CS 406: Context-Free Grammars and Top-Down Parsing

Stefan D. Bruda

Winter 2016

Context-Free Grammars

- A context-free grammar is a tuple $G = (N, \Sigma, R, S)$, where
  - Σ is an alphabet of terminals including the end-of-input token \$
  - N alphabet of symbols called by contrast nonterminals (or variables)
    - $N \cap \Sigma = \emptyset$
    - Traditionally nonterminals are capitalized or surrounded by $\langle$ and $\rangle$, everything else being a terminal
  - $S \in N$ is the axiom (or the start symbol)
  - $R \subseteq N \times (N|\Sigma)^*$ is the set of (rewriting) rules or productions
    - Common ways of expressing $(\alpha, \beta) \in R$: $\alpha \rightarrow \beta$ or $\alpha ::= \beta$
    - Often terminals are quoted (which makes the $\langle$ and $\rangle$ unnecessary)
- Examples:

$$\begin{array}{lcl} \langle \text{exp} \rangle & ::= & CONST \\ & | & ID \\ & | & \langle \text{exp} \rangle\ \langle \text{op} \rangle\ \langle \text{exp} \rangle \\ & | & (\ \langle \text{exp} \rangle\ ) \\ \langle \text{op} \rangle & ::= & + \mid - \mid * \mid / \end{array}$$

$$\begin{array}{lcl} \langle \text{stmt} \rangle & ::= & ; \\ & | & ID = \langle \text{exp} \rangle\ ; \\ & | & \texttt{if}\ (\ \langle \text{exp} \rangle\ )\ \langle \text{stmt} \rangle\ \texttt{else}\ \langle \text{stmt} \rangle \\ & | & \texttt{while}\ (\ \langle \text{exp} \rangle\ )\ \langle \text{stmt} \rangle \\ & | & \{\ \langle \text{seq} \rangle\ \} \\ \langle \text{seq} \rangle & ::= & \varepsilon \mid \langle \text{stmt} \rangle\ \langle \text{seq} \rangle \end{array}$$

  - Notation: $\langle \text{A} \rangle ::= \alpha_1 \mid \alpha_2$ is a shorthand for $\langle \text{A} \rangle ::= \alpha_1$ and $\langle \text{A} \rangle ::= \alpha_2$

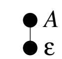
Derivations

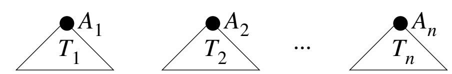
- A context-free grammar is a recipe for creating strings
- $G = (N, \Sigma, R, S)$
- A rewriting rule $A ::= v' \in R$ is used to rewrite its left-hand side (A) into its right-hand side (v'):
  - $u \Rightarrow v$ iff $\exists x, y \in (N|\Sigma)^* : \exists A \in N : u = xAy, v = xv'y, A ::= v' \in R$
- Rewriting can be chained ($\Rightarrow^*$, the reflexive and transitive closure of $\Rightarrow$ = derivation)
  - $s \Rightarrow^* s'$ iff $s = s'$, $s \Rightarrow s'$, or there exist strings $s_1, s_2, \ldots, s_n$ such that $s \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n \Rightarrow s'$
  - $\langle \text{pal} \rangle \Rightarrow 0\langle \text{pal} \rangle 0 \Rightarrow 01\langle \text{pal} \rangle 10 \Rightarrow 010\langle \text{pal} \rangle 010 \Rightarrow 0101010$

$$\langle \text{pal} \rangle ::= \varepsilon \mid 0 \mid 1 \mid 0\ \langle \text{pal} \rangle\ 0 \mid 1\ \langle \text{pal} \rangle\ 1$$

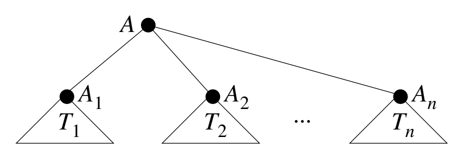
- The language generated by grammar G: exactly all the terminal strings generated from S: $\mathcal{L}(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}$

