Parsing

Parsing in Theory

* The “parse tree” is really the **second** step in the compiling process
* The syntactic analyser uses parsing to:
  + make sure the syntax is correct
    - Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly
    - AS YOU KNOW, what may be syntactically correct may not always work!!
  + build the tree in order to trace where
    - the data and values are to be placed
    - check for syntactically correct **input**
      * values are going to the right spot
      * conditions are being checked
      * data types match
* A *parser* is the component of the analyser that constructs a parse tree for a string (which would be code)
* Two common types of parsers
  + *Top down* - produce the parse tree, beginning at the root
    - Order is that of a leftmost derivation
    - Traces or builds the parse tree in preorder
    - A *recursive descent parser* easily implements a top-down parser for simple grammars
  + *Bottom up* - produce the parse tree, beginning at the leaves
    - Order is that of the reverse of a rightmost derivation

The types of Parsers for Grammars

* An *LL(k)* parser, does a **L**eft-to-right parse, a **L**eftmost-derivation, and **k-**symbol lookahead
  + Grammars where one can decide which rule to use by examining only the *next* token are **LL(1)**
    - next token being the token in the code given

|  |  |
| --- | --- |
| **Types of Parsers** | |
| LL(n) | LR(n) |
| Left to right, Leftmost derivation, look ahead at most n symbols. | Left to right, Right derivation, look ahead at most n symbols. |

|  |  |
| --- | --- |
| **Top Down Example #1 (LL1)** | |
| Given Grammar | Given Input String |
| a    E → T + E  b            | T  c    T → ( E )  d            | int  e            | int \* T | int \* int  3 tokens here |

* LL(1) grammars are widely used in practice, not hard to demonstrate
* A LR(1) or LL(1) parser never need to *“look ahead”* more than *one* input token to know what parser production rule applies
* A **non-predictive** LL/R(1) may have to do some backtracking since some productions might have several options that START with the SAME symbol, but others afterwards, have varying symbols
  + we then have to try each option

|  |
| --- |
| **Non-Predictive LL1 Productions** |
| Given Grammar |
| a    E → T + E  b            | T  c    T → ( E )  d            | int  e            | int \* T |

Top Down “Recursive Parsing” Algorithm

* also called recursive descent
* is an LL(1)
  + Left to right, Leftmost derivation, look ahead at most 1 symbol
* must
  + be given the entire grammar
    - please put line letter or # to locate easier
    - break up each or (|) on a different line **(easier to read)**
  + the lex“ified” (broken up) input stream
  + BE RIGHT RECURSIVE (explain later why)
  + match 1 symbol ahead in the input
  + start with the
    - start symbol in the production (douh)
    - and a marker BEFORE the first token in the input string []
* for the algorithm, try every single combo until (or maybe not) we find a match
  + match means EVERY ***leaf*** in the tree matches the input
    - has to be a COMPLETE MATCH
      * could have ALL of the input in the tree, but the tree has other leaves left
  + again, leftmost first for each production!!
  + in order of productions (top to bottom to be able to backtrack)

First Top Down Example

|  |  |
| --- | --- |
| **Top Down Example #1 Complete** | |
| Given Grammar | Given Input String |
| a    E → T + E  b            | T  c    T → ( E )  d            | int  e            | int \* T | int \* int |

After checking that it is right recursive, try different combinations, (with a pattern)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | rule | tree | input | what happened? |
| 1 | - | TopDownExample1 | [] **int** \* int  red is look ahead token | start production |
| 2 | a | TopDownExample2 | [] **int** \* int | try 1st production of E |
| 3 | c | TopDownExample3 | [] **int** \* int | leftmost on tree, (so T), then first PRODUCTION of T.  Does not fit, ( is not there) |
| 4 | d | TopDownExample4 | [] int **\*** int | when one fit, but not the rest!! |
| 5 | e | TopDownExample5 | [] int **\*** int | Fits int a \*!! Now for T!! |
| 6 | c | TopDownExample6 | [] int \* **int** | No match for ( |
| 7 | d | TopDownExample7 | [] int \* **int** | INPUT String matches, BUT THE TREE has more leaves!! (Must be an exact match!! |
| 8 | e | TopDownExample8 | [] int \* **int** | matches int, but rest is no good. we’re done with T!! back to the start for the input string!! |
| 9 | b | TopDownExample9 | [] **int** \* int |  |
| 10 | c | TopDownExample10 | [] **int** \* int | no match |

What would the rest look like?? **Answerb:**

Review of Left/Right Recursion

* Left Recursion, call (symbol) ***to itself*** is physically LEFT of the operator
* Right Recursion, call (symbol) ***to itself*** is physically RIGHT of the operator

|  |  |
| --- | --- |
| **Recursion Example** | |
| Left Recursion | Right Recursion |
| (from previous examples)  <expr> 🡪 <expr> **+** <term>  | <term>  <term> 🡪 <term> **\*** <factor>  | <factor> | (specifically for power)  <factor> 🡪 <exp> \*\* <factor>  | <exp>  <exp> 🡪 ( <exp> )  | <id> |

