Top Down Parsing
Where We Are

Source code (character stream)

if (b == 0) a = b;

Token stream

if ( b == 0 ) a = b ;

Abstract Syntax Tree (AST)

== if ==

Semiotic Analysis

Syntax Analysis (Parsing)

Lexical Analysis
Outline

• Inference rules for computing grammar properties
• Top-down parsing
• SLL(1) grammars
• Recursive-descent parsing
• Generating parsing tables for SLL(1) grammars
Concepts you should know

• Distinction between language and grammar
  – Language: set of strings over some alphabet
  – Grammar: a set of rules for generating the strings in a language
  – G: grammar, L(G): language generated by grammar

• Recognition vs. parsing, given grammar G and word w
  – Recognition is decision problem - is w in L(G)?
  – Parsing: if w in L(G), show a derivation (proof)

• Context-free grammar: 4-tuple
  – S: Start symbol
  – T: Terminals aka tokens (also written as Σ by some authors)
  – N: Non-terminals (also written as V)
  – P: Productions

• Sentential forms and sentences
  – Sentential form: string that can be obtained by starting with S and using productions as rewrite rules to rewrite nonterminals
  – Sentence: sentential form without nonterminals (word in language)
Concepts you should know

• Derivation of string using grammar
  – Start from S and repeatedly rewrite a nonterminal using the productions of the grammar until there are no nonterminals left
  – Leftmost/rightmost-derivation: rewrite only the leftmost/rightmost nonterminal at each step

• Ambiguous grammar
  – Grammar in which there are two or more leftmost derivations for some word
Inference rules
• Given grammar, we need to compute certain sets that will be used by parser
• These sets are usually specified recursively
  – (e.g.) if A is a member of the set, then B is also a member of that set
• Inference rules are an elegant way to specify these recursive sets without writing code
• From inference rules, it is easy to write down code
Parsing SLL(1) grammars

- Compute relations NULLABLE, FIRST and FOLLOW
- **NULLABLE** ⊆ **N**
  - set of non-terminals that can be rewritten to the empty string
- **FIRST(A)** ⊆ **T** U {ε}
  - if A can be rewritten to a string starting with terminal t, then t is in **FIRST(A)**
  - if A can be rewritten to ε, then ε is in **FIRST(A)**
- **FOLLOW(A)** ⊆ **T**
  - set of terminals that can follow A in a sentential form
    t ∈ **FOLLOW(A)** if we can derive a string ...At...
- These relations are defined for any context-free grammar but if grammar is SLL(1), they can be used to implement a recursive-descent parser.
NULLABLE

• ε–production
  – A production whose righthand side is the empty string ε
  – (e.g.) \( A \rightarrow \varepsilon \)
• Nullable non-terminal
  • Nonterminal that can be rewritten to \( \varepsilon \)
• NULLABLE: set of nullable nonterminals
  – If there is production \( A \rightarrow \varepsilon \), then \( A \in \text{NULLABLE} \)
  – If there is production \( A \rightarrow Y_1..Y_n \) and \( Y_i \in \text{NULLABLE} \) for all \( Y_i \), then \( A \in \text{NULLABLE} \)

Example

\[
\begin{align*}
S & \rightarrow A\$
\quad A \rightarrow BC \mid x \\
B & \rightarrow t \mid \varepsilon \\
C & \rightarrow v \mid \varepsilon
\end{align*}
\]

\( \text{NULLABLE} = \{B,C,A\} \)
Inference rules for NULLABLE

- This can be specified compactly using inference rules

\[
\begin{align*}
\text{NULLABLE} \subseteq N \\
(A \rightarrow \varepsilon) \in P & \quad \Rightarrow \quad A \in \text{NULLABLE} \\
(A \rightarrow Y_1 \ldots Y_n) \in P & \quad Y_1, \ldots, Y_n \in \text{NULLABLE} & \quad \Rightarrow \quad A \in \text{NULLABLE}
\end{align*}
\]

- Given proposal \(X \subseteq N\), we can check if proposal is consistent with inference rules.

