Introduction to Grammars and Parsing

Grammars

Sentence → Noun Verb Noun
Noun → boys
Noun → girls
Noun → dogs
Verb → like
Verb → see

- Grammar: set of rules for generating sentences in a language.
- Our sample grammar has these rules:
  - a Sentence can be a Noun followed by a Verb followed by a Noun
  - a Noun can be 'boys' or 'girls' or 'dogs'
  - a Verb can be 'like' or 'see'
- Examples of Sentence:
  - boys see dogs
  - dogs like girls
  - …..
- Note: white space between words does not matter
- This is a very boring grammar because the set of Sentences is finite (exactly 18 sentences). Work this out as an exercise.

Terminology

- Symbols: names/strings in grammar
  - (eg) Sentence, Noun, Verb, boys, girls, dogs, like, see
- Non-terminals: symbols that occur on the left hand sides of rules
  - (eg) Sentence, Noun, Verb
- Terminals: symbols that do not occur on left hand sides of rules
  - (eg) boys, girls, dogs, like, see
- Start symbol: the symbol used to begin the derivation of sentences
  - (eg) Sentence

Sentence → Noun Verb Noun
Noun → boys
Noun → girls
Noun → dogs
Verb → like
Verb → see

Generating sentences

- Begin with a string consisting of the start symbol
- Replace any *non-terminal* X in the string by a right-hand side of some production
- $X \rightarrow Y_1 \ldots Y_n$
- Repeat (2) until there are only terminals in the string
- The successive strings created in this way are called *sentential forms.*
- This process is called a derivation.
- Example:

  Sentence → Noun Verb Noun → boys Verb Noun → boys like Noun → boys like girls

Recursive grammar

Sentence → Sentence and Sentence
Sentence → Sentence or Sentence
Sentence →Noun Verb Noun
Noun → boys
Noun → girls
Noun → dogs
Verb → like
Verb → see

- Examples of Sentences in this language:
  - boys like girls
  - boys like girls and girls like dogs
  - boys like girls and girls like dogs and girls like dogs
  - boys like girls and girls like dogs and girls like dogs and girls like dogs
  - ……
- This grammar is more interesting than the one in the last slide because the set of Sentences is infinite.
- What makes this set infinite? Answer: recursive definition of Sentence

Detour

- What if we want to add a period at the end of every sentence?
- Does this work?

Sentence → Sentence and Sentence .
Sentence → Sentence or Sentence .
Sentence →Noun Verb Noun .

Noun → ……..

No! This produces sentences like
girls like boys . and boys like dogs . .

Sentences with periods

TopLevelSentence → Sentence .
Sentence → Sentence and Sentence
Sentence → Sentence or Sentence
Sentence →Noun Verb Noun
Noun → boys
Noun → girls
Noun → dogs
Verb → like
Verb → see

- Add a new rule that adds a period only at the end of the sentence.
- Thought exercise: how does this work?
- End of detour

Grammar for simple expressions

Expression → integer
Expression → (Expression + Expression)

- This is a grammar for simple expressions:
  - An E can be an integer.
  - An E can be '(' followed by an E followed by '+' followed by an E followed by ')'
- Set of Expressions defined by this grammar is a recursively-defined set.

E → integer

E → (E + E)

Here are some legal expressions:

2

(3 + 34)

((4+23) + 89)

((89 + 23) + (23 + (34+12)))

Here are some illegal expressions:

(3

3 + 4

Parsing

- Parsing: given a grammar and some text, determine if that text is a legal sentence in the language defined by that grammar
- For many grammars such the simple expression grammar, we can write efficient programs to answer this question.
- Next slides: parser for our small expression language

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

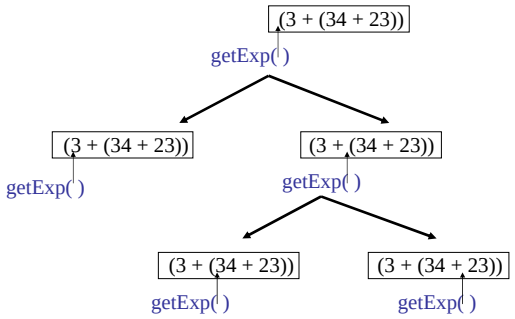
```
static boolean expParser(String fileName) {//parser for expression in file
      try {
            SamTokenizer f = new SamTokenizer (fileName);
            return getExp(f) && (f.peekAtKind() == Tokenizer.TokenType.EOF) ;//must be at EOF
           } catch (Exception e) {
                System.out.println("Aaargh");
                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



Parsing tables

- We produced recursive parser informally by studying the grammar
- More formal approach: parsing table
  - one row for each non-terminal
  - one column for each terminal
  - entry T[NT,t]:
    - production to apply in method for NT if next input token ("look-ahead") is t
- Algorithm for producing parsing table from grammar
  - study later

E → int
E → (E+E)

	(	int	+	)
E	E→(E+E)	E→int		

Template-driven recursive code for parser

- Easy to produce mechanically from the parsing table
- One method for each non-terminal
- Body of method for non-terminal "NT":
  - switch statement should have one case for each terminal symbol in the grammar
  - body of switch statement for look-ahead token "t":
    - production in T[NT,t] tells you what action to take
    - cases:
      - empty production: return true
      - otherwise: conjunction of calls to "check" (for terminals) and appropriate methods (for non-terminals)

Template

