Top Down Parsing

Outline
- Top-down parsing
- SLL(1) grammars
- Transforming a grammar into SLL(1) form
- Recursive-descent parsing

Where We Are

SLL(1) Parsing
- SLL(1) parsing goal
  - Determine a Leftmost derivation of the input while reading the input from Left to right while looking ahead at most 1 input token
  - Beginning with the start symbol, grow a parse tree topdown in left-to-right preorder while looking ahead at most 1 input token beyond the input prefix matched by the parse tree derived so far
Expression Grammar

- Consider the grammar
  \[ E \rightarrow (E + E) \mid \text{int} \]
  and the string \((2 + 3)\)
  \[ E \Rightarrow (E + E) \Rightarrow (2 + E) \Rightarrow (2 + 3) \]
- How could we decide between
  \[ E \Rightarrow \text{int} \]
  \[ E \Rightarrow (E + E) \]
in the first derivation step?
- Examine the next unread token in the input
  - If it is an integer, use the production \( E \Rightarrow \text{int} \)
  - If it is '\(', use the production \( E \Rightarrow (E + E) \)
  - Otherwise, parsing error.
- This rule works for all derivation steps, not just the first.
- Next unread token in input is called “look-ahead”
- For this grammar, a look-ahead of one token is enough to let us parse strings in the language.

Non-SLL(1) Grammar

- Consider the grammar
  \[ S \rightarrow E + S \mid E \]
  \[ E \rightarrow \text{num} \mid (S) \]
- and the two derivations
  \[ S \Rightarrow E \Rightarrow (S) \Rightarrow (E) \Rightarrow (3) \]
  \[ S \Rightarrow E + S \Rightarrow (S) + S \Rightarrow (E) + E \Rightarrow (3) + E \Rightarrow (3) + 4 \]
- How could we decide between
  \[ S \Rightarrow E \]
  \[ S \Rightarrow E + S \]
as the first derivation step based on finite number of lookahead symbols?
- We can’t!
  - The sample grammar is not SLL(1)
  - The sample grammar is not SLL(k) for any k.

Making a grammar SLL(1)

- Problem: can’t decide which \( S \) production to apply until we see symbol after first expression
- Left-factoring: Factor common \( S \) prefix \( E \), add new non-terminal \( S' \) for what follows that prefix
- Also: convert left-recursion to right-recursion

Predictive Parsing

- SLL(1) grammar \( G = \langle V, \Sigma, S, \rightarrow \rangle \)
  - For a given nonterminal, the look-ahead symbol uniquely determines the production to apply
  - Top-down parsing a.k.a. predictive parsing
  - Driven by predictive parsing table that maps \( V \times (\Sigma \cup \{e\}) \) to the production to apply (or error)
Using Table

<table>
<thead>
<tr>
<th>R</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E+E)</td>
<td>(1+2)</td>
<td>(E+E)</td>
</tr>
<tr>
<td>(E+E)</td>
<td>(1+2)</td>
<td>(E+E)</td>
</tr>
<tr>
<td>(int+E)</td>
<td>1</td>
<td>(1+2)</td>
</tr>
<tr>
<td>(int+E)</td>
<td>1</td>
<td>(1+2)</td>
</tr>
<tr>
<td>(int+int)</td>
<td>+</td>
<td>(1+2)</td>
</tr>
<tr>
<td>(int+int)</td>
<td>+</td>
<td>(1+2)</td>
</tr>
<tr>
<td>(int+int)</td>
<td>2</td>
<td>(1+2)</td>
</tr>
<tr>
<td>(int+int)</td>
<td>2</td>
<td>(1+2)</td>
</tr>
<tr>
<td>(int+int)</td>
<td>EOF</td>
<td>(1+2)</td>
</tr>
</tbody>
</table>

Using Table

<table>
<thead>
<tr>
<th>R</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E+E)</td>
<td>(1+2)</td>
<td>(E+E)</td>
</tr>
<tr>
<td>(E+E)</td>
<td>(1+2)</td>
<td>(E+E)</td>
</tr>
<tr>
<td>(int+E)</td>
<td>1</td>
<td>(1+2)</td>
</tr>
<tr>
<td>(int+E)</td>
<td>1</td>
<td>(1+2)</td>
</tr>
<tr>
<td>(int+int)</td>
<td>+</td>
<td>(1+2)</td>
</tr>
<tr>
<td>(int+int)</td>
<td>+</td>
<td>(1+2)</td>
</tr>
<tr>
<td>(int+int)</td>
<td>2</td>
<td>(1+2)</td>
</tr>
<tr>
<td>(int+int)</td>
<td>2</td>
<td>(1+2)</td>
</tr>
<tr>
<td>(int+int)</td>
<td>EOF</td>
<td>(1+2)</td>
</tr>
</tbody>
</table>

