Grammars and Top-Down Parsing

CS 132: Compiler Design

January 24, 2001

Preliminary Definitions

- An *alphabet* is a set of symbols
  - Characters, tokens, etc. as appropriate
- A *sentence* is a finite sequence of symbols taken from the alphabet
- A *language* is a set of sentences
  - Question: given a language and a sentence, is the sentence in the language?
- A *grammar* is a (finite) description of a language.

Phase Structure Grammars

- A *phase structure grammar* is specified by four components
  - A set of symbols called *terminals*
    - A string of terminals is called a *sentence*
  - A set of symbols called *nonterminals*
    - A string of terminals and/or nonterminals is called a *sentential form.*
  - A set of rewrite rules (pairs of sentential forms)
  - A distinguished nonterminal called the *start symbol.*

Generating Sentences

- Given such a grammar, a production step is the result of applying one of the rewriting rules to a substring of a sentential form

- The language generated by a grammar is the set of sentences which can be reached in a finite number of production steps from the start symbol

Example PS Grammar

Terminals: {`tom`, `dick`, `harry`, `,` , and}
Nonterminals: {`Name`,`Sentence`,`List`,`End`}
Start Symbol: `Sentence`
Rewrite Rules:

```
Name         → tom
Name         → dick
Name         → harry
Sentence     → Name
Sentence     → List End
List         → Name
List         → List , Name
, Name End   → and Name
```

Abbreviated Representation

Terminals: {`tom`, `dick`, `harry`, `,` , and}
Nonterminals: {`Name`,`Sentence`,`List`,`End`}
Start Symbol: `Sentence`
Rewrite Rules:

```
Name         → tom | dick | harry
Sentence     → Name | List End
List         → Name | List , Name
, Name End   → and Name
```

Example Derivation

```
Sentence
   → List End
   → Name , List End
   → Name , Name , List End
   → Name , Name , Name End
   → Name , Name and Name
   → Name , Name and harry
   → Name , dick and harry
   → tom , dick and harry
```

Chomsky Hierarchy

- Type 0
  - A language is said to be Type 0 if it can be generated by some phase structure grammar.

Chomsky Hierarchy

- Type 1
  - A language is said to be Type 1 if it can be generated by a *monotonic* grammar:
    - Left-hand side of a rule cannot be longer than the right-hand side

```
Name       → tom | dick | harry
Sentence   → Name | List
List       → EndName | Name , List
, EndName  → and Name
```

Chomsky Hierarchy

- Type 1 (Context Sensitive)
  - Equivalent definition: those languages produced by *context-sensitive grammars*:
    - Rewriting rules can only change a single nonterminal.

```
Name           → tom     | dick | harry
Sentence       → Name    | List
List           → EndName | Name Comma List
Comma EndName  → and EndName
and EndName    → and Name
Comma          → ,
```

Chomsky Hierarchy

- Every Type 1 language is a Type 0 language, but not vice-versa:

  "...there are languages that can be generated by a Type 0 grammar but not by any Type 1. Strangely enough no simple examples of such languages are known. Although the difference between Type 0 and Type 1 is fundamental and is not just a whim of Mr. Chomsky, grammars for which the difference matters are too complicated to write down; only their existence can be proved."

Chomsky Hierarchy

- Type 2 (Context-Free)
  - A language is said to be Type 2 if it can be generated by a *context-free grammar*:
    - Left-hand side of a rule must be a single nonterminal

```
Name       → tom | dick | harry
Sentence   → Name | List and Name
List       → Name , List | Name
```

Chomsky Hierarchy

- Type 3 (Regular)
  - A language is said to be Type 3 if it can be generated by a *regular grammar*:
    - Left-hand side of a rule must be a single nonterminal
    - Right-hand side is a sentence or a terminals followed by a single nonterminal

```
Sentence → tom | dick | harry | List
List     → tom LTail | dick LTail
          | harry LTail
LTail    → , List
          | and tom | and dick | and harry
```

Example Languages

- Type 0
  - Set of all *terminating* Java programs
- Type 1 (Context-sensitive)
  - Set of all *valid* Java programs
- Type 2 (Context-free)
  - Set of all *syntactically correct* Java programs
- Type 3 (Regular)
  - Set of all *lexically correct* Java programs

Languages and Grammars

- Need to be careful to distinguish
  - When a language falls in a certain class
  - When a grammar for that language falls in a certain class.
  - e.g., just because you have a context-sensitive grammar for a language doesn't mean that the language isn't regular
- When automating parsing, generally it's the type of the grammar that determines success

Context-Free Grammars

- Usual method for describing language syntax
  - Began with BNF (Backus-Naur Form) for Algol 60.
  - Only major exception was Algol 68
    - Context-sensitive grammar enforced declaration, consistent use of variables
    - Defined via a W-grammar (Van Wijngaarden grammar)
      - Two-level grammar

Definitions

- A sequence of production steps is called a *leftmost derivation* if at each step the leftmost nonterminal is rewritten.
- Similarly a sequence is a *rightmost derivation* if at each step the rightmost nonterminal is rewritten.