Try a top down parser on THIS grammar. There is something wrong when you begin your work. It will become apparent. What is it?

|  |  |
| --- | --- |
| **Top Down Example #2** | |
| Given Grammar | Given Input String |
| a   expr → expr  +  term  b              |   expr   -  term  c              |   term  d   term → term  \*  factor  e              |   term  /  factor  f              |   factor  g   factor → num  h             |   id       // (identifier) | id - num \* id |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | rule | tree | input | what happened? |
| 1 | - |  | []**id** - num \* id | start production |
| 2 | a |  | []**id** - num \* id |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |
| 5 |  |  |  |  |

So what’s the issue??Left-recursive grammars and its elimination

* (Direct) Left Recursion
  + X calls the leftist most symbol of production, which is X
* Indirect left recursion
  + X calls A (leftist most symbol of production), then A calls X

|  |  |
| --- | --- |
| **Types of Recursion Calls** | |
| Direct | Indirect |
| X -> X β | X -> A β  A -> X |

* Left-recursive grammar elimination
  + We can manually or automatically rewrite a grammar removing left-recursion, making it ok for a top-down parser.
  + Thankfully there is a pattern, but we do have to add a new non-terminal and terminal symbol (usually)

|  |  |
| --- | --- |
| **Two cases of left recursion** | |
| |  |  | | --- | --- | | ***#*** | ***Production rule*** | | **a**  **b**  **c** | ***expr*   →  *expr  +  term***  ***|   expr   -  term***  ***|   term*** | | |  |  | | --- | --- | | ***#*** | ***Production rule*** | | **d**  **e**  **f** | ***term* →  *term  \*  factor***  ***|   term  /  factor***  ***|   factor*** | |

|  |  |
| --- | --- |
| **Transform as follows** | |
| |  |  | | --- | --- | | ***#*** | ***Production rule*** | | **a**  **b**  **c**  **d** | ***expr*    →   *term  expr2***  ***expr2*  →   *+  term expr2***  ***|     -  term expr2***  ***|    Ɛ*** | | |  |  | | --- | --- | | ***#*** | ***Production rule*** | | **e**  **f**  **g**  **h** | ***term* →  *factor term2***  ***term2* →  *\*  factor  term2***  ***|   /  factor   term2***  ***|   Ɛ*** | |

Notice the pattern. Try to keep it simple.

Did left recursion elimination really work?? Try creating a parse tree to match the syntax:

term + term – term

for BOTH grammars

Answerb:

|  |  |
| --- | --- |
| **term + term - term** | |
| ***expr*   →  *expr  +  term***  ***|   expr   -  term***  ***|   term*** | ***expr*    →   *term  expr2***  ***expr2*  →   *+  term expr2***  ***|     -  term expr2***  ***|    Ɛ*** |
|  |  |

Strategies for Eliminating Left Recursion

* remember, we have to get rid of left recursion in order for our parser to work!

|  |
| --- |
| **The general algorithm to convert left recursion** |
| For each rule of the form  A \rightarrow A\alpha_1\,|\,\ldots\,|\,A\alpha_n\,|\,\beta_1\,|\,\ldots\,|\,\beta_m  where:   * A is a left-recursive nonterminal * \alpha is a sequence of non-terminals and terminals that is not null (\alpha \ne \epsilon ) * \beta is a sequence of non-terminals and terminals that does not start with A.   replace the ***A-starting*** production by the production:  A \rightarrow \beta_1A^\prime\, |\, \ldots\,  |\,  \beta_mA^\prime  And create a new nonterminal (denoted by the ‘ or “prime”)  A^\prime \rightarrow \epsilon\, |\, \alpha_1A^\prime\,  |\,  \ldots\, |\, \alpha_nA^\prime  This newly created symbol is often called the "tail", or the "rest". |

|  |
| --- |
| **Other templates for Left Recursion Elimination** |

|  |  |  |
| --- | --- | --- |
| Left Recursive | Strings Produced | Conversion |
| expr 🡪 expr + expr      | int      | string | *int + int*  *int + int + int*  *int + string + int + int*  *string + string + string + int* | expr 🡪 int REST               | string REST  REST 🡪 + expr               | ε |
| **same as above** |  | expr 🡪 int REST               | string REST  REST 🡪 + expr REST               | ε |
| S → S α         | β | β  β α   * α α   … | S → β S’  S’ → α S’          | ε |
| exp 🡪 exp addop term             | term  (much like first one above) | *term addop term*  *term addop term addop term*  *term addop term addop term addop term* | exp 🡪 term exp’  exp’ 🡪 addop term exp’              |  ε |

Try converting this grammar from a left to right recursive. Please create a few possible strings, then convert, and test with those possible strings:

|  |  |
| --- | --- |
| S ==> Sa      | b | **Answer:** |
| S ==> Sab      | cd | **Answer:** |
| S ==> S U S      | S S      | S\*      | (S)      | a  \*U is union here | **Answer:** |
| S ==> Aa      | b  A ==> Ac      | Sd      | epsilon | **Answer:** |
| rexpr ==> rexpr U rterm | rterm  rterm ==> rterm rfactor | rfactor  rfactor ==> rfactor\* | rprimary  rprimary ==> a | b | **Answer:** |
| S 🡪 A     | B  A 🡪 ABc    | AAdd    | a    | aa  B 🡪 Bee    | b | Answer: |