How to Implement, Version 1

- A table-driven parser

```c
void parse(nonterminal A) { 
    int i;
    let A -> X_0...X_n = TABLE[A,token] 
    for (i=0; i<=n; i++) { 
        if (X_i in Σ) 
            if (token == X_i) token = input.read(); 
            else throw new ParseError(); 
        else parse(X_i); 
    } 
    return; 
}
```

How to Implement, Version 2

- Convert table into a recursive-descent parser

```c
num + ( ) EOF 
S -> ES' 
E -> num -> (S) 
S' -> +S -> (ES) -> c 
```

- Three procedures: parse_S, parse_S', parse_E
Recursive-Descent Parser

```java
void parse_S() {
    // ... lookahead token
    switch (token) {
        case num: parse_E(); parse_S(); return;
        case '(': parse_E(); parse_S(); return;
        default: throw new ParseError();
    }
}
```

S → E S' | c
S' → + S | c
E → num | (S)

Recursive-Descent Parser

```java
void parse_E() {
    switch (token) {
        case number: token = input.read(); return;
        case '(': token = input.read(); parse_S();
            if (token != ')') throw new ParseError();
            token = input.read(); return;
        default: throw new ParseError();
    }
}
```

E → num | (S)

Recursive-Descent Parser

```java
void parse_S'() {
    switch (token) {
        case '+': token = input.read(); parse_S(); return;
        case ')': return;
        case EOF: return;
        default: throw new ParseError();
    }
}
```

S → E S' | c
S' → + S | c
E → num | (S)

Recursive-Descent Parser

```java
Call Tree = Parse Tree
```

```
parse_S
   /  \                  (1+2+(3+4))+5
parse_E   parse_S'      S
parse_S   parse_S'      S
parse_E   parse_S'      S
parse_S   parse_S'      S
parse_S   parse_S'      S
parse_S   parse_S'      S
parse_S   parse_S'      S
```
How to Construct Parsing Tables

• Needed: algorithm for automatically generating a predictive parsing table from a grammar

• Easy for Bali

\[
\begin{align*}
S &\rightarrow ES' \\
S' &\rightarrow \varepsilon | +S \\
E &\rightarrow \text{num} | (S)
\end{align*}
\]

Filling in parsing table

• First, consider grammars that do not have \(\varepsilon\)-productions.

• Suppose we have a non-terminal \(E\) for which all productions begin with a terminal symbol
  - (eg) \(E \rightarrow aXY | bYY | stX\)
  - Easy to fill in parsing table
    - in Table\([E,a]\) put rule \(E \rightarrow aXY\) etc.

• What if we have a non-terminal in which some rules begin with a non-terminal?
  - (eg) \(E \rightarrow aXY | YZ\)
  - For what look-ahead symbols should we use the rule \(E \rightarrow YZ\)?
    - Obvious answer: for any terminal symbol that begins a string that can be derived from \(Y\)

---

Formalization

• Formalize intuition using FIRST relation

  \(\text{FIRST}(E) \subseteq T\) (recall: \(T\) is the set of terminals)
  - terminal symbol \(t\) is in \(\text{FIRST}(E)\) iff there is a string starting with \(t\) that can be derived from \(E\)
  - useful to extend \(\text{FIRST}\) to terminal symbols
    - by definition \(\text{FIRST}(t) = \{t\}\)
  - in general, \(\text{FIRST}(\alpha)\) contains terminal \(t\) if there is a string starting with \(t\) that can be derived from \(\alpha\)

  \[
  (A \rightarrow Y_1 \ldots Y_n) \in P \quad t \in \text{FIRST}(Y_i) \\
  \frac{A \rightarrow \varepsilon}{t \in \text{FIRST}(A)}
  \]

