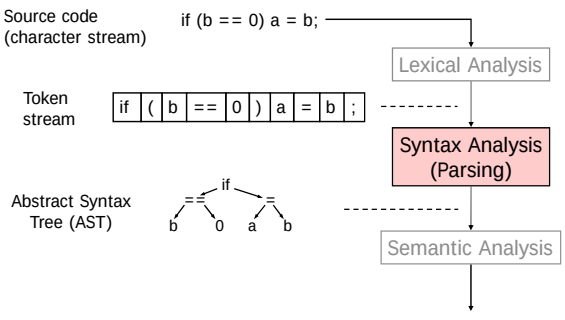
Top Down Parsing

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 → (E + E) | int
  and the string (2+3)
  E ⇒ (E + E) ⇒ (2 + E) ⇒ (2 + 3)
- How could we decide between
  E ⇒ int
  E ⇒ (E+E)
  in the first derivation step?
- Examine the next unread token in the input
  - If it is an integer, use the production E ⇒ int
  - If it is '(', use the production E ⇒ (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 → E + S | E
  E → **num** | (S)
- and the two derivations
  S ⇒ E ⇒ (S) ⇒ (E) ⇒ (3)
  S ⇒ E+S ⇒ (S)+S ⇒ (E)+E ⇒ (3)+E ⇒ (3)+4
- How could we decide between
  S ⇒ E
  S ⇒ 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)

S → E+S
S → E
E → **num**
E → (S)

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

- 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 = ⟨V,Σ,S,→⟩
  - 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 $V \times (\Sigma \cup \{\varepsilon\})$ to the production to apply (or error)

Using Table

E → **int** | (E+E)

E	(	(1+2)
⇒ **(E+E)**	(	(1+2)
⇒ (**E+E**)	1	(1+2)
⇒ (**int+E**)	1	(1+2)
⇒ (int+E)	+	(1+2)
⇒ (int+**E**)	2	(1+2)
⇒ (int+**int**)	2	(1+2)
⇒ (int+int)	)	(1+2)
⇒ (int+int)	EOF	(1+2)

	int	+	(	)	EOF
E	→ **int**		→ (E+E)		

Using Table

S → ES'
S' → ε | +S
E → **num** | (S)

S	(	(1+2+(3+4))+5
⇒ **E**S'	(	(1+2+(3+4))+5
⇒ (**S**)S'	1	(1+2+(3+4))+5
⇒ (**E**S')S'	1	(1+2+(3+4))+5
⇒ (1**S'**)S'	+	(1+2+(3+4))+5
⇒ (1+**S**)S'	2	(1+2+(3+4))+5
⇒ (1+**E**S')S'	2	(1+2+(3+4))+5
⇒ (1+2**S'**)S'	+	(1+2+(3+4))+5

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

How to Implement, Version 1

- A table-driven parser

```
void parse(nonterminal A) {
  int i;
  let A → X0X1...Xn = TABLE[A,token]
  for (i=0; i<=n; i++) {
      if (Xi in Σ)
          if (token == Xi) token = input.read();
          else throw new ParseError();
      else parse(Xi);
  }
  return;
}
```


How to Implement, Version 2

- Convert table into a recursive-descent parser

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

- Three procedures: parse_S, parse_S', parse_E

Recursive-Descent Parser

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

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

Recursive-Descent Parser

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

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

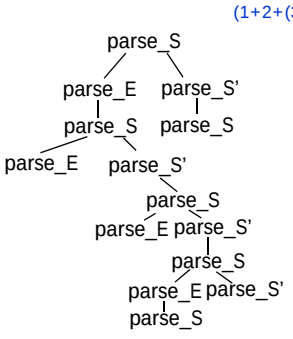
Recursive-Descent Parser

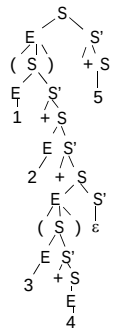
```
void parse_E() {
  switch (token) {
      case number: token = input.read(); return;
      case '(': token = input.read(); parse_S();
              if (token != ')') throw new ParseError();
              token = input.read(); return;
      default: throw new ParseError(); }
}
```

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

Call Tree = Parse Tree

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





How to Construct Parsing Tables

- Needed: algorithm for automatically generating a predictive parsing table from a grammar
- Easy for Bali

S → ES'
S' → ε | +S
E → **num** | (S)

?

	N	+	(	)	ε
S	ES'		ES'		
S'		+S		ε	ε
E	N		(S)		

Filling in parsing table

- First, consider grammars that do not have ε-productions.
- Suppose we have a non-terminal E for which all productions begin with a terminal symbol
  - (eg) E → aXY | bYY | stX
  - Easy to fill in parsing table
    - in Table[E,a] put rule E →aXY etc.
- What if we have a non-terminal in which some rules begin with a non-terminal?
  - (eg) E→ aXY | YZ
  - For what look-ahead symbols should we use the rule E → YZ?
  - Obvious answer: for any terminal symbol that begins a string that can be derived from Y

Formalization

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

$$\frac{(A \rightarrow Y_1..Yn) \in P \quad t \in FIRST(Y_1)}{t \in FIRST(A)}$$

- Example
  - S → AB|BA A→tB|a B→xA
  - FIRST(A) = {t,a} FIRST(B) = {x} 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
- FIRST(α) $\subseteq$ T U {ε}
  - if α can be rewritten to a string starting with terminal t, then t is in FIRST(α)
  - if α can be rewritten to ε, then ε is in FIRST(α)
- FOLLOW(A) $\subseteq$ T
  - set of terminals that can follow A in a sentential form

