    |                              { [] }
```

Bonus topic: A little LR theory...

- The shift-or-reduce decision seems very mysterious: We know what to do when we already have the parse tree, but how can we know, based only on the grammar, what will be the correct action in every case?
- We can start by looking at the “stack configurations” of s/r parses. Define $SC(G) = \{ \alpha \in S^* \mid \alpha \text{ can be the stack in a shift-reduce parse for } G \}$.
- Consider this example: $A \rightarrow \text{id} \mid (A)$

More examples of $SC(G)$

$$E \rightarrow E + T \mid T$$
$$T \rightarrow \text{id}$$
$$E \rightarrow T + E \mid T$$
$$T \rightarrow \text{id}$$

A little LR theory (cont.)

- **Theorem [Knuth]** For any grammar G , $SC(G)$ is a finite-state language over S .
- yacc starts by constructing the “characteristic DFA” for the specified grammar. To parse a sentence, repeat the following:
 - Take the stack and concatenate the lookahead symbol. If the result is in $SC(G)$, then shift; o/w reduce.
- Actually, constructing the characteristic DFA is just the start. The simple parsing method just given does not always work. (E.g. even when it says to reduce, it may not be clear which production to reduce by.) The full construction of the parser is quite involved; see CS426, or a compiler textbook.