

CS421 Lecture 11: Parsing¹

Mark Hills

`mhills@cs.uiuc.edu`

University of Illinois at Urbana-Champaign

June 23, 2008

¹Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

Objectives and Review

Quick review

Parsing

Parser Generation with OCaml

Objectives

At the end of this lecture, you should be able to

- ▶ Explain two different style of parsing
 - ▶ Top-down parsing
 - ▶ Bottom-up parsing
- ▶ Understand when you would use one versus the other
- ▶ Use `ocamlyacc` to write parser definitions

Context-free Grammars

Def: A **Context-free Grammar** (CFG) is a 4-tuple:

$$G = (N, \Sigma, P, S)$$

where:

1. N is the set of nonterminals
2. Σ is the set of terminals (tokens)
3. P is the set of productions (rules)
4. S is the start symbol, and is one of the nonterminals in N

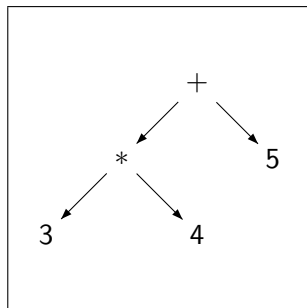
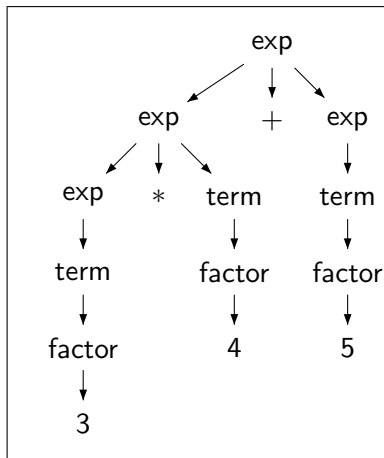
Derivations and Parse Trees

- ▶ A **Derivation** is a sequence of steps that, starting with the start symbol, leads to a sentence in the language (a string with only terminals). Each derivation step replaces one nonterminal with the right-hand side of one production.
- ▶ A **Parse Tree** is a tree representation of a derivation, with nonterminals as nodes and either nonterminals or terminals as leaves (only terminals for a derivation, both while building the tree). A node's children are based directly on the terminals and nonterminals on the right-hand side of the rule used to construct them.

AST versus Parse Tree

While a parse tree keeps track of all productions used to build the parse tree, an AST is a condensed form of this with just the information needed in later stages of compilation or interpretation.

AST vs Parse Tree – Example



Example Grammar: Arithmetic Expressions

$G = (N, \Sigma, P, S)$ where:

$$N = \{E, T, F\}$$

$$\Sigma = \{(\,), +, *, \underline{\text{id}}\}$$

$$P = \{E \rightarrow T$$

$$E \rightarrow E + T$$

$$T \rightarrow F$$

$$T \rightarrow T * F$$

$$F \rightarrow \underline{\text{id}}$$

$$F \rightarrow (E)\}$$

$$S = E$$

Note: $P \subseteq N \times V^*$, where

$$V = N \cup \Sigma = \{E, T, F, (\,), +, *, \underline{\text{id}}\}$$

Note: $(A, \alpha) \in P$ is usually written:

$$A \rightarrow \alpha$$

or $A ::= \alpha$

or $A : \alpha$

Parsing Options

We can use CFGs and Backus-Naur form to describe context-free languages, but how do we take a stream of tokens and figure out which derivation/parse tree is correct?

- ▶ General parsing algorithms: CYK
- ▶ Restricted parsing methods: LL, LR

CYK Algorithm

This algorithm, named after Cocke, Younger, and Kasami, is a general algorithm for testing membership in a context-free language.

Advantage Algorithm works for any unambiguous grammar

Disadvantage Running time is $O(n^3)$, much too slow for practical use (n is the length of the input)

For more information, see §7.4.4 in Hopcroft, Motwani, and Ullman.

LL Parsing

LL parsing methods are faster than the CYK algorithm, but work only on grammars with some restrictions. LL stands for

- ▶ L – Left-to-right scan of input
- ▶ L – Leftmost derivation

LL methods are categorized by the number of tokens lookahead needed. So, LL(1)-methods are left-to-right methods using a leftmost derivation and one token lookahead.

LR Parsing

LR parsing methods are also faster than the CYK algorithm, but also work only on grammars with some restrictions. LR stands for

- ▶ L – Left-to-right scan of input
- ▶ R – Rightmost derivation

Similarly to LL, LR also has lookahead – LR(1) means one token of lookahead. LR methods are more flexible than LL methods – they handle a wider class of languages.