Parse Trees

- Definition:
  1. For every $a \in N|\Sigma$ the following is a parse tree (with yield a): 
  2. For every $A ::= \varepsilon \in R$ the following is a parse tree (with yield ε): 
  3. If the following are parse trees (with yields $y_1, y_2, \ldots, y_n$, respectively):



  and $A ::= A_1 A_2 \ldots A_n \in R$, then the following is a parse tree (w/ yield $y_1 y_2 \ldots y_n$):



- Yield: concatenation of leaves in inorder

Derivations and Parse Trees

- Every derivation starting from some nonterminal has an associated parse tree (rooted at the starting nonterminal)
- Two derivations are similar iff only the order of rule application varies = can obtain one derivation from the other by repeatedly flipping consecutive rule applications
  - Two similar derivations have identical parse trees
  - Can use a "standard" derivation: leftmost ($A \overset{L}{\Rightarrow}{}^* w$) or rightmost ($A \overset{R}{\Rightarrow}{}^* w$)

Theorem

The following statements are equivalent:

- *there exists a parse tree with root A and yield w*
- $A \Rightarrow^* w$
- $A \overset{L}{\Rightarrow}{}^* w$
- $A \overset{R}{\Rightarrow}{}^* w$

- Ambiguity of a grammar: there exists a string that has two derivations that are not similar (i.e., two derivations with diferent parse trees)
  - Can be inherent or not

Reduced Grammars

- Notation: $|w|_X$ = the length of string w after all the occurrences of symbols not in the set X have been erased
- A grammar may contain "useless" nonterminals, which do not participate in the derivation of strings
  - Unreachable nonterminal: a nonterminal A such that there does not exist a derivation $S \Rightarrow^* w$ such that $|w|_A \neq 0$
  - Non-productive nonterminal: a nonterminal A such that $A \Rightarrow^* w$ implies that $|w|_N \neq 0$
- Both unreachable and non-productive nonterminals can be found algorithmically
  - They can then be erased from the grammar (together with all the rules that contain them) without changing the language
  - We thus obtain a reduced grammar

Parsing

- Interface to lexical analysis:

```
typename vocab;        /* tokens + end-of-string */
const vocab EOS;       /* end-of-string pseudo-token */
vocab gettoken(void); /* returns next token */
```

- Parsing = determining whether the current input belongs to the given language
  - In practice a parse tree is constructed in the process as well
- Three types of parsers:
  - General parsers: not as efficient as for finite automata
    - Several possible derivations starting from the axiom, must choose the right one
    - Careful housekeeping (dynamic programming) reduces the otherwise exponential complexity to $O(n^3)$ – still too inefficient
  - Top-down parsers: construct the parse tree from root to leaves
    - Input is scanned left to right
    - Work only with the restricted class of LL grammars
    - Parsers usually (but not always) constructed by hand
  - Bottom-up parsers: construct the parse tree from leaves to root
    - Input is also scanned left to right
    - Work with the larger class of LR grammars
    - Parsers usually constructed using automated tools

Recursive Descent (Top Down) Parsing

- Construct a (possibly recursive) function for each nonterminal
- Decide which function to call based on the next input token = linear complexity

```
typedef enum { ID, EQ, IF, ELSE, WHILE, OPN_BRACE, CLS_BRACE,
               OPN_PAREN, CLS_PAREN, SEMICOLON, EOS            } vocab;
vocab gettoken() {...}
vocab t;
void MustBe (vocab ThisToken) {
    if (t != ThisToken) { printf("reject"); exit(0); }
    t = gettoken();
}

void Statement();
void Sequence();

int main() {
    t = gettoken();
    Statement();    /*axiom*/
    if (t != EOS) printf("String not accepted\n");
    return 0;        }
```

Recursive Descent Parsing (cont'd)

```
void Statement() {
    switch(t) {
    case SEMICOLON: /* ; */
        t = gettoken();
        break;
    case ID: /* <var> = <exp> */
        t = gettoken();
        MustBe(EQ);
        Expression();
        MustBe(SEMICOLON);
        break;
    case IF: /* if (<expr>) <statement> else <statement> */
        t = gettoken();
        MustBe(OPEN_PAREN);
        Expression();
        MustBe(CLS_PAREN);
        Statement();
        MustBe(ELSE);
        Statement();
        break;
```