• Example
  - \(S \rightarrow AB|BA\ A \rightarrow B|a\ B \rightarrow xa\)
  - \(\text{FIRST}(A) = \{t, a\} \text{ FIRST}(B) = \{x\} \text{ FIRST}(S) = \{t, a, x\}\)

---

General picture

• Need to compute three relations called NULLABLE, FIRST and FOLLOW

  • NULLABLE:
    - set of non-terminals that can be rewritten to the empty string

  \(\text{FIRST}(\alpha) \subseteq T \cup \{\varepsilon\}\)
    - if \(\alpha\) can be rewritten to a string starting with terminal \(t\), then \(t\) is in \(\text{FIRST}(\alpha)\)
    - if \(\alpha\) can be rewritten to \(\varepsilon\), then \(\varepsilon\) is in \(\text{FIRST}(\alpha)\)

  • FOLLOW(A) \(\subseteq T\)
    - set of terminals that can follow A in a sentential form
Constructing parsing table

- Enter production $A \rightarrow Y_1...Y_n$ into Table[$A$, $t$] for all $t$ such that
  - $t \in \text{FIRST}(Y_1...Y_n)$
  - $\varepsilon \in \text{FIRST}(Y_1...Y_n)$ and $t \in \text{FOLLOW}(A)$

NULLABLE and FIRST

$G = \langle N, T, P, S \rangle$ (N: set of non-terminals, T: set of terminals, P: set of productions, S: start symbol)

Notation: $t$ will stand for an element of $T$.

$a$ will denote a string of terminals and non-terminals (possibly empty)

### NULLABLE

- $A \in \text{NULLABLE}$
- $(A \rightarrow \varepsilon) \in P$
- $(A \rightarrow Y_1...Y_n) \in P \quad Y_1...Y_n \in \text{NULLABLE}$

### FIRST

- $A \in \text{NULLABLE}$
- $t \in \text{FOLLOW}(A)$
- $A \in \text{NULLABLE}$
- $(A \rightarrow Y_1...Y_n) \in P 
  \begin{array}{ll}
  Y_1...Y_n \in \text{NULLABLE} & t \in \text{FIRST}(Y_k...Y_n) \\
  t \in \text{FIRST}(A)
  \end{array}$

Extend FIRST to strings

- Helper function: $*$, binary operation on subsets of $T \cup \{\varepsilon\}$
- Concatenate and truncate to 1 symbol
- $(\cdot, a, b) + 1 (\cdot, a) = (\cdot, a, b)$
- $\text{FIRST}(\cdot) = \{\varepsilon\}$
- $\text{FIRST}([t]) = \{t\}$
- $\text{FIRST}(Y_1...Y_j) = \text{FIRST}(Y_1) + 1 \, \text{FIRST}(Y_2) + 1 \, ... \, \text{FIRST}(Y_j)$

FOLLOW

- $\text{FOLLOW}(A) \subseteq T \cup \{\varepsilon\}$
- $\varepsilon \in \text{FOLLOW}(S)$
- $(A \rightarrow aB_1...Y_n) \in P 
  \begin{array}{ll}
  t \in \text{FIRST}(Y_1...Y_n) & t \in \text{FOLLOW}(B) \\
  t \in \text{FOLLOW}(A)
  \end{array}$
- $(A \rightarrow aB_1...Y_n) \in P 
  \begin{array}{ll}
  t \in \text{FIRST}(Y_1...Y_n) & t \in \text{FOLLOW}(A) \\
  t \in \text{FOLLOW}(B)
  \end{array}$
### Non-trivial example

- **Grammar for arithmetic expressions**
  
  \[
  \begin{align*}
  S & \rightarrow E \\
  E & \rightarrow E + T | T \\
  T & \rightarrow T * F | F \\
  F & \rightarrow (E) | \text{int}
  \end{align*}
  \]

- **Grammar is not LLI(1)**

- **Massaged grammar**
  
  \[
  \begin{align*}
  S & \rightarrow E \\
  E & \rightarrow T E' \\
  E' & \rightarrow + T E' | \epsilon \\
  T & \rightarrow F T' \\
  T' & \rightarrow * F T' | \epsilon \\
  F & \rightarrow (E) | \text{int}
  \end{align*}
  \]