- Examples:
  - \{\}\: no
  - \{A\}: no
  - \{A,B,C\}: yes
  - \{S,,B,C\}: yes

- For our problems, we want the smallest set that is consistent with all inference rules
  - \{A,B,C\}

Grammar

\[
\begin{align*}
S & \rightarrow A$ \\
A & \rightarrow BC \mid x \\
B & \rightarrow t \mid \varepsilon \\
C & \rightarrow v \mid \varepsilon
\end{align*}
\]
Computing NULLABLE

- Initialize NULLABLE to {}.
- Round-based computation:
  - In each round, visit every production and see if you can add more elements to NULLABLE.
  - Terminate when NULLABLE does not change in some round.

\[
\begin{align*}
\text{NULLABLE} & \subseteq N & \text{Smallest set for which} \\
(A \rightarrow \varepsilon) \in P & \implies A \in \text{NULLABLE} \\
(A \rightarrow Y_1..Y_n) \in P & \implies Y_1..Y_n \in \text{NULLABLE} \\
& \implies A \in \text{NULLABLE}
\end{align*}
\]

Computing NULLABLE:

\[
\text{NULLABLE} = \{ \} \\
\text{repeat} \\
\quad \text{for each production } A \rightarrow Y_1..Y_n \\
\quad \quad \text{if } Y_1..Y_n = \varepsilon \text{ or all } Y_i \text{ are in NULLABLE, add } A \text{ to NULLABLE} \\
\text{until NULLABLE set does not change}
\]
Slicker way to write inference rules

• Define predicate NULLABLE: string $\rightarrow$ true/false
  - $\text{NULLABLE}(\varepsilon) = \text{true}$
  - $\text{NULLABLE}(t:T) = \text{false}$
  - $\text{NULLABLE}(Y_1..Y_n) = \text{NULLABLE}(Y_1) \& ..\& \text{NULLABLE}(Y_n)$

$$
\begin{align*}
\text{NULLABLE} & \subseteq N & \text{Smallest set for which} \\
(A \rightarrow \varepsilon) \in P & \quad \Rightarrow \quad A \in \text{NULLABLE} & (A \rightarrow Y_1..Y_n) \in P & \quad Y_1,..Y_n \in \text{NULLABLE} & \quad \Rightarrow \quad A \in \text{NULLABLE}
\end{align*}
$$

$$
\begin{align*}
\text{NULLABLE} & \subseteq N & \text{Smallest set for which} \\
(A \rightarrow Y_1..Y_n) \in P & \quad \Rightarrow \quad \text{NULLABLE}(Y_1,..Y_n) & \quad \Rightarrow \quad A \in \text{NULLABLE}
\end{align*}
$$
Another relation: FIRST

• \text{FIRST}(A) \subseteq T \cup \{\varepsilon\} \ (A \text{ is nonterminal})
  
  – if \( A \) can be rewritten to a string starting with terminal \( t \), then \( t \in \text{FIRST}(A) \)
  
  – if \( A \) can be rewritten to \( \varepsilon \) (that is, \( \text{NULLABLE}(A) \)), then \( \varepsilon \in \text{FIRST}(A) \)

• Convenient to define \( F = T \cup \{\varepsilon\} \)
  
  – \( f \) denotes element of \( F \) (so either terminal or \( \varepsilon \))

• As with \( \text{NULLABLE} \), it is convenient to extend this to arbitrary strings
General FIRST relation

\[
\text{FIRST}(\alpha) \subseteq F = T \cup \{\varepsilon\}
\]

if \(\alpha\) can be rewritten to a string starting with terminal \(t\),
then \(t\) is in FIRST(\(\alpha\))

if \(\alpha\) can be rewritten to \(\varepsilon\), then \(\varepsilon\) is in FIRST(\(\alpha\))

Helper function: \(+_1\)

binary operation on subsets of \(T \cup \{\varepsilon\}\)
concatenate each element of first set with each element of second set
and truncate to 1 symbol

\(\{\varepsilon, a, b\} +_1 \{\varepsilon, c\} = \{\varepsilon, a, b, c\}\)

FIRST(\(\varepsilon\)) = \{\varepsilon\}
FIRST(t) = \{t\}
FIRST(\(Y_1...Y_n\)) = FIRST(\(Y_1\)) +_1 FIRST(\(Y_2\)) +_1 ...FIRST(\(Y_n\))

So if we can compute FIRST for all nonterminals, we can compute it for any string
Inference rule for FIRST

\[
\text{FIRST}(A) \subseteq F = T \cup \{\epsilon\} \quad \text{Smallest sets for which}
\]

\[
\begin{align*}
(A \rightarrow Y_1Y_2...Y_n) &\in P \\
&\quad f \in \text{FIRST}(Y_1Y_2...Y_n) \\
&\quad f \in \text{FIRST}(A)
\end{align*}
\]