Example

```
S → E
E → 1 | 2 | E - E
```

- Leftmost derivation:

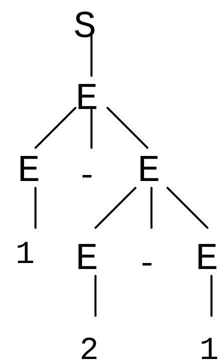
```
S → E → E - E → 1 - E → 1 - E - E
  → 1 - 2 - E → 1 - 2 - 1
```

- Rightmost derivation:

```
S → E → E - E → E - E - E → E - E - 1
  → E - 2 - 1 → 1 - 2 - 1
```

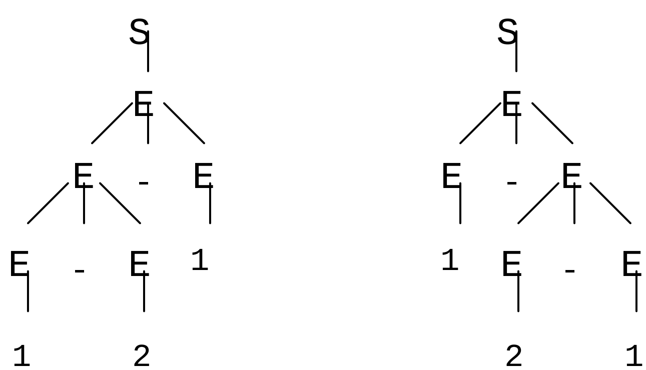
Parse Trees

- Diagram showing what rules were applied where in deriving a sentence



Ambiguous Grammars

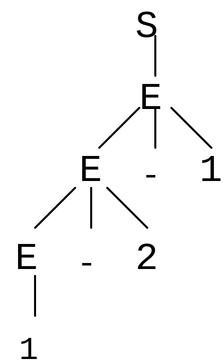
- A grammar is said to be *ambiguous* if one string permits distinct parse trees.



Ambiguous Grammars

- Ambiguity can usually be removed by rewriting the grammar (without changing the language being defined).

```
S → E
E → 1 | 2
    | E - 1 | E - 2
```



Parsers

- A parser is a program that takes a string (sequence of tokens) and determines whether this string is in the grammar
  - And if so, *how*?
- A *top-down* parser tries to build a parse tree from the root and working downward
- A *bottom-up* parser attempts to build the parse tree from the roots up (by running the production rules backwards if adjacent tokens match the right-hand side of a rewrite rule.)

Parsing Context-Free Languages

- Algorithms exist for parsing arbitrary context-free languages.
  - But these take $O(n^3)$ time, where n is the length of the input string.
  - Theoretically possible to do better, e.g., $O(n^{2.81})$
  - No known CF grammar can't be parsed in linear time with an ad-hoc parser.
- By considering restricted CF grammars, can perform unambiguous, linear-time parsing.

Predictive Parsing

- A top-down parser is sometimes called a *predictive* parser
- At each step, choose a non-terminal and "predict" how it will be expanded
  - Try to use information about the input to guide prediction
  - In worst case, end up trying all possibilities (breadth-first search)
  - We will consider grammars where there's always a unique prediction

Predictive Parsing Example

Prediction Stack	Input Stream	Action
S	a a b b	Predict $S \rightarrow aB$
a B	a a b b	Match
B	a b b	Predict $B \rightarrow aBb$
a B b	a b b	Match
B b	b b	Predict $B \rightarrow b$
b b	b b	Match
b	b	Match
success		

$S \rightarrow a\ B$
$B \rightarrow b\ |\ a\ B\ b$

Recursive Descent

- Define a function for each nonterminal
- Tries to find a prefix of the input stream matching that nonterminal

- Works by choosing a production for the nonterminal and recursively matching the right-hand-side against the input stream.

Recursive Descent Example

```
S → if E then S else S
  | begin S L
  | print E
L → end
  | ; S L
E → num = num
```

```
datatype token = IF | THEN | ELSE | BEGIN | END
               | PRINT | SEMI | NUM | EQ
val tok = ref (getToken())
fun advance() = (tok := getToken())
fun eat(t) = if (!tok = t) then advance() else error()
```

Recursive Descent Example

```
fun S() = case !tok of
            IF => (eat(IF); E(); eat(THEN); S();
                   eat(ELSE); S())
          | BEGIN => (eat(BEGIN); S(); L())
          | PRINT => (eat(PRINT); E())
and L() = case !tok of
            END => (eat(END))
          | SEMI => (eat(SEMI); S(); L())
          | PRINT => (eat(PRINT); E())
and E() = (eat(NUM); eat(EQ); eat(NUM))
```

When Does This Work?

- Consider the grammar
  `E → 1 | 2 | E - 1 | E - 2`
- The corresponding code would look like:

```
fun E() = case !tok of
            ??? => eat(ONE)
          | ??? => eat(TWO)
          | ??? => (E(); eat(PLUS); eat(ONE))
          | ??? => (E(); eat(PLUS); eat(TWO))
```

- What to do on input `1` or `1-1`?

When Does This Work?

- Recursive descent works only on grammars where the *first symbol(s)* of each subexpression provides enough information to know how with what rule it can be produced.