```
static boolean getNon-terminal(SamTokenizer f) {
        switch (f.peekAtKind()) {
          case INTEGER: ……..
          case OPERATOR:
                    {switch (f.peekOp()) {
                        case '(': {.. }
                        case '+':{...}
                        ……
                        default: return false;
                    }
          case WORD: ……..

          ……..
          default:   return false;
        }
}
```

Template-driven recursive code for parser

E → int
E → (E+E)

	(	int	+	)
E	E→(E+E)	E→int		

```
static boolean getE(SamTokenizer f) {
        switch (f.peekAtKind()) {
          case INTEGER: return f.checkInt();
          case OPERATOR:
                    {switch (f.peekOp()) {
                        case '(': return f.check('(' )&&
                                         getE(f) &&
                                         f.check('+') &&
                                         getE(f) &&
                                         f.check(')')
                                         }
                        default: return false;
                    }
          default:  return false;
        }
}
```

Caveat

- You may not be able to come up with a parsing table for some grammars
  - some entry in the table will have multiple productions in it
- Example:
  E→ int
  E→ (E + E)
  E→ (E – E)
- What should be the entry for Table[E,(]?
  - RHS of second and third productions both begin with (
- We can build unambiguous parsing tables only for LL(1) grammars
  - define this later
- If grammar is not LL(1), we can often massage it to become LL(1)
  E→ int
  E→ (E Op E)
  Op→ +
  Op→ -

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

	+	*	(	)	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	

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

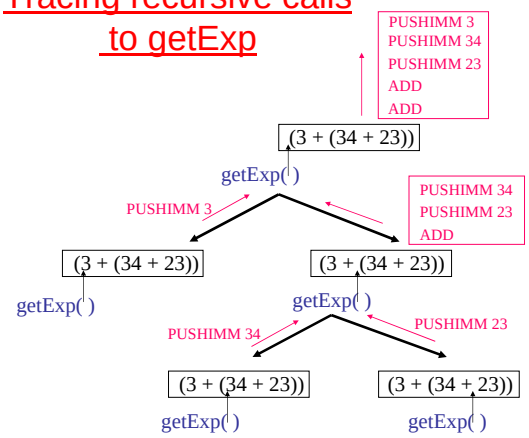


Table-driven code generation

- Earlier we saw how to produce a parser mechanically from an LL(1) grammar
  - compute parsing table from grammar
  - use template to produce parser from parsing table
- Advantage:
  - need to write and maintain just one parser-generator
  - easy to maintain parser for a given language
    - you just have to maintain the grammar, which is a nice, abstract specification
- How do we extend this idea from parsing to code generation?
  - see later: formal concept is called "attribute grammars"

Syntax-Directed Definition

- Solution: syntax-directed definition
  - Extends each grammar production with an associated semantic action (code):

$$E \rightarrow (E+E) \quad \{ \text{ action } \}$$

  - The parser generator adds these actions into the generated parser
  - Each action is executed when the corresponding production is executed
- Formal concept: attribute grammars
- Study this later

Conclusion

- The parsers we have written are called "recursive descent parsers"
  - parser is essentially a set of mutually recursive functions that can be written down directly from the grammar
- Not all grammars can be parsed by a recursive descent parser
  - most grammars require more complex parsers