FIRST sets can be computed using iterative algorithm as before

```
for each A:N do {
    FIRST[A] = { };
}
repeat
    for each production A \rightarrow Y_1..Y_n do {
        FIRST[A] = FIRST[A] U FIRST(Y_1..Y_n)
    }
until FIRST sets do not change
```
Example

Grammar

S → A$
A → BC | x
B → t | ε
C → v | ε

for each A:N do {
    FIRST[A] = { };
}
repeat
    for each production A → Y1..Yn do {
        FIRST[A] = FIRST[A] U FIRST(Y1..Yn)
    }
until FIRST sets do not change

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Round</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td>{}</td>
<td>{}</td>
<td>{x}</td>
<td>{x,t,v,$}</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td>{}</td>
<td>{x}</td>
<td>{x,t,v,ε}</td>
<td>{x,t,v,ε}</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>{}</td>
<td>{t,ε}</td>
<td>{t,ε}</td>
<td>{t,ε}</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>{}</td>
<td>{v,ε}</td>
<td>{v,ε}</td>
<td>{v,ε}</td>
</tr>
</tbody>
</table>
FOLLOW

- FOLLOW(B) ⊆ T U {EOF} (EOF is end-of-file)
  - t ∈ FOLLOW(B) if we can derive a string ...Bt...
- By convention, EOF ∈ FOLLOW(S)
  - To avoid confusion, let us denote EOF by $
- First sentential form:
  - S $ so FOLLOW(S) = {$
- Second sentential form.: say we use S → A₁...Aᵢ...Aₘ
  - String becomes A₁..Aᵢ..Aₘ$
  - What is FOLLOW(Aᵢ)?
  - Answer: FIRST(Aᵢ₊₁...Aₘ$) = FIRST(Aᵢ₊₁..Aₘ) +₁ FOLLOW(S)
- In general: production A → X₁..XₖBY₁..Yₘ
  - t ∈ (FIRST(Y₁..Yₘ) +₁ FOLLOW(A)) → t ∈ FOLLOW(B)
Inference rules for FOLLOW

\[
\text{FOLLOW}(A) \subseteq T \cup \{\text{EOF}\}
\]

\[
\text{EOF} \in \text{FOLLOW}(S)
\]

\[
(A \rightarrow X_1..X_kBY_1..Y_n) \in P \quad t \in \text{FIRST}(Y_1..Y_n) + \_1 \text{FOLLOW}(A)
\]

\[
t \in \text{FOLLOW}(B)
\]
Computing all relations

for each A do {
    NULLABLE[A] = false;
    FIRST[A] = FOLLOW[A] = { };  
}

FOLLOW[S] = {EOF};
repeat
  for each production A → Y1..Yn do {
    if NULLABLE(Y1..Yn) then NULLABLE[A] = true;
    FIRST[A] = FIRST[A] U FIRST(Y1..Yn)
    for each Yi do
      FOLLOW[Yi] = FOLLOW[Yi] U (FIRST(Yi+1..Yk) + FOLLOW[A])
  }
until sets do not change
Recursive-descent parsing
SLL(1) Parsing Goal

• **String view:**
  – Determine a **Leftmost derivation** of the input while reading the input from **Left** to right while looking ahead at most 1 input token

• **Tree view:**
  – Beginning with the start symbol, grow a parse tree top-down 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

  Grammar: \[ E \rightarrow (E + E) \mid \text{int} \]
  String: \((2 + 3)\)

• Leftmost derivation

  \[ E \Rightarrow (E + E) \Rightarrow (2 + E) \Rightarrow (2 + 3) \]

• How can we decide which production to use in first step?

  \[ E \rightarrow \text{int} \]
  \[ E \rightarrow (E+E) \]

• Answer: examine next unread token in input. Three cases:

  \( \text{int} \) : use the production \( E \rightarrow \text{int} \)
  \( \text{‘(‘} \) : use the production \( E \rightarrow (E+E) \)
  \text{Otherwise} : parsing error.

• This rule works for all derivation steps, not just the first.

• Next unread token in input is called “look-ahead”
Recursive-Descent Recognizer

\[ E \rightarrow (E + E) \mid \text{int} \]

token = input.read(); //global variable
parse_E();

//precondition: global variable “token” has look-ahead token
void parse_E() {
    switch (token) {
        case int: token = input.read(); return;
        case ‘(‘:
            {token = input.read();
             parse_E();
             if (token != ‘+’) throw new ParseError();
             token = input.read();
             parse_E();
             if (token != ‘)’) throw new ParseError();
             token = input.read(); return;
            }
        default: throw new ParseError(); }

    //postcondition: global variable “token” has look-ahead token
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 one (or even some finite number k) of look-ahead tokens?