- **Nullable = \{E', T'\}**
- **FIRST**
  
  \[
  \begin{align*}
  S & \rightarrow \{S \rightarrow E S \} \\
  E & \rightarrow \{E \rightarrow T E, E \rightarrow \epsilon \} \\
  T & \rightarrow \{T \rightarrow F T, T \rightarrow \epsilon \} \\
  F & \rightarrow \{F \rightarrow (E), F \rightarrow \text{int} \}
  \end{align*}
  \]

### Summary

- **SLL(k) grammars**
  - left-to-right scanning
  - leftmost derivation
  - can determine what production to apply from the next k symbols
  - Can automatically build predictive parsing tables

- **Predictive parsers**
  - Can be easily built for SLL(1) grammars from the parsing tables
  - Also called recursive-descent, or top-down parsers

### Non-SLL(1) grammars

- **Construction of predictive parse table for grammar results in conflicts**

  \[
  S \rightarrow S + S | S * S | \text{num}
  \]

  \[
  \text{FIRST}(S+S) = \text{FIRST}(S*S) = \text{FIRST}(\text{num}) = \{ \text{num} \}
  \]

### Assignment

- **For Bali grammar, we can write simple recursive-descent parser that consists of a set of mutually recursive procedures**
  - one procedure for each non-terminal in the grammar
  - responsible for reading in a substring and parsing it as that non-terminal
  - body of procedure is a switch/case statement that looks at the next unread terminal symbol and decides which production to use for that non-terminal

- **Hand-crafted recursive-descent parsers can handle some non-SLL(1) grammars using ad hoc techniques**
  - more difficult to do in table-driven approach
Helper class: SamTokenizer

• Read the on-line code for
  – Tokenizer: interface
  – SamTokenizer: code

• Code lets you
  – open file for input:
    • SamTokenizer f = new SamTokenizer(String-for-file-name)
  – examine what the next thing in file is: f.peekAtKind()
    • TokenType: enum {INTEGER, FLOAT, WORD, OPERATOR,…}
      – INTEGER: such as 3, -34, 46
      – WORD: such as x, r45, y78z (variable name in Java)
      – OPERATOR: such as +, -, *, ( , ) , etc.
  – read next thing from file (or throw TokenizerException):
    • f.getInt/peekInt:() \rightarrow int
    • f.getWord/peekWord:() \rightarrow String
    • f.getOp/peekOp:() \rightarrow char
    • get eats up token from file, while peek does not advance the pointer into the file

• Useful methods in SamTokenizer class:
  – f.check(char c): char \rightarrow boolean
    • Check if next thing in input is c
      – if so, eat it up and return true
      – otherwise, return false
  – f.check(String s): String \rightarrow boolean
    • Example of its use: f.check("x");
    • Check if next word in input matches s
      – if so, eat it up and return true
      – otherwise, return false
  – f.checkInt(): \rightarrow boolean
    • check if next token is an integer and if so, eat it up and return true
    • otherwise, return false
  – f.match(char c): char \rightarrow void
    • like f.check but throws TokenizerException if next token in input is not "c"
  – f.match(String s): string \rightarrow void
    • (e.g.) f.match("if")

Parser for simple expressions

Expression \rightarrow integer
Expression \rightarrow ( Expression + Expression )

• Input: file
• Output: true if a file contains a single expression as defined by this grammar, false otherwise
• Note: file must contain exactly one expression
  File: (2+3) (3+4)
  will return false

Parser for expression language

static boolean expParser(String fileName) {//parser for expression in file
  try
  {
    SamTokenizer f = new SamTokenizer(fileName);
    return getExp(f) && (f.peekAtKind() == Tokenizer.TokenType.EOF) ;//must be at EOF
  }
  catch (Exception e) {
    System.out.println("Aaargh");
    return false
  }
}
static boolean getExp(SamTokenizer f){
  switch (f.peekAtKind()){  
    case INTEGER: //E \rightarrow integer
      {f.checkInt();
       return true
     }
    case OPERATOR: //E \rightarrow (E+E)
      return f.check('(') &&
             getExp(f) &&
             f.check('+') &&
             getExp(f) &&
             f.check(')');
    default:
      return false;
  }
}
Note on boolean operators

• Java supports two kinds of boolean operators:
  – E1 & E2:
    • Evaluate both E1 and E2 and compute their conjunction (i.e., “and”)
  – E1 && E2:
    • Evaluate E1. If E1 is false, E2 is not evaluated, and value of expression is false. If E1 is true, E2 is evaluated, and value of expression is the conjunction of the values of E1 and E2.