  - most grammars can be massaged so that they become LL(1) but grammar becomes clumsy
- There is a systematic way of generating parsing tables for recursive-descent parsers
  - see later when we study formal methods
- 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

Grammar concepts

Overview

- To understand different parsing strategies, it is useful to introduce some formalism
  - derivations
  - leftmost and rightmost derivations
- Let us also look at ambiguous grammars

Derivations and Parse Trees

- A *derivation* is a sequence of sentential forms resulting from the application of a sequence of productions to start symbol

  $S \rightarrow \ldots \rightarrow \ldots$

- *Parse tree: summary of derivation w/o specifying completely the order in which rules were applied*
  - Start symbol is the tree's root
  - For a production $X \rightarrow Y_1 \ldots Y_n$ add children $Y_1, \ldots, Y_n$ to node X

Derivation Example

- Grammar

  $E \rightarrow E + E \mid E * E \mid (E) \mid int$

- String

  int * int + int

Derivation in Detail (1)

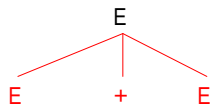
E

E

Derivation in Detail (2)

E

→ E + E

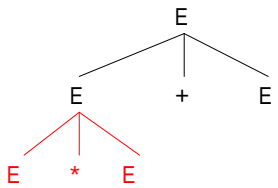


Derivation in Detail (3)

E

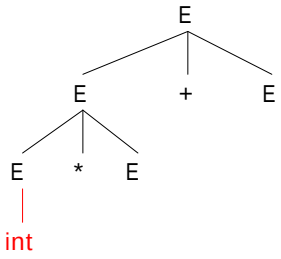
→ E + E

→ E * E + E



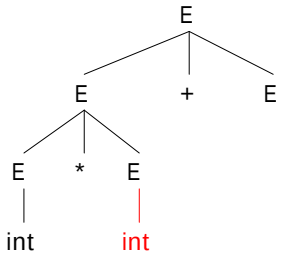
Derivation in Detail (4)

E
→ E + E
→ E * E + E
→ int * E + E



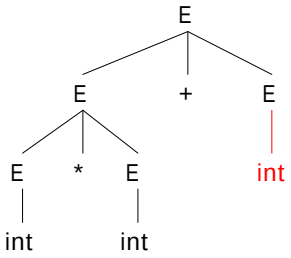
Derivation in Detail (5)

E
→ E + E
→ E * E + E
→ int * E + E
→ int * int + E



Derivation in Detail (6)

E
→ E + E
→ E * E + E
→ int * E + E
→ int * int + E
→ int * int + int



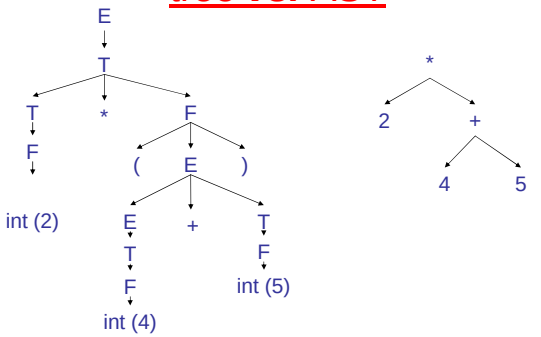
Notes on Derivations

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- A left-right traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not !
  - There may be multiple ways to match the input
  - Derivations (and parse trees) choose one

AST vs. Parse Tree

- AST is condensed form of a parse tree
  - operators appear at *internal* nodes, not at leaves.
  - "Chains" of single productions are collapsed.
  - Lists are "flattened".
  - Syntactic details are omitted
    - e.g., parentheses, commas, semi-colons
- AST is a better structure for later compiler stages
  - omits details having to do with the source language,
  - only contains information about the *essential* structure of the program.

Example: 2 * (4 + 5) Parse tree ***vs.*** AST



Leftmost and Rightmost Derivations

E	E
→ E + E	→ E + E
→ E * E + E	→ E + int
→ int * E + E	→ E * E + int
→ int * int + E	→ E * int + int
→ int * int + int	→ int * int + int
Leftmost derivation: always act on leftmost non-terminal	*Rightmost derivation:* always act on rightmost non-terminal

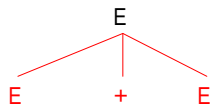
rightmost Derivation in Detail (1)

E

E

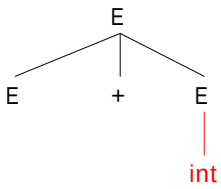
rightmost Derivation in Detail (2)

E
→ E + E



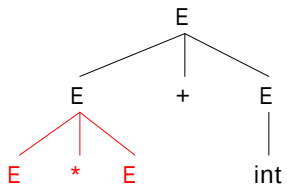
rightmost Derivation in Detail (3)

E
→ E + E
→ E + int



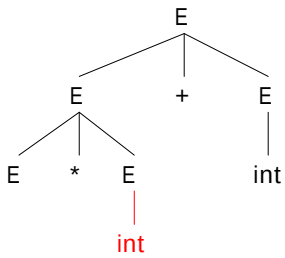
rightmost Derivation in Detail (4)

E
→ E + E
→ E + int