Problems with top down “recursive parsing”

* When going forward, the parser consumes tokens from the input, so what happens if we have to back up?
  + What **data structures** can we use??
* Algorithms that use backup tend to be, in general, inefficient
  + There might be a large number of possibilities to try before finding the right one or giving up
  + ***BUT,*** A top down grammar can limit backtracking if
    - it only has one unique starting symbol per rule in each non-terminal
    - The technique of rule factoring can be used to eliminate multiple rules for a non-terminal

|  |
| --- |
| **Left Factoring Example** |
| leftFactoringExample |

* Grammar rules which are left-recursive lead to non-termination!
  + so we have to change the Grammar if Left-Recursive!!
  + one with a rule like: E -> E + T
  + **ewwww**, but not that bad

Summary of Recursive Descent Parsing

* Simple and general parsing strategy
  + Left-recursion must be eliminated first
  + … but that can be done automatically
* Unpopular because of backtracking
  + Thought to be too inefficient
* In practice, backtracking is eliminated by further restricting the grammar to allow us to successfully *predict* which rule to use

Predictive Parsers

* The goal of predictive parsing is to construct a top-down parser that **never backtracks.**
  + there can be many rules for a non-terminal makes parsing hard
* A *predictive parser* processes the input stream typically from left to right
  + Is there any other way to do it?
  + Yes, for other programming languages!
* It uses information from **peeking** ahead at the *upcoming terminal symbols* in the input string to decide which grammar rule to use next
* It *always* makes the right choice of which rule to use
  + if the grammar is set up properly
* issues
  + How much it can peek ahead
  + ***ANYTHING on the RHS of the grammar must begin with a unigue symbol!! (per production)***

|  |
| --- |
| **What makes a good Predictive Parsing Grammar #1** |
| whatMakesAGoodPredictiveParserGrammar |

|  |  |
| --- | --- |
| **What makes a good Predictive Parsing Grammar #2** | |
| Good Predictive Grammar | Why?? |
| *S* → **if** *E* **then** *S* **else** *S*  *S* → **begin** *S L*  *S* → **print** *E*  *L* → **end**  *L* → **;** *S L*  *E* → num = num | *S* expression starts **either** with  an IF, BEGIN, or PRINT token,  *L* expression start with an  END **or a** SEMICOLON token,  *E* expression has only one production. |

Good or Bad Predictive Grammars?

|  |  |
| --- | --- |
| E →T E'  E' →+ T E'       | - T E'       | ε  T →F T'  T' →\* F T'       | / F T'       | ε  F → num      | id |  |
| Consider the grammar  E → T + E  E → T  T → int  T → int \* T  T → ( E ) | Answer: |

What you JUST did is called “pairwise disjointness test”

What if the Grammar fails disjointness test?

* Must **left-factor** to create a predictive parsing grammar
  + Left-factoring involves rewriting rules so that
    - if a non-terminal has > 1 rule, each begins with a **terminal**
    - add new non-terminals (new letters) to factor out common prefixes of rules

|  |
| --- |
| **Left Factoring Example** |
|  |

Left factor the grammar below:

|  |  |
| --- | --- |
| S ==> 0S1      | 01 | **Answer:** |
| S ==> abx      | aby      | acx      | acy | **Answer:** |
| A  🡪 bcW     | bcE  W 🡪 x  E  🡪 q | Answer: |

Using Parsing Tables for Predictive Parsers

* LL(1) Predictive Parsers means that for each non-terminal and token there is only **one** production
  + passes disjointness test
* Can be represented as a simple table
  + One dimension for current non-terminal to expand
  + One dimension for next token
  + A table entry contains one rule’s action or empty if error
* Method similar to recursive descent, except
  + For each non-terminal S
  + We look at the next token *a*
  + And chose the production shown at table cell [S, a]
* Use a stack to keep track of pending non-terminals
* Reject when we encounter an error state, accept when we encounter end-of-input

|  |
| --- |
| **Reading a Parse Table** |
| LLParseTableExample |

|  |
| --- |
| **LL 1 Predictive Parsing Table Full Example #1** |
| LLParseTableExercise1  [Video](https://www.youtube.com/watch?v=FulRzWXRpZ0&list=PLC7fNkE1QplaeBUwuDL2FHx6ngLCELL0d&index=19): (many need to adjust screen resolution)  **missed ( and ) as terminals in video!!** |