Constructing parsing table

- Enter production A → $Y_1 \ldots Y_n$ into Table[A,t] for all t such that
  - $t \in FIRST(Y_1 \ldots Yn)$
  - $\varepsilon \in FIRST(Y_1..Yn)$ and $t \in FOLLOW(A)$

NULLABLE and FIRST

G = <N,T,P,S> (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,

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

NULLABLE $\subseteq$ N

$$\frac{(A \rightarrow \varepsilon) \in P}{A \in NULLABLE} \qquad \frac{(A \rightarrow Y_1..Yn) \in P \quad Y_1,..Yn \in NULLABLE}{A \in NULLABLE}$$

FIRST(A) $\subseteq$ T U {ε}

$$\frac{A \in NULLABLE}{\varepsilon \in FIRST(A)} \qquad \frac{(A \rightarrow Y_1..Yn) \in P \quad Y_1,\ldots,Yk \in NULLABLE \quad t \in FIRST(Yk_{+1})}{t \in FIRST(A)}$$

Extend FIRST to strings

Helper function: $+_1$
binary operation on subsets of T U {e}
concatenate and truncate to 1 symbol

{ε,a,b} $+_1$ {ε,a} = {ε,a,b}

FIRST(ε) = {ε}
FIRST(t) = {t}
FIRST($Y_1 \ldots Y_n$) = FIRST(Y_1) $+_1$ FIRST(Y_2) $+_1$ …FIRST(Y_n)

FOLLOW

FOLLOW(A) $\subseteq$ T U {$}

$$\frac{}{\$ \in FOLLOW(S)}$$

$$\frac{(A \rightarrow \alpha BY_1..Yn) \in P \quad t \in FIRST(Y1..Yn)}{t \in FOLLOW(B)}$$

$$\frac{(A \rightarrow \alpha BY_1..Yn) \in P \quad \varepsilon \in FIRST(Y_1 \ldots Yn) \quad t \in FOLLOW(A)}{t \in FOLLOW(B)}$$

Non-trivial example

- Grammar for arithmetic expressions
  S →E $
  E → E+T | T
  T → T*F | F
  F → (E) | int
- Grammar is not LL(1)
- Massaged grammar
  S → E $
  E → T E'
  E' → + T E' | ε
  T → F T'
  T' → * F T' | ε
  F → (E) | int
- Nullable = {E',T'}
- FIRST
  - E = {(,int}
- FOLLOW
  - E' ={$,)}
  - T' = {$,)}

	+	*	(	)	int	$
S			S→E $		S→E $	
E			E→T E'		E→T E'	
E'	+ T E'			E'→ ε		E'→ ε
T			T→F T'		T→F T'	
T'	T'→ε	T'→*FT'		T'→ε		T'→ε
F			F→(E)		F→int	

Non-SLL(1) grammars

- Construction of predictive parse table for grammar results in conflicts

S → S+S | S*S | **num**

FIRST(S+S) = FIRST(S*S) = FIRST(**num**) = { **num** }

	num	+	*	ε
S	→**num**, →S+S, →S*S			

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

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
    - 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")

Parser for simple expressions

Expression → integer
Expression → (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

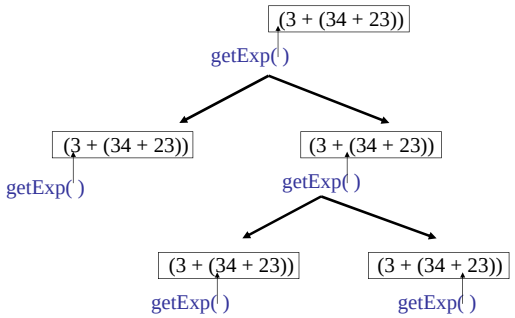
Parser for expression language

```
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 -> 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
```

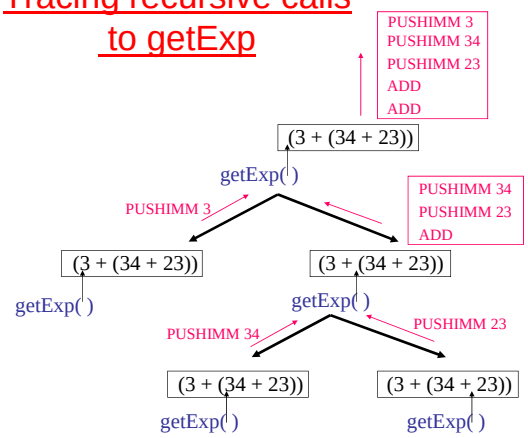
Idea

- 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"

CodeGen for expression language

```
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";
              }
}

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

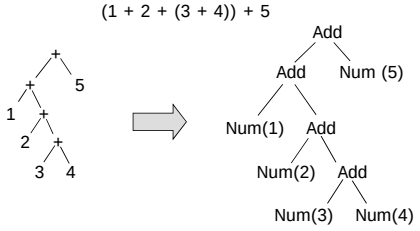
Creating the AST

```
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; }
}
```

Class Hierarchy

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:

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


AST Creation: parse_E

```
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