⟨stmt⟩	::=	;
	|	*ID* = ⟨exp⟩ ;
	|	`if` (⟨exp⟩) ⟨stmt⟩ `else` ⟨stmt⟩
	|	`while` (⟨exp⟩) ⟨stmt⟩
	|	{ ⟨seq⟩ }
⟨seq⟩	::=	ε | ⟨stmt⟩ ⟨seq⟩

Recursive Descent Parsing (cont'd)

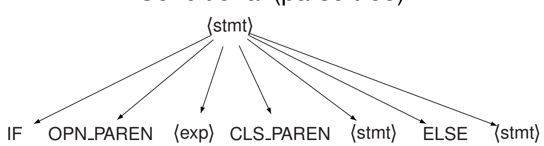
```
    case WHILE: /* while (exp) <statement> */
        t = gettoken();
        MustBe(OPEN_PAREN);
        Expression();
        MustBe(CLS_PAREN);
        Statement();
        break;
    default: /* { <sequence> } */
        MustBe(OPN_BRACE);
        Sequence();
        MustBe(CLS_BRACE);
    } /* switch */
} /* Statement () */

void Sequence() {
    if (t == CLS_BRACE) /* <empty> */ ;
    else { /* <statement> <sequence> */
        Statement();
        Sequence();
    }   }
```

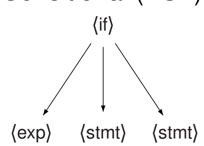

Parse Trees vs. Abstract Syntax Trees

- In practice the output of a parser is often a somehow simplified parse tree called abstract syntax tree (AST)
  - Some tokens in the program being parsed have only a syntactic role (to identify the respective language construct and its components)
  - Node information can be augmented to replace them
  - These tokens have no further use and so they are omitted form the AST
  - Other than this omission the AST looks exactly like a parse tree
- Examples of parse trees versus AST

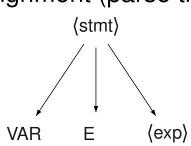
Conditional (parse tree):



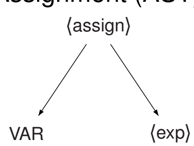
Consitional (AST):



Assignment (parse tree):



Assignment (AST):



Constructing the Parse Tree

- The parse tree/AST can be constructed through the recursive calls:
  - Each function creates a current node
  - The children are populated through recursive calls
  - The current node is then returned

```
class Node {...};

Node* Sequence() {
    Node* current = new Node(SEQ, ...);
    if (t == CLS_BRACE) /* <empty> */ ;
    else { /* <statement> <sequence> */
        current.addChild(Statement());
        current.addChild(Sequence());
    }
    return current;
}
```

Constructing the Parse Tree (cont'd)

```
Node* Statement() {
    Node* current;
    switch(t) {
    case SEMICOLON: /* ; */
        t = gettoken();
        return new Node(EMPTY);
        break;
    case ID: /* <var> = <exp> */
        current = new Node(ASSIGN, ...);
        current.addChild(ID, ...);
        t = gettoken();
        MustBe(EQ);
        current.addChild(Expression());
        MustBe(SEMICOLON);
        break;
    case IF: /* if (<expr>) <statement> else <statement> */
        current = new Node(COND, ...);
    /* ... */
    }
    return current;
}
```


Recursive Descent Parsing: Left Factoring

- Not all grammars are suitable for recursive descent:

⟨stmt⟩	::=	⟨empty⟩
	|	*ID* := ⟨exp⟩
	|	`IF` ⟨exp⟩ `THEN` ⟨stmt⟩ `ELSE` ⟨stmt⟩
	|	`WHILE` ⟨exp⟩ `DO` ⟨stmt⟩
	|	`BEGIN` ⟨seq⟩ `END`
⟨seq⟩	::=	⟨stmt⟩ | ⟨stmt⟩ ; ⟨seq⟩

  - Both rules for ⟨seq⟩ begin with the same nonterminal
  - Impossible to decide which one to apply based only on the next token
  - Fortunately concatenation is distributive over union so we can fix the grammar (left factoring):

⟨seq⟩	::=	⟨stmt⟩ ⟨seqTail⟩
⟨seqTail⟩	::=	⟨empty⟩ | ; ⟨seq⟩

Recursive Descent Parsing: Ambiguity

- Some programming constructs are inherently ambiguous

⟨stmt⟩	::=	`if` (⟨exp⟩) ⟨stmt⟩
	|	`if` (⟨exp⟩) ⟨stmt⟩ `else` ⟨stmt⟩

- Solution: choose one path and stick to it (e.g., match the else-statement with the nearest else-less if statement)

```
case IF:
    t = gettoken();
    MustBe(OPEN_PAREN);
    Expression();
    MustBe(CLS_PAREN);
    Statement();
    if (t == ELSE) {
        t = gettoken();
        Statement();
    }
```


Recursive Descent Parsing: Closure, etc.