Left Recursion

- A grammar is said to be left-recursive if there is a production sequence of the form

$$X \rightarrow^{+} X...$$

- Left-recursive rules break recursive descent.

```
E → E - N
E → N
N → 0 | 1
```

Eliminating Left Recursion

- Left recursion can be replaced by right recursion in a mechanical way.

```
E → N E'
E' → - N E' | ε
N → 0 | 1
```

- The new grammar describes the same language, but the parse trees differ.

Left Factoring

- Another problem arises if two productions for the same nonterminal begin with the same nonterminal:

```
S → if E then S else S
S → if E then S
```

Left Factoring

- We can remove such conflicts by left factoring
  - Separating out the common part.

```
S → if E then S X
X → else S | ε
```

First, Follow, and Nullable

- We use α and β to denote sentential forms
- FIRST(α) is the set of terminals `t` such that

$$\alpha \rightarrow^{*} \mathtt{t}\beta$$

  That is, FIRST(α) is the set of terminals that can begin a string derived from α.
- FOLLOW(X) is the set of terminals t such that

$$\mathtt{S} \rightarrow^{*} \alpha\mathtt{Xt}\beta$$

  That is, the set of terminals that can immediately follow X in some derivation
- Finally, we say that α is *nullable* if $\alpha \rightarrow^{*} \varepsilon$.

Nullable Nonterminals

- To compute whether a nonterminal is nullable:
  - Initially assume not.
  - Run through all the productions to see if any nonterminals are seen to be nullable.
  - Iterate until the answer doesn't change

Nullable Example

$$
\begin{aligned}
Z &\to d \mid X\ Y\ Z \\
Y &\to \varepsilon \mid c \\
X &\to Y \mid a
\end{aligned}
$$

0th iteration: Z no, Y no, X no.

1st iteration: Z no, Y yes, X no.

2nd iteration: Z no, Y yes, X yes.

3rd iteration: Z no, Y yes, X yes.

Computing First Sets

- To compute First sets for all nonterminals:
  - Assume all these sets are empty
  - Run through all the productions and see what nonterminals must be added to the first sets
    - e.g., if $Z \to d \mid X\ Y$

      then First(Z) must contain d, must include First(X), and must include First(Y) if X is nullable.
  - Iterate to convergence.

First Example

$Z \rightarrow d \mid X\ Y\ Z$
$Y \rightarrow \varepsilon \mid c$
$X \rightarrow Y \mid a$

0th iteration: F(Z) = {}, F(Y)={}, F(X)={}

1st: F(Z) = {d}, F(Y)={c}, F(X)={a}

2nd: F(Z) = {d,a,c}, F(Y)={c}, F(X)={a,c}

3rd: F(Z) = {d,a,c}, F(Y)={c}, F(X)={a,c}

Computing Follow Sets

- To compute First sets for all nonterminals:
  - Assume all these sets are empty
  - Run through all the productions and see what nonterminals must be added to the first sets
    - e.g., if $Z \rightarrow X\ Y\ Z$

      then Follow(X) must include First(Y), must include First(Z) if Y is nullable, and must include Follow(Z) if Y and Z are nullable.
  - Iterate to convergence.

Follow Example

$$
\begin{aligned}
Z &\to d \mid X\ Y\ Z \\
Y &\to \varepsilon \mid c \\
X &\to Y \mid a
\end{aligned}
$$

0th iteration: F(Z) = {}, F(Y)={}, F(X)={}

1st: F(Z) = {}, F(Y)={d}, F(X)={c,d}

2nd: F(Z) = {}, F(Y)={d,a,c}, F(X)={c,d,a}

3rd: F(Z) = {}, F(Y)={d,a,c}, F(X)={c,d,a}

Building a Predictive Parser

- Given $A \to \alpha_1 | \ldots | \alpha_n$, define the function:

```
fun A() =
   if (!tok ∈ FIRST(α1)) or
      (nullable(α1) and (!tok ∈ FOLLOW(A)) then
       ...match α1 against input stream...
   else if (!tok ∈ FIRST(α2)) or
      (nullable(α2) and (!tok ∈ FOLLOW(A)) then
       ...match α2 against input stream...
   ...
   else if (!tok ∈ FIRST(αn)) or
      (nullable(αn) and (!tok ∈ FOLLOW(A)) then
       ...match αν against input stream...
   else ERROR
```

LL(1)

- A grammar is said to be LL(1) if this procedure works
  - First L: reads input from "left to right"
  - Second L: generates a leftmost derivation
  - 1: makes decisions based on 1 symbol lookahead
- In particular, all the cases in any test must be mutually disjoint for all inputs.
- Yields a deterministic parser that runs in linear time (no backtracking).
- No LL(1) grammar can be ambiguous.

Alternate Implementation

- Instead of recursive functions, build a table
  - Indexed by current nonterminal and current lookahead symbol
  - Table entry says which rule to use to expand the nonterminal.
- An LL(1) grammar is one where there are every table entry gives at most one rule.
- An language is said to be LL(1) if there exists some LL(1) grammar that produces it