LL Parsing

LL parsing is often called *recursive-descent* parsing or *predictive* parsing (predictive parsing is recursive-descent parsing without backtracking).

- ▶ Most hand-written parsers are LL parsers, since they are easier to write than LR parsers
- ▶ Few tools use LL techniques – the most notable is ANTLR, which works over $LL(n)$ grammars (grammars that allow arbitrary lookahead)

These are *top-down* parsing techniques, since they start by expanding the start symbol and then expand out nonterminals based on the input, building the parse tree from the top down.

FIRST Sets

FIRST sets specify the first terminals that can be part of a member of the vocabulary, based on the productions. Essentially, they tell us when an element of our language vocabulary is *starting*. From §4.4, Dragon book:

- ▶ If X is a terminal, then $\text{FIRST}(X)$ is $\{X\}$
- ▶ If $X \rightarrow \epsilon$ is a production, add ϵ to $\text{FIRST}(X)$
- ▶ If $X \rightarrow Y_1 Y_2 \cdots Y_n$ is a production, add token a if a is in $\text{FIRST}(Y_j)$ and for all $i < j$ ϵ is in $\text{FIRST}(Y_i)$; if for all i $\text{FIRST}(Y_i)$ includes ϵ , add ϵ to $\text{FIRST}(X)$

FOLLOW Sets

FOLLOW sets specify the first terminals that can come after a given nonterminal. Essentially, they tell us when a nonterminal is *ending*. We don't define this for terminals, since it would be trivial (all terminals end immediately). From §4.4, Dragon book:

- ▶ Add $\$$ to $\text{FOLLOW}(S)$, where $\$$ is the input right endmarker
- ▶ For all productions of the form $A \rightarrow \alpha B \beta$, add the contents of $\text{FIRST}(\beta)$ to $\text{FOLLOW}(B)$
- ▶ For all productions of the form $A \rightarrow \alpha B$, or of the form $A \rightarrow \alpha B \beta$ where $\text{FIRST}(\beta)$ includes ϵ , add the contents of $\text{FOLLOW}(A)$ to $\text{FOLLOW}(B)$

PREDICT

With FIRST and FOLLOW sets, we can then construct PREDICT tables, which tell us which production we will process based on which we are currently processing and what the next input token is. We will build a table M of productions indexed by nonterminal and then terminal. From §4.4, Dragon book:

- ▶ For each production $A \rightarrow \alpha$
 - ▶ For each terminal a in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to table entry $M[A, a]$
 - ▶ If ϵ is in $\text{FIRST}(\alpha)$ then add $A \rightarrow \alpha$ to $M[A, b]$ for all b in $\text{FOLLOW}(A)$; if $\$$ was also in $\text{FOLLOW}(A)$, then add $A \rightarrow \alpha$ to $M[A, \epsilon]$
- ▶ Each entry in the table containing no productions is defined as an **error**

Recursive Construction

Using the information from PREDICT, we can now hand-construct a predictive parser

- ▶ For each production, create a new function
- ▶ Inside this function, read each token and call the appropriate function based on the contents of the PREDICT table
- ▶ Return from the function when the end of the production is reached – should return some data structure that is the representation of the AST

Limitations: Left Recursion

One weakness with LL parsing is that it cannot handle *left recursion*. Remember from last lecture that this occurs either

- ▶ directly, when the same nonterminal appears on the left-hand side and as the first vocabulary item on the right-hand side

$$E \rightarrow E + E$$

- ▶ indirectly, when a different nonterminal appears first on the right-hand side but expands immediately to the same nonterminal as is on the left

$$A \rightarrow B C$$

$$B \rightarrow A D$$

Handling Left Recursion

In cases where we have left recursion, we can remove it by modifying the grammar. See Algorithm 4.1 in §4.3 of the Dragon book for more information. Note that the resulting grammar is generally *not* as “clean” as the original, which is one of the disadvantages of LL parsing.

Pairwise Disjointedness and Left Factoring

What if we have two productions that both start identically?

$$\begin{aligned}
 stmt &\rightarrow \mathbf{if} \ expr \ \mathbf{then} \ stmt \ \mathbf{else} \ stmt \\
 stmt &\rightarrow \mathbf{if} \ expr \ \mathbf{then} \ stmt
 \end{aligned}$$

How do we know which to choose? We need to *left factor* the grammar – modify it so that we “factor out” matching left-hand portions of productions into a single production, adding multiple productions for the parts that are different. More details in §4.3 of the Dragon book. This ensures that the productions are *pairwise disjoint* – they do not have overlapping FIRST sets.