• In our parser code, we use &&
  – if “f.check(‘) returns false, we simply return false without trying to read anything more from input file. This gives a graceful way to handling errors.

Modifying parser to do SaM code generation

• Let us modify the parser so that it generates SaM code to evaluate arithmetic expressions: (eg)

  2 : PUSHIMM 2
  STOP

  (2 + 3) : PUSHIMM 2
   PUSHIMM 3
   ADD
   STOP

Tracing recursive calls to getExp

- Recursive method getExp should return a string containing SaM code for expression it has parsed.
- Top-level method expParser should tack on a STOP command after code it receives from getExp.
- Method getExp generates code in a recursive way:
  – For integer i, it returns string “PUSHIMM” + i + “STOP”
  – For (E1 + E2),
    • recursive calls return code for E1 and E2
      – say those are strings S1 and S2
    • method returns S1 + S2 + “ADDn”

Idea
CodeGen for expression language

```java
static String expCodeGen(String fileName) { //returns SaM code for expression in file
    try {
        SamTokenizer f = new SamTokenizer(fileName);
        String pgm = getExp(f);
        return pgm + "STOP\n";
    } catch (Exception e) {
        System.out.println("Aaargh");
        return "STOP\n";
    }
}

def getExp(SamTokenizer f) {
    switch (f.peekAtKind()) {
        case INTEGER: //E -> integer
            return "PUSHIMM " + f.getInt() + "\n";
        case OPERATOR: //E ->(E+E)
            f.match('('); // must be '('
            String s1 = getExp(f);
            f.match('+'); //must be '+'
            String s2 = getExp(f);
            f.match(')'); //must be ')'
            return s1 + s2 + "ADD\n";
        default:
            return "ERROR\n";
    }
}
```

Tracing recursive calls to getExp

<table>
<thead>
<tr>
<th>Expression</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 + (34 + 23)</td>
<td>PUSHIMM 3  PUSHIMM 34  ADD  ADD</td>
</tr>
<tr>
<td>(3 + (34 + 23))</td>
<td>PUSHIMM 34  PUSHIMM 3 ADD  ADD</td>
</tr>
<tr>
<td>(3 + (34 + 23))</td>
<td>PUSHIMM 34  PUSHIMM 23  ADD  ADD</td>
</tr>
<tr>
<td>(3 + (34 + 23))</td>
<td>PUSHIMM 34  PUSHIMM 23  ADD  ADD</td>
</tr>
</tbody>
</table>

Top-Down Parsing

- We can use recursive descent to build an abstract syntax tree too

Creating the AST

```java
abstract class Expr { }

class Add extends Expr {
    Expr left, right;
    Add(Expr L, Expr R) {
        left = L; right = R;
    }
}

class Num extends Expr {
    int value;
    Num (int v) { value = v; }
}
```
AST Representation

(1 + 2 + (3 + 4)) + 5

How can we generate this structure during recursive-descent parsing?

Creating the AST

• Just add code to each parsing routine to create the appropriate nodes!

• Works because parse tree and call tree have same shape

• parse_S, parse_S’, parse_E all return an Expr:

```java
void parse_E() Expr parse_E()
void parse_S() Expr parse_S()
void parse_S'() Expr parse_S'()
```

AST Creation: parse_E

```java
Expr parse_E() {
    switch (token) {
        case num: // E -> num
            Expr result = Num (token.value);
            token = input.read();
            return result;
        case '(': // E -> ( S )
            token = input.read();
            Expr result = parse_S();
            if (token != ')')
                throw new ParseError();
            token = input.read();
            return result;
        default:
            throw new ParseError();
    }
}
```

Conclusion

• There is a systematic way of generating parsing tables for recursive-descent parsers
• Recursive descent parsers were among the first parsers invented by compiler writers
• Ideally, we would like to be able generate parsers directly from the grammar
  − software maintenance would be much easier
    • maintain the “parser-generator” for everyone
    • maintain specification of your grammar
• Today we have lots of tools that can generate parsers automatically from many grammars
  − JavaCC, ANTLR: produce recursive descent parsers from suitable grammars