- Any left recursion in the grammar will cause the parser to go into an infinite loop:

⟨exp⟩ ::= ⟨exp⟩ ⟨addop⟩ ⟨term⟩ | ⟨term⟩

- Solution: eliminate left recursion using a closure

⟨exp⟩	::=	⟨term⟩ ⟨closure⟩
⟨closure⟩	::=	⟨empty⟩
	|	⟨addop⟩ ⟨term⟩ ⟨closure⟩

  - Not the same language theoretically, but differences not relevant in practice
- This being said, some languages are simply not parseable using recursive descent

⟨palindrome⟩ ::= ⟨empty⟩ | 0 | 1 | 0 ⟨palindrome⟩ 0 | 1 ⟨palindrome⟩ 1

  - No way to know when to choose the ⟨empty⟩ rule
  - No way to choose between the second and the fourth rule
  - No way to choose between the third and the fifth rule

Recursive Descent Parsing: Sufficient Conditions

- $\textsc{First}(\alpha)$ = set of all initial tokens in the strings derivable from α
- $\textsc{Follow}(\langle N \rangle)$ = set of all initial tokens in nonempty strings that may follow $\langle N \rangle$ (possibly including `EOS`)
- Sufficient conditions for a grammar to allow recursive descent parsing:
  - For $\langle N \rangle ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ must have $\textsc{First}(\alpha_i) \cap \textsc{First}(\alpha_j) = \emptyset$, $1 \leq i < j \leq n$
  - Whenever $\langle N \rangle \Rightarrow^* \varepsilon$ must have $\textsc{Follow}(\langle N \rangle) \cap \textsc{First}(\langle N \rangle) = \emptyset$
- Grammars that do not have these properties may be fixable using left factoring, closure, etc.
- Method for constructing the recursive descent function `N()` for the nonterminal $\langle N \rangle$ with rules $\langle N \rangle ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$:
  1. For $\alpha_i \neq \varepsilon$ apply the rewriting rule $\langle N \rangle ::= \alpha_i$ whenever the next token in the input is in $\textsc{First}(\alpha_i)$
  2. For $\alpha_i = \varepsilon$ apply the rewriting rule $\langle N \rangle ::= \alpha_i$ (that is, $\langle N \rangle ::= \varepsilon$) whenever the next token in the input is in $\textsc{Follow}(\langle N \rangle)$
  3. Signal a syntax error in all the other cases

Algorithms for Computing First and Follow Sets

```
function First(α ∈ (Σ ∪ N)*) returns 2^Σ:
    foreach A ∈ N do
        VisitedFirst[A] ← False
    return AuxFirst(α)

function AuxFirst(α ∈ (Σ ∪ N)*) returns 2^Σ:
    if α = ε then return ∅
    x ← Head(α)
    β ← Tail(α)
    if x ∈ Σ then return {x}
    ans ← ∅
    if not VisitedFirst[x] then
        VisitedFirst[x] ← True
        foreach rule x ::= r do
            ans ← ans ∪ AuxFirst(r)
    if x ⇒* ε then ans ← ans ∪ AuxFirst(β)
    return ans

function Follow(A ∈ N) returns 2^Σ:
    foreach B ∈ N do
        VisitedFollow[B] ← False
    return AuxFollow(A)

function AuxFollow(A ∈ N) returns 2^Σ:
    ans ← ∅
    if not VisitedFollow[A] then
        VisitedFollow[A] ← True
        foreach rule X ::= uAw do
            ans ← ans ∪ First(w)
        if w ⇒* ε then
            ans ← ans ∪ AuxFollow(X)
    return ans
```