|  |
| --- |
| **LL 1 Predictive Parsing Table Full Example #2** |
| S 🡪 (1) if <Expr> then <S> else <S>  (2) while <Expr> do <S>   |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | **if** | **then** | **else** | **while** | **do** | **begin** | **end** | **id** | **;** | **$** | | <S> | **1** |  |  | **2** |  | **3** |  |  |  |  | | <Stmt> | **4** |  |  | **4** |  | **4** | **5** |  |  |  | | <Expr> |  |  |  |  |  |  |  | **6** |  |  |   (3) begin <Stmt> end  Stmt 🡪 (4) <S> ; <Stmt>   (5) epsilon  Expr 🡪 (6) id   |  |  |  | | --- | --- | --- | | Stack | Input | Action | | $ | while id do begin begin end ; end $ | Push() // starting symbol | | <S> $ | while id do begin begin end ; end $ | Pop(); push(while <Expr> do <S>); | | while <Expr> do <S> $ | while id do begin begin end ; end $ | **Pop(); next++;  // since we have a match** | | <Expr> do <S> $ | id do begin begin end ; end $ | Pop(); push(id); | | id do <S> $ | id do begin begin end ; end $ | **Pop(); next++;  // since we have a match** | | do <S> $ | do begin begin end ; end $ | **Pop(); next++;  // since we have a match** | | <S> $ | begin begin end ; end $ | Pop(); push(begin <Stmt> end); | | begin <Stmt> end $ | begin begin end ; end $ | **Pop(); next++;  // since we have a match** | | <Stmt> end $ | begin end ; end $ | Pop(); push(<S> ; <Stmt>); | | <S> ; <Stmt> end $ | begin end ; end $ | Pop(); push(begin <Stmt> end); | | begin <Stmt> end  ; <Stmt> end $ | begin end ; end $ | **Pop(); next++;  // since we have a match** | | <Stmt> end  ; <Stmt> end $ | end ; end $ | pop(); push(epsilon); // which is nothing | | end  ; <Stmt> end $ | end ; end $ | **Pop(); next++;  // since we have a match** | | ; <Stmt> end $ | ; end $ | **Pop(); next++;  // since we have a match** | | <Stmt> end $ | end $ | pop(); push(epsilon); // which is nothing | | end $ | end $ | **Pop(); next++;  // since we have a match** | | $ | $ | DONE!!! | |

Using predictive parsing and the same grammar and table, try:

1. int \* int + int
2. (int + int) \* int
3. (int + int + int) int

Introduction to Bottom-Up Parsing

* preferred method in practice
* also called LR parsing
  + tokens read ***L***eft to right
  + R is ***R***ightmost derivation
* less needy than Top Down
* Don’t need left-factored grammars
  + Left recursion fine
  + Just as efficient
  + Builds on ideas in top-down parsing

The Bottom-up Parsing algorithm

* using the grammar given, it “reduces” the input string to its start symbol
  + complete a RIGHTMOST parse tree to MATCH your input
  + the just FLIP your parse tree
* then it would use the same recursion to solve

|  |  |
| --- | --- |
| **Bottom Up Example #1** | |
| Given Grammar | Given Input String |
| a    E → T + E  b            | T  c    T → ( E )  d            | int  e            | int \* T | int \* int + int |

|  |  |  |
| --- | --- | --- |
| # | rule | Tree |
| 1 | - | BottomUpExample1 |
| 2 | a | BottomUpExample2 |
| 3 | b | BottomUpExample3(solved E first since Rightmost has priority!!) |
| 4 | d | BottomUpExample4 |
| 5 | e | BottomUpExample5 |
| 6 | d | BottomUpExample6 |
| 7 | FLIP | So it would reduce the original input to its’ start symbol.BottomUpExample7-flip   (Arrows should go OTHER direction) |

Bottom up parsing basics

Basics of Shift Reduce Parsing (LR(1))

* used for efficiency
  + no backtracking or backup (recursion)
    - this obviously saves time/memory/etc…

|  |
| --- |
| **Input String** |
| ↑ id = id + id \* int $ |

* input stream (string)
  + LR(1) – 1 lookahead (now called LA) in the input stream
  + Split input string into two substrings
    - Right substring (a string of terminals) is as yet unexamined by parser
    - Left substring has terminals and non-terminals
  + The dividing point is marked by a ↑
    - The ↑ is not part of the string

|  |
| --- |
| **Grammar** |
| Assign ← *id* = Sums  Sums ← Sums + Products  Sums ← Products  Products ← Products \* Value  Products ← Value  Value ← *int*  Value ← *id* |

* Bottom-up parsing uses only two kinds of actions: Shift and Reduce
  + *Shift:* Move ↑ one place to the right, add LA to stack
    - Shifts a terminal to the left string
    - E + (↑ int )  ⇒ E + (int ↑ )
  + *Reduce:* Apply an inverse production at the right end of the left string
    - meaning, use the LHS to reduce the portion the marker has uncovered
* We are checking to make sure a given input string matches a Grammar
* Must be given
  + Grammar that is LEFT RECURSIVE
  + Input string (input MAY or MAY NOT be correct)
  + $ is added to input string to denote end

|  |  |
| --- | --- |
| **Example Grammar and Input for LR(1) Shift Reduce Parser** | |
| Grammar | Input String |
| Assign ← *id* = Sums  Sums ← Sums + Products  Sums ← Products  Products ← Products \* Value  Products ← Value  Value ← *int*  Value ← *id* | id = id + id \* int $ |