LL Parsing Example: The Grammar

$$E \rightarrow E + T$$

$$T \rightarrow T * F$$

$$F \rightarrow \text{id}$$

$$E \rightarrow E - T$$

$$T \rightarrow T / F$$

$$F \rightarrow \text{num}$$

$$E \rightarrow T$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

- ▶ This example, with slight modification, is from Appel's Modern Compiler Implementation in Java, Second Edition

Adding the Start Symbol

$$S \rightarrow E\$$$

$$E \rightarrow E + T$$

$$E \rightarrow E - T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow T / F$$

$$T \rightarrow F$$

$$F \rightarrow \text{id}$$

$$F \rightarrow \text{num}$$

$$F \rightarrow (E)$$

Removing Left Recursion

$$S \rightarrow E\$$$

$$E \rightarrow T E'$$

$$T \rightarrow F T'$$

$$E' \rightarrow + T E'$$

$$T' \rightarrow * F T'$$

$$F \rightarrow \text{id}$$

$$E' \rightarrow - T E'$$

$$T' \rightarrow / F T'$$

$$F \rightarrow \text{num}$$

$$E' \rightarrow \epsilon$$

$$T' \rightarrow \epsilon$$

$$F \rightarrow (E)$$

FIRST and FOLLOW Sets

	FIRST	FOLLOW
S	(id num	\$
E	(id num) \$
E'	+ - ϵ) \$
T	(id num) + - \$
T'	* / ϵ) + - \$
F	(id num) * / + - \$

Left Factoring/Pairwise Disjointness

$$S \rightarrow E\$$$

$$E \rightarrow T E'$$

$$E' \rightarrow + T E'$$

$$E' \rightarrow - T E'$$

$$E' \rightarrow \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T'$$

$$T' \rightarrow / F T'$$

$$T' \rightarrow \epsilon$$

$$F \rightarrow \text{id}$$

$$F \rightarrow \text{num}$$

$$F \rightarrow (E)$$

Do we need to make any changes to this grammar to ensure productions are pairwise disjoint wrt the FIRST sets? In other words, does a terminal in the first set identify just one production, or more than one?

Left Factoring/Pairwise Disjointness

$$S \rightarrow E\$$$

$$E \rightarrow T E'$$

$$E' \rightarrow + T E'$$

$$E' \rightarrow - T E'$$

$$E' \rightarrow \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T'$$

$$T' \rightarrow / F T'$$

$$T' \rightarrow \epsilon$$

$$F \rightarrow \text{id}$$

$$F \rightarrow \text{num}$$

$$F \rightarrow (E)$$

Do we need to make any changes to this grammar to ensure productions are pairwise disjoint wrt the FIRST sets? In other words, does a terminal in the first set identify just one production, or more than one? Fortunately, we're fine – no need to left factor.

The PREDICT Table

	+	*	id	()	\$
<i>S</i>			$S \rightarrow E\$$	$S \rightarrow E\$$		
<i>E</i>			$E \rightarrow TE'$	$E \rightarrow TE'$		
<i>E'</i>	$E' \rightarrow +TE'$				$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
<i>T</i>			$T \rightarrow FT'$	$T \rightarrow FT'$		
<i>T'</i>	$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$			$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
<i>F</i>			$F \rightarrow \text{id}$	$F \rightarrow (E)$		

- ▶ left out $-$, $/$, and `num`, which are almost identical to $+$, $*$, and `id`, respectively

A Sample Parser in OCaml

LR Parsing

LR parsing is often called *shift-reduce* parsing because of the method used during the parsing process.

- ▶ LR parsing is strictly more powerful than LL parsing – all LL grammars can be parsed by LR techniques
- ▶ LR parsers are generally machine-generated, since the tables used in the technique would be difficult to generate by hand
- ▶ Most familiar parsing tools are based on LR techniques, including Yacc, Bison, and (for us) `ocamlyacc`

These are *bottom-up* parsing techniques, since they start by grouping tokens into nonterminals, which are then grouped into larger and larger nonterminals, building the parse tree from the leaves up. We essentially build a rightmost derivation in reverse.