• 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)

- **Left-factoring**: Factor common $S$ prefix $E$, add new non-terminal $S'$ for what follows that prefix

- Convert **left-recursion** to right-recursion

- Not all context-free languages have an SLL(1) grammar

```
S → E+S
S → E
E → num
E → (S)

S → ES'
S' → ε
S' → +S
E → num
E → (S)
```
General picture:
parser for SLL(1) grammar

- One procedure for each non-terminal
- Global variable *token* contains look-ahead token
- Procedure for non-terminal N
  - Precondition: variable token has look-ahead token
  - Action of procedure: read in a sequence of terminals that can be derived from non-terminal N
  - Postcondition: variable token has look-ahead token
- Body of procedure is a big case statement
- Each case:
  - one possible look-ahead token (say t)
  - invokes parsing actions for a production of N
- Question: how do we determine which production to use for a given [N,t] combination?
Abstraction: predictive parsing table

Grammar

<table>
<thead>
<tr>
<th></th>
<th>num</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>→ ES'</td>
<td>→ ES'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S'</td>
<td>→ +S</td>
<td></td>
<td>→ ε</td>
<td>→ ε</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>→ num</td>
<td></td>
<td>→ (S)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One row for each non-terminal in V
One column for each terminal in T U {EOF}
Table [r,c] is the production to be used
  when expanding non-terminal r and look-ahead token is c
  (empty table entries: throw parsing error)
Given parsing table, it is easy to generate recursive-descent parser
Recursive-Descent Parser

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

**Grammar:**

```
S → ES'  
S' → +S   →  ε  →  ε  
E → num  →  (S)  
```

**Lookahead token:**

- **num**
- **+**
- **(**
- **EOF**
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();
    }
}
```

The parsing rules for the expression language are as follows:

- **S**: S → ES' → ES'
- **S'**: S' → +S → ε → ε
- **E**: E → num → (S)
- **num**: num
- **+**: +
- **()**: ()
- **EOF**: EOF

These rules define the syntax for a simple expression language with addition and parentheses.
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(); }
}
```

### Grammar

```
S → ES'       → ES'       → EOF
S' → +S       → ε       → ε
E → num       → (S)
num + ( ) EOF
```

Call Tree = Parse Tree

(1+2+(3+4))+5
Constructing parsing tables

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

• Grammar
  – has no $\varepsilon$-productions
  – every production begins with a terminal symbol
  – (eg) $E \rightarrow aXY \mid bYY \mid stX$

• Easy to fill in parsing table
  – in Table[$E,a$] put production $E \rightarrow aXY$ etc.
  – if there are two or more productions in a given spot in table, grammar is not SLL(1)
Generalizing construction (I)

• What if some productions begin with non-terminal (assume no $\varepsilon$-productions)?
  
  Example grammar: $S \rightarrow dx \mid Ay$ $A \rightarrow ax \mid bx$
  
  – For what look-ahead symbols should we use $S \rightarrow Ay$?
  – Obvious answer: for any terminal $t \in \text{FIRST}(A)$

• Constructing parsing table:
  
  – for each production $A \rightarrow Y_1Y_2..Y_n$

    Enter production into Table$[A,t]$ for each terminal $t$ in $\text{FIRST}(Y_1)$
Example

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

$$
\begin{align*}
S & \rightarrow A$ \\
A & \rightarrow BC \mid x \\
B & \rightarrow t \\
C & \rightarrow v \\
\text{FIRST}(A) & = \{x, t\} \\
\text{FIRST}(B) & = \{t\} \\
\text{FIRST}(C) & = \{v\} \\
\text{FIRST}(S) & = \{x, t\}
\end{align*}
$$

$$
\begin{align*}
\text{Table}[S,x] & = S \rightarrow A$ \\
\text{Table}[S,t] & = S \rightarrow A$ \\
\text{Table}[A,t] & = A \rightarrow BC \\
\text{Table}[A,x] & = A \rightarrow x \\
\text{Table}[B,t] & = B \rightarrow t \\
\text{Table}[C,v] & = C \rightarrow v
\end{align*}
$$
Generalizing construction (II)

• Handle $\varepsilon$-productions

Example grammar: $S \rightarrow Ax \quad A \rightarrow ab \mid \varepsilon \quad (A \text{ is nullable})$

Language has two strings \{abx, x\}

$S \rightarrow Ax \rightarrow abx$
$S \rightarrow Ax \rightarrow x$

• Two problems:
  