→ E * E + int



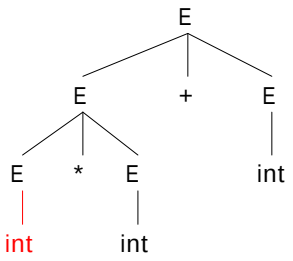
rightmost Derivation in Detail (5)

E
→ E + E
→ E + int
→ E * E + int
→ E * int + int



rightmost Derivation in Detail (6)

E
→ E + E
→ E + int
→ E * E + int
→ E * int + int
→ int * int + int



Aside: Canonical Derivations

- Take a look at that last derivation *in reverse.*
- The active part (red) tends to move left to right.
- We call this a *reverse rightmost* or *canonical* derivation.
- Comes up in *bottom-up parsing.* We'll return to it in a couple of lectures.

Ambiguity

- Grammar

  $E \rightarrow E + E \mid E * E \mid (E) \mid int$

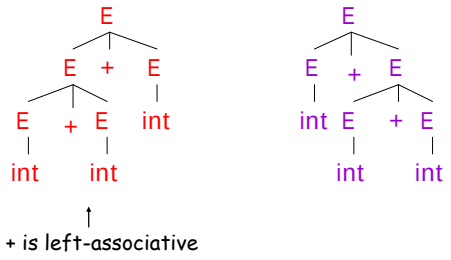
- Strings

  int + int + int

  int * int + int

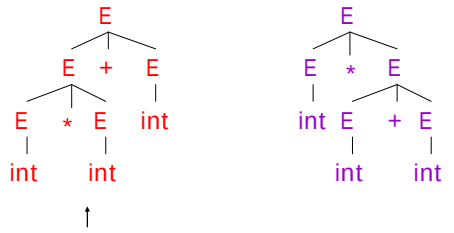
Ambiguity. Example

The string int + int + int has two parse trees



Ambiguity. Example

The string int * int + int has two parse trees



Which code should generate??

Ambiguity (Cont.)

- A grammar is *ambiguous* if it has more than one parse tree for some string
  - Equivalently, there is more than one rightmost or leftmost derivation for some string
- Ambiguity is *bad*
  - Leaves meaning of some programs ill-defined
- Ambiguity is *common* in programming languages
  - Arithmetic expressions
  - IF-THEN-ELSE

Dealing with Ambiguity

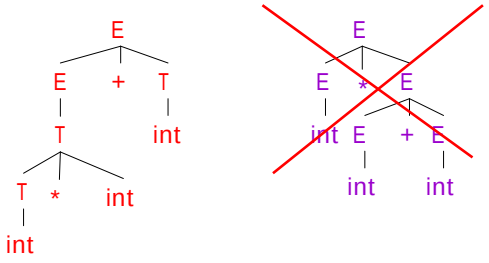
- There are several ways to handle ambiguity
- Most direct method is to rewrite the grammar unambiguously

  E → E + T | T

  T → T * int | int | (E)
- Enforces precedence of * over +
- Enforces left-associativity of + and *

Ambiguity. Example

The int * int + int has only one parse tree now



Ambiguity: The Dangling Else

- Consider the grammar

  E → if E then E
  | if E then E else E
  | OTHER

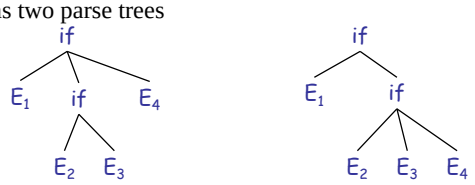
- This grammar is also ambiguous

The Dangling Else: Example

- The expression

  if E_1 then if E_2 then E_3 else E_4

  has two parse trees



- Typically we want the second form

The Dangling Else: A Fix

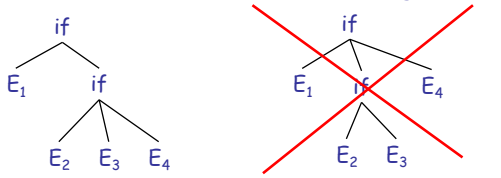
- else matches the closest unmatched then
- We can describe this in the grammar (distinguish between matched and unmatched "then")

```
E →  MIF          /* all then are matched */
   | UIF          /* some then are unmatched */
MIF → if E then MIF else MIF
    | OTHER
UIF → if E then E
    | if E then MIF else UIF
```

- Describes the same set of strings

The Dangling Else: Example Revisited

- The expression if E_1 then if E_2 then E_3 else E_4



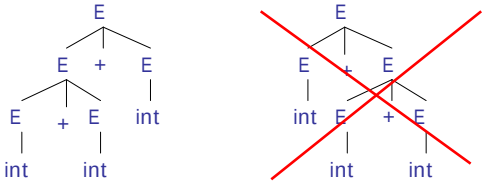
- A valid parse tree (for a UIF)
- Not valid because the then expression is not a MIF

Ambiguity

- Impossible to convert automatically an ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - But we need disambiguation mechanisms
- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations
- Most tools allow *precedence and associativity declarations* to disambiguate grammars
- Examples …

Associativity Declarations

- Consider the grammar $E \rightarrow E + E \mid int$
- Ambiguous: two parse trees of int + int + int



- Left-associativity declaration: %left '+'