Handles

A **handle** is a substring of the current input string (the program we are parsing) that matches the right-hand side of a production.

- ▶ to parse, we want to identify handles in the input
- ▶ handles can then be “pruned”, replacing them with their left-hand sides
- ▶ each pruning operation will identify a node in the parse tree, with the children being in the handle
- ▶ a successful parse will lead to a pruning operation that just leaves the start symbol

LR Parser Components

An LR parser is made up of several components:

- ▶ A **stack** to hold states and grammar symbols
- ▶ The **input** program
- ▶ The parsing **output**
- ▶ The **driver program** that determines which parsing steps to take
- ▶ The **parsing table**, divided up into two components, a parsing action function **action** and a state transition function **goto**.

The parser acts as a state machine, consuming input, transitioning between states, and manipulating the stack until either the input is accepted or an error is detected.

The Action and Goto Functions

- ▶ The action function determines what step should be taken, given the current state (on top of the stack) and the current input symbol. Possibilities are
 - ▶ shift state s onto the stack
 - ▶ reduce the stack contents by a production $A \rightarrow \beta$
 - ▶ accept
 - ▶ error
- ▶ The goto function determines which state to go to next, based on the current state and current grammar symbol, and is only used during the reduce action.

Configurations

Our LR parser will be an automaton, moving between configurations. Each configuration will be of the form:

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m, a_i a_{i+1} a_n \$)$$

With the first component the stack of states and grammar symbols (terminals and nonterminals) and the second the unprocessed input.

Action: Shift

In state s_m , on input a_i , if the action is shift s , the new configuration will be:

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m a_i s, a_{i+1} a_n \$)$$

The symbol and the current state are both pushed onto the stack. Note we are assuming LR(1) parsing – only 1 token of lookahead.

Action: Reduce

In state s_m , on input a_i , if the action is reduce $A \rightarrow \beta$, we enter configuration

$$(s_0 \ X_1 \ s_1 \ X_2 \ s_2 \ \cdots \ X_{m-r} \ s_{m-r} \ A \ s, a_i \ a_{i+1} \ a_n \$)$$

where r is the length of β , and s is the result of calling goto on state s_{m-r} and symbol A . Essentially, for a production of length r , we will pop off r states and r grammar symbols, and then goto the new state based on the goto function, pushing this new state onto the stack. We will **not** consume any input.

Action: Accept

This action means that parsing is complete, and we have a word that is in the language of the grammar (essentially, a syntactically valid program).

Action: Error

This action means that we have discovered an error in the input. We will then need to use whatever error recovery logic we have in the parser.

Conflicts

We can have two types of conflicts:

- ▶ Shift/Reduce: This happens when we have matched enough to form a handle (which we would want to reduce) and also have the option of shifting to continue building a longer handle (so we would want to shift).
- ▶ Reduce/Reduce: This happens when we have a handle which matches the right-hand sides of multiple productions.

Technically, a grammar with either problem is not LR, but at least in the first case we can often handle it. We usually resolve the first in favor of shifting; the second often requires changes to the grammar.

Varieties of LR Parsers

LR parsers come in a confusing variety of acronyms:

- ▶ LR(0) – LR with no lookahead (just uses the stack)
- ▶ SLR – Simple LR, similar to LR(0) but modifies the method for adding reduce actions to the action table
- ▶ LR(1) – similar to LR(0), but with one token of lookahead and a more complex notion of *item* (items are augmented productions used for building the parsing tables)
- ▶ LALR(1) – lookahead LR(1), similar to LR(1) but condenses the parsing tables by merging similar entries– slightly less powerful than LR(1)

ocaml yacc and LR Parsing

For more information on how to manually build the parsing tables, see §4.7 of the Dragon book. This provides a very detailed introduction to this topic. Instead of building the tables ourselves, we will make use of `ocaml yacc` to do it for us.

Using ocamlyacc

Similarly to `ocamllex`, `ocamlyacc` expects input in a file in its own format and then *converts* it to OCaml code. Given a file with a parser grammar called `grammar.mly`,

```
ocamlyacc grammar.mly
```

will generate file

```
grammar.ml
```

for the parser code, and will also produce an interface file, `grammar.mli`, with type declarations and token definitions.

ocamlyacc Grammar Format

Grammars written for ocamlyacc have a format similar to those for ocamllex.

```

1  %{
2      ... header here ...
3  %}
4  ... declarations here ...
5  %%
6  ... rules here ...
7  %%
8  ... trailer here ...
  