  – Where should production $A \rightarrow \varepsilon$ go in parsing table??
    • Insert $A \rightarrow \varepsilon$ into Table[A,x] if x in FOLLOW(A)
  
  – Where should $S \rightarrow Ax$ go in parsing table?
    • If we just look at FIRST(A), we miss the fact that the lookahead symbol might also be x because A is NULLABLE
    • Enter production $S \rightarrow Ax$ into Table[A,t] for all t for which $t \in \text{FIRST}(Ax)$
Constructing parsing table

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

• Example

  $S \rightarrow A\$
  $A \rightarrow BC \mid x$
  $B \rightarrow t \mid \epsilon$
  $C \rightarrow v \mid \epsilon$

<table>
<thead>
<tr>
<th>NULLABLE</th>
<th>FIRST(A)</th>
<th>FIRST(B)</th>
<th>FIRST(C)</th>
<th>FIRST(S)</th>
<th>FOLLOW(A)</th>
<th>FOLLOW(B)</th>
<th>FOLLOW(C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{A,B,C}</td>
<td>{x,t,v,\epsilon}</td>
<td>{t,\epsilon}</td>
<td>{v,\epsilon}</td>
<td>{x,t,v,$}</td>
<td>{$}</td>
<td>{v,$}</td>
<td>{$}</td>
</tr>
</tbody>
</table>

  $A \rightarrow BC$ is in table entries for $t,v,\$, $S \rightarrow A\$ is in table entries for $x,t,v,\$,
  $B \rightarrow \epsilon$ is in table entries for $v,\$, $C \rightarrow \epsilon$ is in table entries for $\$. 
Non-trivial example

- Grammar for arithmetic expressions
  \[ S \rightarrow E \, $ \]
  \[ E \rightarrow E + T \mid T \]
  \[ T \rightarrow T * F \mid F \]
  \[ F \rightarrow (E) \mid \text{int} \]
- Grammar is not LL(1)
- Massaged grammar
  \[ S \rightarrow E \, $ \]
  \[ E \rightarrow T \, E' \]
  \[ E' \rightarrow + \, T \, E' \mid \varepsilon \]
  \[ T \rightarrow F \, T' \]
  \[ T' \rightarrow \times \, F \, T' \mid \varepsilon \]
  \[ F \rightarrow (E) \mid \text{int} \]
- Nullable = \{E', T'\}
- FIRST
  - E = \{(\), int\}
- FOLLOW
  - E' = \{\$, \}
  - T' = \{\$, \}

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>int</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S→E $</td>
<td>S→E $</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>E→T E'</td>
<td>E→T E'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>E'→ε</td>
<td>E'→ε</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T→F T'</td>
<td>T→F T'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>T'→ε</td>
<td>T'→ε</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F→(E)</td>
<td>F→(E)</td>
<td></td>
<td></td>
<td>F→int</td>
<td></td>
</tr>
</tbody>
</table>
Non-SLL(1) grammars

- Construction of predictive parse table for grammar results in conflicts

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

\[
\text{FIRST}(S+S) = \text{FIRST}(S*S) = \text{FIRST}(\text{num}) = \{ \text{num} \}
\]
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
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 token 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
    - 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 () → int
    - f.getWord/peekWord:() → String
    - f.getOp/peekOp:() → 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 → boolean`
    • Example: `f.check('*');` //true if next thing in input is *
    • Check if next thing in input is c
      – if so, eat it up and return true
      – otherwise, return false
  – `f.check(String s): String → boolean`
    • Example of its use: `f.check("if");`
    • Check if next word in input matches s
      – if so, eat it up and return true
      – otherwise, return false
  – `f.checkInt(): () → 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 → void`
    • like `f.check` but throws TokenizerException if next token in input is not “c”
  – `f.match(String s): string → void`
    • (eg) `f.match("if")`
Recognizer 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
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 -> integer
            f.checkInt();
            return true;
        case OPERATOR: //E ->(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.
Tracing recursive calls to `getExp`
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
Idea

- 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 + “\n”
  - For (E1 + E2),
    - recursive calls return code for E1 and E2
    - say these are strings S1 and S2
    - method returns S1 + S2 + “ADD\n”
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";
    }
}

code

static String 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`
Top-Down Parsing

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

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

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

  void parse_E()
  Expr parse_E()

  void parse_S()
  Expr parse_S()

  void parse_S’()
  Expr parse_S’()
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 to 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