* At every parse step, the entire input text is divided into
  + a parse stack
  + **current** lookahead symbol
  + and remaining unscanned text

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Example LR(1) Parse using Shift Reduce** | | | | | |
| **id = id + id \* int $** | | | | | |
| **Step** | **Parse Stack** | **Look Ahead** | **Unscanned** | **Parser Action** | |
| 0 | *empty* | *id* | = id + id\*int $ |  | Just the start |
| … | *…* | *…* | … | … | … |
| 8 | *id = Sums + id* | *\** | int $ | A | Reduce “id” to “Value” |

* The parser's next action is determined by the
  + the lookahead symbol THEN the rightmost stack symbol(s)
    - this one is tricky, notice it can be MORE than one symbol but it ***starts*** from the right
  + The action is read from a table containing all syntactically valid combinations of stack and lookahead symbols.
  + I created a table for you to better explain some of the actions

|  |
| --- |
| **Grammar** |
| Assign ← *id* = Sums  Sums ← Sums + Products  Sums ← Products  Products ← Products \* Value  Products ← Value  Value ← *int*  Value ← *id* |

* BUT, the parser wants the BIGGEST match possible for RHS in Stack
  + so it might wait until something doesn’t match
  + this works only if 2 or more productions START with the same symbol
    - like <Value> and <Assign>, both start with “id”
  + in many references, this is not mentioned

A LR(1) Shift Reduce Example

* again must be given a Grammar and Input String
* use LR Shift Reduce Parser Table
* please use overall exercise table given
  + broken up into, step, parse stack, look ahead, etc…

|  |  |
| --- | --- |
| **Example Grammar and Input for LR(1) Shift Reduce Parser** | |
| Grammar | Input String |
| Assign ← *id* = Sums  Sums ← Sums + Products  Sums ← Products  Products ← Products \* Value  Products ← Value  Value ← *int*  Value ← *id* | id = id + id \* int $ |

|  |  |
| --- | --- |
| **LR(1) Shift Reduce Parser Response** | |
| **Shift** | 1. lookahead has a partial Grammar match, wait for more 2. combo of lookahead and rightmost in parse stack has a partial Grammar match, wait for more |
| **Parse** | no match for lookahead, reduce rightmost symbol in stack (find largest match) |

|  |  |  |
| --- | --- | --- |
| **LR(1) Shift Reduce Parser Response** | | |
|  | **What’s in the Stack** | **Reaction** |
| A | no production match in ***ENTIRE*** stack to ANY production | reduce **rightmost** non-terminal |
| B | partial **rightmost** stack match | Shift, parser wants to see more before deciding |
| C | whole **rightmost** production matches in stack and lookahead **!=** $ | Shift, still have more reading to do |
| D | **rightmost** whole production matches in stack and lookahead **is** $ | reduce **rightmost** production |
| E | **rightmost** whole production matches START and lookahead **is** $ | ACCEPT!! |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Step** | **Parse Stack** | **Look Ahead** | **Unscanned** | **Parser Action** | |
| 0 | *empty* | *id* | = id + id \* int $ |  | Just the start |
| 1 | *id* | *=* | id + id \* int $ | B | Shift, Parser wants as big of a match as possible |
| 2 | *id =* | *id* | + id \* int $ | B | Shift, Parser wants as big of a match as possible |
| 3 | *id = id* | *+* | id \* int $ | A | Reduce “id” to “Value” |
| 4 | *id = Value* | *+* | id \* int $ | A | Reduce “Value” to “Products” |
| 5 | *id = Products* | *+* | id \* int $ | A | Reduce “Products” to “Sums” |
| 6 | *id = Sums* | *+* | id \* int $ | C | Shift, since more to go (lookahead) |
| 7 | *id = Sums +* | *id* | \* int $ | B | Shift, since Sums + is a partial match, parser wants to see more |
| 8 | *id = Sums + id* | *\** | int $ | A | Reduce “id” to “Value” |
| 9 | *id = Sums + Value* | *\** | int $ | A | Reduce “Value” to “Products” |
| 10 | *id = Sums + Products* | *\** | int $ | C | Shift, still have more to do |
| 11 | *id = Sums + Products \** | *int* | $ | B | Shift, since Products \* is a partial match, parser wants to see more |
| 12 | *id = Sums + Products \* int* | *$* |  | A | Reduce “int” to “Value” |
| 13 | *id = Sums + Products \* Value* | *$* |  | D | Reduce “Products \* Value” to “Products” |
| 14 | *id = Sums + Products* | *$* |  | D | Reduce “Sums + Products” to “Sums” |
| 15 | *id = Sums* | *$* |  | D | Reduce “id = Sums” to “Assign” |
| 16 | *Assign* | *$* |  | E | ACCEPT |

Try your own example:

Expr→ Expr Op Expr

Expr→ (Expr)