```

Header and Trailer Sections

Similarly to `ocamllex`, the header and trailer in the file are inserted into the generated `grammar.ml` file, and can contain arbitrary code.

- ▶ typically used to declare types and functions used in the rules, potentially for language semantics
- ▶ the footer may “bootstrap” the parser to get it going, similarly to the sample lexers we saw
- ▶ both sections may be omitted

Declarations

- ▶ `%token symbol ... symbol`
declares the given symbols as tokens/terminals in the language, and are represented as constant constructors
- ▶ `%token <type> symbol ... symbol`
declares the given symbols as tokens in the language with attached data, and are represented as constructors with a data item of this type (useful for strings, ints, etc, where we want to value associated with the token)

Declarations

- ▶ `%start symbol ... symbol`
declares the given symbols to be start symbols for the grammar (also referred to as entry points)
- ▶ `%type <type> symbol ... symbol`
specifies the semantic type of the given symbols, represented as an OCaml type – this is only needed for start symbols, and is inferred for the rest
- ▶ `%left, %right, or %nonassoc symbol ... symbol`
provides precedence and associativity for symbols, and is a cleaner solution than using stratification since it makes the grammars easier to read; higher precedence symbols are on later lines

Rules

Rules are of the form

```

1 nonterminal :
2     symbol ... symbol { semantic-action }
3     | ...
4     | symbol ... symbol { semantic-action }
```

- ▶ semantic actions are arbitrary OCaml expressions, which should have the same type as declared or inferred for the symbol nonterminal
- ▶ information from the productions can be accessed by position – the first symbol is \$1, the second \$2, etc (including tokens!)

Example

```

1 (* File: expr.ml *)
2 type expr =
3     Term_as_Expr of term
4     | Plus_Expr of (term * expr)
5     | Minus_Expr of (term * expr)
6 and term =
7     Factor_as_Term of factor
8     | Mult_Term of (factor * term)
9     | Div_Term of (factor * term)
10 and factor =
11     Id_as_Factor of string
12     | Parenthesized_Expr_as_Factor of expr
  
```

Example Lexer

```

1 { (*open Exprparse*) }
2 let numeric = ['0' - '9']
3 let letter = ['a' - 'z' 'A' - 'Z']
4 rule token = parse
5   | "+" {Plus_token}
6   | "-" {Minus_token}
7   | "*" {Times_token}
8   | "/" {Divide_token}
9   | "(" {Left_parenthesis}
10  | ")" {Right_parenthesis}
11  | letter (letter|numeric|"_" )* as id {Id_token id}
12  | '\n' {EOL}
13  | [' ' '\t'] {token lexbuf}

```

Example Parser – Header and Declarations

```
1  %{ (*open Expr*)  
2  %}  
3  %token <string> Id_token  
4  %token Left_parenthesis Right_parenthesis  
5  %token Times_token Divide_token  
6  %token Plus_token Minus_token  
7  %token EOL  
8  %start main  
9  %type <expr> main  
10 %%
```

Example Parser – Rules

```

1  expr:
2    term                                { Term_as_Expr $1 }
3    | term Plus_token expr              { Plus_Expr ($1, $3) }
4    | term Minus_token expr             { Minus_Expr ($1, $3) }
5  term:
6    factor                               { Factor_as_Term $1 }
7    | factor Times_token term           { Mult_Term ($1, $3) }
8    | factor Divide_token term          { Div_Term ($1, $3) }
9  factor:
10   Id_token                             { Id_as_Factor $1 }
11   | Left_parenthesis expr Right_parenthesis
12                                     {Parenthesized_Expr_as_Factor $2 }
13 main:
14   | expr EOL                           { $1 }
  
```

Parsing Code

The generated file, `grammar.ml`, provides the parsing functions. One is generated for each start symbol (entry point). The parsing function takes a lexing function and a buffer, returning a value representing the semantic "content" of the parsed program (based on the declared return type of the entry point).

Example – Using the Parser

```

1 # #use "expr.ml";;
2 ...
3 # #use "exprparse.ml";;
4 ...
5 # #use "exprlex.ml";;
6 ...
7 # let test s =
8   let lexbuf = Lexing.from_string (s^"\n") in
9     main token lexbuf;;
10 # test "a + b";;
11 - : expr =
12 Plus_Expr
13 (Factor_as_Term (Id_as_Factor "a"),
14  Term_as_Expr (Factor_as_Term (Id_as_Factor "b")))

```