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

<table>
<thead>
<tr>
<th>Name</th>
<th>→ tom</th>
<th>dick</th>
<th>harry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence</td>
<td>→ Name</td>
<td>List and Name</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td>→ Name , List</td>
<td>Name</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Sentence</th>
<th>→</th>
<th>tom</th>
<th>dick</th>
<th>harry</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>→</td>
<td>tom LTail</td>
<td>dick LTail</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>harry LTail</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LTail</td>
<td>→</td>
<td>, List</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>and tom</td>
<td>and dick</td>
<td>and harry</td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow 1 \mid 2 \mid E - E
\end{align*}
\]

• Leftmost derivation:
  \[
  S \rightarrow E \rightarrow E - E \rightarrow 1 - E \rightarrow 1 - E - E \\
  \quad \rightarrow 1 - 2 - E \rightarrow 1 - 2 - 1
  \]

• Rightmost derivation:
  \[
  S \rightarrow E \rightarrow E - E \rightarrow E - E - E \rightarrow E - E - 1 \\
  \quad \rightarrow E - 2 - 1 \rightarrow 1 - 2 - 1
  \]
Parse Trees

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

```
S
   E
      E
         E
            E
                1
                E
                    E
                        1
                        E
                            2
                            1
```

Ambiguous Grammars

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

```
S
   E
      E
         E
            E
                1
                E
                    E
                        1
                        E
                            2
                            1
```
Ambiguous Grammars

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

$S \rightarrow E$
$E \rightarrow 1 \mid 2$
$\mid E - 1 \mid E - 2$

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>E - 1</td>
<td>E - 2</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Prediction Stack</th>
<th>Input Stream</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>a a b b</td>
<td>Predict S→aB</td>
</tr>
<tr>
<td>a B</td>
<td>a a b b</td>
<td>Match</td>
</tr>
<tr>
<td>B</td>
<td>a b b</td>
<td>Predict B→aBb</td>
</tr>
<tr>
<td>a B b</td>
<td>b b</td>
<td>Match</td>
</tr>
<tr>
<td>B b</td>
<td>b b</td>
<td>Match</td>
</tr>
<tr>
<td>b b</td>
<td>b</td>
<td>Match</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>success</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S$$
$$| \; \text{begin } S \; L$$
$$| \; \text{print } E$$
$$L \rightarrow \text{end}$$
$$| \; ; \; S \; L$$
$$E \rightarrow \text{num} = \text{num}$$

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
  \[ E \rightarrow 1 \mid 2 \mid E - 1 \mid E - 2 \]
• The corresponding code would look like:

```haskell
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
  \[X \rightarrow^+ X \ldots\]

- Left-recursive rules break recursive descent.

```
E \rightarrow E - N
E \rightarrow N
N \rightarrow 0 | 1
```

Eliminating Left Recursion

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

```
E \rightarrow N E'
E' \rightarrow - N E' | \epsilon
N \rightarrow 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:

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S \\
S \rightarrow \text{if } E \text{ then } S
\]

Left Factoring

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

\[
S \rightarrow \text{if } E \text{ then } S \ X \\
X \rightarrow \text{else } S \mid \epsilon
\]
First, Follow, and Nullable

- We use $\alpha$ and $\beta$ to denote sentential forms
- $\text{FIRST}(\alpha)$ is the set of terminals $t$ such that $\alpha \rightarrow^* t \beta$
  That is, $\text{FIRST}(\alpha)$ is the set of terminals that can begin a string derived from $\alpha$.
- $\text{FOLLOW}(X)$ is the set of terminals $t$ such that $S \rightarrow^* \alpha X t \beta$
  That is, the set of terminals that can immediately follow $X$ in some derivation
- Finally, we say that $\alpha$ is nullable if $\alpha \rightarrow^* \epsilon$.

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

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

<table>
<thead>
<tr>
<th>Z</th>
<th>d</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>ε</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
<td>a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 \( \text{Follow}(X) \) must include \( \text{First}(Y) \), must include \( \text{First}(Z) \) if \( Y \) is nullable, and must include \( \text{Follow}(Z) \) if \( Y \) and \( Z \) are nullable.
  - Iterate to convergence.
Follow Example

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 $A \rightarrow \alpha_1 | \ldots | \alpha_n$, define the function:

```plaintext
fun A() = 
  if (!tok $\in$ FIRST($\alpha_1$)) or
    (nullable($\alpha_1$) and (!tok $\in$ FOLLOW(A)) then
      ...match $\alpha_1$ against input stream...
    else if (!tok $\in$ FIRST($\alpha_2$)) or
      (nullable($\alpha_2$) and (!tok $\in$ FOLLOW(A)) then
      ...match $\alpha_2$ against input stream...
    ... 
    else if (!tok $\in$ FIRST($\alpha_n$)) or
      (nullable($\alpha_n$) and (!tok $\in$ FOLLOW(A)) then
      ...match $\alpha_n$ 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