Expr→ -Expr

Expr→ num

Op→ + | - | \*

|  |  |  |
| --- | --- | --- |
| A | no production match in ***ENTIRE*** stack to ANY production | reduce **rightmost** non-terminal |
| B | partial **rightmost** stack match | Shift, parser wants to see more before deciding |
| C | whole **rightmost** production matches in stack and lookahead **!=** $ | Shift, still have more reading to do |
| D | **rightmost** whole production matches in stack and lookahead **is** $ | reduce **rightmost** production |
| E | **rightmost** whole production matches START and lookahead **is** $ | ACCEPT!! |

Input: num \* (num + num)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Step** | **Parse Stack** | **Look Ahead** | **Unscanned** | **Parser Action** | |
| 0 | *empty* | *num* | \* (num + num) |  | Just the start |
| 1 |  |  |  |  |  |
| 2 |  |  |  |  |  |
| 3 |  |  |  |  |  |
| 4 |  |  |  |  |  |
| 5 |  |  |  |  |  |
| 6 |  |  |  |  |  |
| 7 |  |  |  |  |  |
| 8 |  |  |  |  |  |

Answerb:

Top down vs. bottom up parsing

* Both are general techniques that can be made to work for most (Top-Down) if not all (bottom-up) languages (but not all grammars!).
* Recall that a given language can be described by several grammars

|  |  |
| --- | --- |
| **Different Grammar but same language??** | |
| Left Recursive | Right Recursive |
| E -> E + Num  E -> Num  \*\* this one causes problems later! | E -> Num + E  E -> Num |
| Both of these grammars describe the same language!!  1 + 2 + 3 = (L Recursive) = valid  1 + 2 + 3 = (R Recursive) = valid | |

* The parsing problem is to connect the root node S with the tree leaves, given the input (code)
* **Top-down parsers**
  + starts constructing the parse tree at the top (root) and move down towards the leaves.
  + Easy to implement by hand, but requires restricted grammars.
    - Predictive parsers (e.g., LL(k)) **covered later**
* **Bottom-up parsers**
  + build nodes on the bottom of the parse tree first. Suitable for automatic parser generation, handles larger class of grammars.
    - shift-reduce parser (or LR(*k*) parsers) **covered later**

FYI Section

Intro. to Top Down Parsing Methods

* Simplest method is a full-backup, *recursive descent* parser
* Often used for parsing simple languages
* Write recursive recognizers (subroutines) for each grammar rule
  + If rules succeeds perform some action (i.e., build a tree node, emit code, etc.)
  + If rule fails, return failure.  Caller may try another choice or fail
  + On failure it “backs up”, tries another pro

|  |
| --- |
| **Top Down Example on Paper** |
| S -> NP VP    VP -> V NP | V NP PP    PP -> P NP    V -> "saw" | "ate" | "walked"    NP -> "John" | "Mary" | "Bob" | Det N | Det N PP    Det -> "a" | "an" | "the" | "my"    N -> "man" | "dog" | "cat" | "telescope" | "park"    P -> "in" | "on" | "by" | "with" |
| RecursiveDescentParsingExample  notice by nature it’s Left Most!!  Create another sentence that will work, and one that does not (but may look close) |

Converting an entire Grammar to Right recursive

Try if for the entire grammar:

|  |  |
| --- | --- |
| exp --> exp addop term | term  addop --> + | -  term --> term mulop factor | factor  mulop --> \*  factor --> ( exp ) | number | Answer: |

Parsing complexity

* Parsing an ***arbitrary CFG*** is O(n3) --  it can take time proportional the cube of the number  of input symbols
* If we constrain the grammar somewhat, we can always parse in linear time.
* Linear-time parsing
  + LL parsers
    - Recognize LL grammar
    - Use a top-down strategy
  + LR parsers
    - Recognize LR grammar
    - Use a bottom-up strategy
* LL vs LR we will cover in a minute
  + super easy

Which is faster?? O(n3) or O(n)?

<Unfinished>

* Here how you can confirm your Shift Reduce Parsing answer
  + Draw a left most parse tree
  + flip it
  + compare to answer in Parse Stack by level

|  |  |  |
| --- | --- | --- |
| **Confirming your LR(1) Shift Reduce** | | |
| Regular Parse Tree | Inverted Parse Tree | Parse Stack in Shift Reduce Answer |
|  |  |  |
|  |  |  |

Now DRAW how this happens!!! (Hint: The parser builds up the [parse tree](http://en.wikipedia.org/wiki/Parse_tree) incrementally, bottom up, and left to right. Start with the input at the very bottom)

Answers Section

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | rule | tree | input | what happened? |
| 10 | c | TopDownExample10 | [] **int** \* int | no match |
| 11 | d | TopDownExample11topDownParseExampleGrammar11 | [] int **\*** int | matched one, but not the rest!!! And no option to proceed!! |
| 12 | e | TopDownExample12topDownParseExampleGrammar12 | [] int \* **int** | great, have 2 matches, but we now need to move to our 2nd Production of T!! |
| 13 | c | TopDownExample13topDownParseExampleGrammar13 | [] int \* **int** | no match on (, move on |
| 14 |  | TopDownExample14topDownParseExampleGrammar14 | [] int \* int | MATCHES and no nodes unaccounted for!! |

