Programming Languages and Computation

Week 2: Context Free Grammars

* 1. Consider the following CFG G with start symbol R:

$$\begin{array}{lcl} R & ::= & XRX \mid S \\ S & ::= & aTb \mid bTa \\ T & ::= & XTX \mid X \mid \epsilon \\ X & ::= & a \mid b \end{array}$$

(a) What are the non-terminals of G?

(b) What are the terminals of G?

(c) Give three strings in $L(G)$.

(d) Give three strings not in $L(G)$.

(e) True or false: $T \to aba$.

(f) True or false: $T \to^* aba$.

(g) True or false: $T \to T$.

(h) True or false: $T \to^* T$.

(i) True or false: $XXX \to^* aba$.

(j) True or false: $X \to^* aba$.

(k) True or false: $T \to^* XX$.

(l) True or false: $T \to^* XXX$.

(m) True or false: $S \to^* \epsilon$.

Solution

(a) R, S, T, X

(b) a, b

(c) ab, ba, aab

The syntax of the *Brischeme* programming language (ignoring whitespace) is given by the CFG with:

- Terminals: $0, 1, \ldots, 9$, $a, b, \ldots, z, A, B, \ldots, Z$, _, !, ?, <, =, +, −, ∗, /, (,), define, lambda, not, or, and, #t, #f.
- Nonterminals: *Prog*, *Form*, *SExpr*, *Ident*, *Literal*, *Num*, *Bool*, *Digit*, *LChar*, *IdChar*, *Primop*.
- The production rules are:

$$
\begin{array}{rcl}
\textit{Prog} & ::= & \textit{Form}^* \\
\textit{Form} & ::= & \textit{SExpr} \\
 & | & \text{(define } \textit{Ident SExpr } \text{)} \\
\textit{SExpr} & ::= & \textit{Literal} \\
 & | & \textit{Ident} \\
 & | & \text{(} \textit{SExpr SExpr}^* \text{)} \\
 & | & \text{(} \textit{Primop SExpr}^* \text{)} \\
 & | & \text{(lambda (} \textit{Ident}^* \text{) } \textit{SExpr } \text{)} \\
\textit{Ident} & ::= & \textit{LChar IdChar}^* \\
\textit{LChar} & ::= & \text{a} \mid \text{b} \mid \cdots \mid \text{z} \\
\textit{IdChar} & ::= & a \mid b \mid \cdots \mid z \mid A \mid B \mid \cdots \mid \text{Z} \mid \text{!} \mid \text{?} \mid \text{\_} \\
\textit{Literal} & ::= & \textit{Bool} \mid \textit{Num} \\
\textit{Bool} & ::= & \text{\#t} \mid \text{\#f} \\
\textit{Num} & ::= & \textit{Digit Digit}^* \\
\textit{Digit} & ::= & 0 \mid 1 \mid \cdots \mid 9 \\
\textit{Primop} & ::= & \text{+} \mid \text{-} \mid \text{*} \mid \text{/} \mid \text{and} \mid \text{or} \mid \text{not} \mid \text{<} \mid \text{=}
\end{array}
$$

Figure 1: Brischeme (ignoring whitespace).

(d) aa, bb, ϵ

(e) false

(f) true

(g) false

(h) true

(i) true

(j) false

(k) true

(l) true

(m) false

* 2. Figure 1 contains a grammar for the Brischeme language from this week's lab sheet.

You will need to read the section Grammars can Express Sequences from the end of in the course notes to understand the notation *Form**, *SExpr**, *Ident** and *IdChar** from this grammar. Unfortunately, I did not have time to cover this in the lecture.

Which of the following are valid Brischeme programs (i.e. strings in the language of that grammar)? You do not have to give the derivations (but you should work through them in your head).

(a) (define x 32) (+ 4 x)

(b) (define x 32) (+ 4 y)

(c) (define x 32) (+ 4 5x)

(d) (define aC3! 32) (+ 4 aC3!)

(e) (define AC3! 32) (+ 4 AC3!)

(f) (+ (define x 32) x 4)

(g) (lambda x (+ x x))

(h) (define f (lambda (x y) (+ x (* 2 y))))

(i) (lambda (x b) (+ x (not b)))

(j) (lambda (x b) (if x))

(k) (lambda (x b) ((if b + -) 2 3))

(l) ((lambda () 3))

Solution

This question is a little unfair because I have been inconsistent on how I use whitespace (a question with such ambiguity would not appear in an exam). On the one hand, I told you that we essentially ignore whitespace for now, and on the other it seems to crucial to understand some of these examples. We will discuss whitespace in Week 3.

(a) yes

(b) yes

(c) no – identifiers cannot start with a digit. However, if we really ignore whitespace then it is possible to parse (+ 4 5x) as (+ 4 5 x) which is a valid s-expression.

(d) no – identifiers cannot contain a digit (although this is a discrepency with the version of Brischeme used in the implementation)

(e) no – identifiers must start with a lowercase letter.