Top Down with Issues:

|  |  |
| --- | --- |
| **Top Down Example #2** | |
| Given Grammar | Given Input String |
| a   expr → expr  +  term  b              |   expr   -  term  c              |   term  d   term → term  \*  factor  e              |   term  /  factor  f              |   factor  g   factor → num  h             |   id       // (identifier) | id - num \* id |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | rule | tree | input | what happened? |
| 1 | - | TopDownExampleWithIssues1topDownParseExampleGrammarWithIssues1 | []**id** - num \* id | start production |
| 2 | a | TopDownExampleWithIssues2topDownParseExampleGrammarWithIssues1 | []**id** - num \* id |  |
| 3 | a | TopDownExampleWithIssues3topDownParseExampleGrammarWithIssues1 | []**id** - num \* id |  |
| 4 | a | TopDownExampleWithIssues4topDownParseExampleGrammarWithIssues1 | []**id** - num \* id |  |
| 5 | a | and it keeps on going!! | []**id** - num \* id | wait a minute!!! |

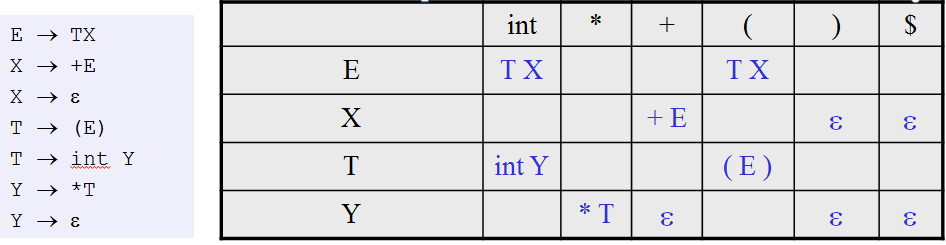
So what’s the issue??

The grammar is LEFT recursive!!! Now you know why top down will only work if the grammar is RIGHT recursive!!

Left Recursion Work??

|  |  |
| --- | --- |
| **term + term - term** | |
| ***expr*   →  *expr  +  term***  ***|   expr   -  term***  ***|   term*** | ***expr*    →   *term  expr2***  ***expr2*  →   *+  term expr2***  ***|     -  term expr2***  ***|    Ɛ*** |
| LeftRecursionRemovalProofA1 | LeftRecursionRemovalProofA1 |
| LeftRecursionRemovalProofA2 | LeftRecursionRemovalProofB2 |
| LeftRecursionRemovalProofA3 | LeftRecursionRemovalProofB3 |
| LeftRecursionRemovalProofA4 | LeftRecursionRemovalProofB4 |
|  | LeftRecursionRemovalProofB5 |

LL1 Predictive parsing #1



**Stack Input Action**

E $ int \* int + int $ pop(); push(T X);

T X $ int \* int + int $ pop(); push(int Y);

int Y X $ int \* int + int $ pop(); next++;

Y X $ \* int + int $ pop(); push(\* T);

\* T X $ \* int + int $ pop(); next++;

T X $ int + int $ pop(); push(int Y);

int Y X $ int + int $ pop(); next++;

Y X $ + int $ pop(); // no push Ɛ

X $ + int $ pop(); push(+ E);

+ E $ + int $ pop(); next++;

E $ int $ pop(); push(T X);

T X $ int $ pop(); push(int Y);

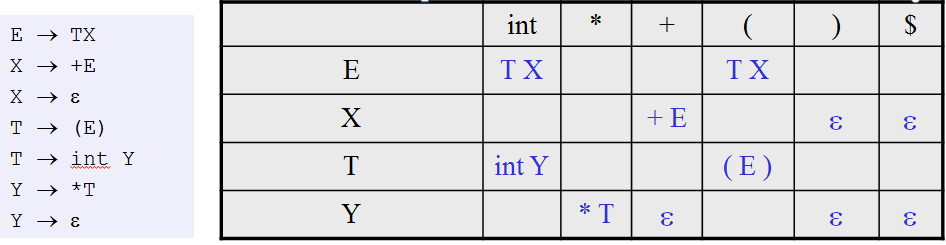
int Y X $ int $ pop(); next++;

Y X $ $ pop(); // no push Ɛ

X $ $ pop(); // no push Ɛ

$ $ ACCEPT!!!

LL1 Predictive parsing #2



**Stack Input Action**

E $ int \* int + int $ pop(); push(T X);

|  |
| --- |
| **LR Shift Reduce 1 Exercise #1** |
|  |

LR Shift Reduce 1 Exercise

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Step** | **Parse Stack** | **Look Ahead** | **Unscanned** | **Parser Action** | |
| 0 | *empty* |  |  |  |  |
| 1 |  |  |  |  |  |
| 2 |  |  |  |  |  |
| 3 |  |  |  |  |  |
| 4 |  |  |  |  |  |
| 5 |  |  |  |  |  |
| 6 |  |  |  |  |  |
| 7 |  |  |  |  |  |
| 8 |  |  |  |  |  |
| 9 |  |  |  |  |  |
| 10 |  |  |  |  |  |
| 11 |  |  |  |  |  |
| 12 |  |  |  |  |  |
| 13 |  |  |  |  |  |
| 14 |  |  |  |  |  |
| 15 |  |  |  |  |  |
| 16 |  |  |  |  |  |
| 17 |  |  |  |  |  |

|  |  |
| --- | --- |
| Grammar | Input String |
| Assign ← *id* = Sums  Sums ← Sums + Products  Sums ← Products  Products ← Products \* Value  Products ← Value  Value ← *int*  Value ← *id* | id = id + id \* int $ |
| **Shift(S)** | 1. lookahead has a partial Grammar match, wait for more 2. combo of lookahead and rightmost in parse stack has a partial Grammar match, wait for more |
| **Parse(P)** | no match for lookahead, reduce rightmost symbol in stack (find largest match) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Step** | **Parse Stack** | **Look Ahead** | **Unscanned** | **Parser Action** | |
| 0 | *empty* | *id* | = id + id\*int $ | - | Just the start |
| 1 | *id* | *=* | id + id\*int $ | S(b) | Shift, Parser wants as big of a match as possible |
| 2 | *id =* | *id* | + id \* int $ | S(a) | Shift, Parser wants as big of a match as possible |
| 3 | *id = id* | *+* | id \* int $ | P | Reduce “id” to “Value” |
| 4 | *id = Value* | *+* | id \* int $ | P | Reduce “Value” to “Products” |
| 5 | *id = Products* | *+* | id \* int $ | P | Reduce “Products” to “Sums” |
| 6 | *id = Sums* | *+* | id \* int $ | S(b) | Shift, since more to go (lookahead) |
| 7 | *id = Sums +* | *id* | \* int $ | S(a) | Shift, since Sums + is a partial match, parser wants to see more |
| 8 | *id = Sums + id* | *\** | int $ | P | Reduce “id” to “Value” |
| 9 | *id = Sums + Value* | *\** | int $ | P | Reduce “Value” to “Products” |
| 10 | *id = Sums + Products* | *\** | int $ | S(b) | Shift, still have more to do |
| 11 | *id = Sums + Products \** | *int* | $ | S(a) | Shift, since int is a match |
| 12 | *id = Sums + Products \* int* | *$* |  | P | Reduce “int” to “Value” |
| 13 | *id = Sums + Products \* Value* | *$* |  | P | Reduce “Products \* Value” to “Products” |
| 14 | *id = Sums + Products* | *$* |  | P | Reduce “Sums + Products” to “Sums” |
| 15 | *id = Sums* | *$* |  | P | Reduce “id = Sums” to “Assign” |
| 16 | *Assign* | *$* |  |  | ACCEPT |
| 17 | *Assign* | *$* |  | P |  |

Resources:

Top Down:

<http://www.cs.wright.edu/~tkprasad/courses/cs780/L101TDP.pdf>

[www.cs.virginia.edu/kim/courses/cs671/lec/cs671-04-**topdown**.ppt](http://www.cs.virginia.edu/kim/courses/cs671/lec/cs671-04-topdown.ppt)

Bottom up

[www.cs.virginia.edu/kim/courses/**cs671**/lec/**cs671**-05-**bottomup**.ppt](http://www.cs.virginia.edu/kim/courses/cs671/lec/cs671-05-bottomup.ppt)

<http://www.cs.wright.edu/~tkprasad/courses/cs780/L12BUP.pdf>

Sebesta

<http://nltk.googlecode.com/svn/trunk/doc/book/ch08.html>

[http://www.cs.sun.ac.za/rw711/2012term2/lectures/lec9ε2010/l9ε2010.pdf](http://www.cs.sun.ac.za/rw711/2012term2/lectures/lec9_2010/l9_2010.pdf)

[http://en.wikipedia.org/wiki/Leftεrecursion](http://en.wikipedia.org/wiki/Left_recursion)

<http://www.cs.nott.ac.uk/~nhn/G52MAL/LectureNotes/lecture18-9up.pdf>

Syntax Analyser

<http://homepages.cwi.nl/~daybuild/daily-books/learning-about/syntax-analysis/syntax-analysis.html>

Shift Reduce

U of Virginia CS 671

Thanks to Dr. Kim Hazelwood for some of her notes

How Shift Reduce it uses the Stack

<http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-035-computer-language-engineering-spring-2010/lecture-notes/MIT6_035S10_lec03.pdf>

LL(1)

<https://courses.cs.washington.edu/courses/cse401/04sp/slides/03b-LL1-example.pdf>

LR(1)

<http://www.montefiore.ulg.ac.be/~geurts/Cours/compil/2012/03-syntaxanalysis-2-2012-2013.pdf>  (pg 28)

Sources:

http://en.wikipedia.org/wiki/Shift-reduce\_parser