(f) no – define is not derivable from *SExpr*, so cannot be nested in another s-expression. However, if we really ignore whitespace, then this could be understood as (+ (defi ne x 32) x 4), which is a valid s-expression.

(g) no – formal parameters to a lambda function must be parenthesized. However, if we really ignore whitespace, then this could be understood as (lam bda x (+ x x)), which is a valid s-expression.

(h) yes

(i) yes

(j) yes

(k) no – + and - are not derivable from *SExpr*

(l) yes

** 3. Design CFGs for the following programming language lexemes over the ASCII alphabet. You will find it convenient to use abbreviations like $\cdots$ to help present the expressions compactly.

(a) A *C program identifier* is any string of length at least 1 containing only letters ('a'–'z', lower and uppercase), digits ('0'–'9') and the underscore, and which begins with a letter or the underscore.

(b) An *integer literal* is any string taking one of the following forms:

- a non-empty sequence of digits (decimal)
- a non-empty sequence of characters from '0'–'9','a'–'e' (upper or lowercase) that are preceded by "0x" (hexadecimal)
- a non-empty sequence of bits '0' and '1' that are preceded by "0b" (binary)

Solution

(a) This is one way to describe it, I will write terminal symbols in terminal type to distinguish them:

$$\begin{array}{rcl} S & ::= & LM \\ M & ::= & LM \mid DM \mid \epsilon \\ L & ::= & \mathsf{a} \mid \mathsf{b} \mid \cdots \mid \mathsf{z} \mid \_ \mid \mathsf{A} \mid \mathsf{B} \mid \cdots \mid \mathsf{Z} \\ D & ::= & 0 \mid 1 \mid \cdots \mid 9 \end{array}$$

(b) One straightforward way is as follows (here I use terminal type for the terminal symbols to distinguish them from nonterminals):

$$\begin{array}{rcl} S & ::= & D \mid H \mid B \\ D & ::= & ND \mid N \\ N & ::= & 0 \mid 1 \mid \cdots \mid 9 \\ H & ::= & 0\mathsf{x}G \\ G & ::= & AG \mid A \\ A & ::= & N \mid \mathsf{a} \mid \mathsf{b} \mid \cdots \mid \mathsf{e} \mid \mathsf{A} \mid \mathsf{B} \mid \cdots \mid \mathsf{E} \\ B & ::= & 0\mathsf{b}Z \\ Z & ::= & YZ \mid Y \\ Y & ::= & 0 \mid 1 \end{array}$$

* 4. Consider the following grammar, which describes the structure of statements in an imperative programming language.

$$\begin{array}{rcll} \mathit{Prog} & ::= & \mathit{Stmt}\ \mathit{Stmts} & (1) \\ \mathit{Stmt} & ::= & \mathsf{if\ exp\ then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt} & (2) \\ & \mid & \mathsf{while\ exp\ do}\ \mathit{Stmt} & (3) \\ & \mid & \mathsf{skip} & (4) \\ & \mid & \mathsf{id} \leftarrow \mathsf{exp} & (5) \\ & \mid & \{\mathit{Stmt}\ \mathit{Stmts}\} & (6) \\ \mathit{Stmts} & ::= & ;\ \mathit{Stmt}\ \mathit{Stmts} & (7) \\ & \mid & \epsilon & (8) \end{array}$$

In it expressions appear only as a terminal symbols exp because the structure of expressions is not important to the exercises. In total, the terminal symbols are: if, then, else, while, do, skip, id, exp, the left and right braces, the end-of-input marker and the semicolon. The rules are numbered to make constructing the parse tables easier. The language of this grammar (start symbol *Prog*) includes strings such as:

$$\mathsf{while\ exp\ do\ id} \leftarrow \mathsf{exp}$$

i.e. strings that show the control structure of the program without specifying the particular expressions involved.
The nullable, first and follow maps for the nonterminals in this grammar are as follows:

Nonterminal	Nullable?	First	Follow
Prog	×	if, while, skip, id, {	
Stmt	×	if, while, skip, id, {	else, ;, }
Stmts	✓	;	}

(a) For each rule $X ::= \alpha$ numbered (1) – (8), compute First(α), the set of terminal symbols that can start any string derivable from the rule right-hand side α.

(b) Construct the parsing table for the grammar.

(c) Is the grammar LL(1)?

Solution

(a)

$$\begin{aligned}
\mathsf{First}(\mathit{Stmt}\ \mathit{Stmts}) &= \mathsf{if},\ \mathsf{while},\ \mathsf{skip},\ \mathsf{id}\,\{\\
\mathsf{First}(\mathsf{if}\ \mathsf{exp}\ \mathsf{then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt}) &= \mathsf{if}\\
\mathsf{First}(\mathsf{while}\ \mathsf{exp}\ \mathsf{do}\ \mathit{Stmt}) &= \mathsf{while}\\
\mathsf{First}(\mathsf{skip}) &= \mathsf{skip}\\
\mathsf{First}(\mathsf{id} \leftarrow \mathsf{exp}) &= \mathsf{id}\\
\mathsf{First}(\{\mathit{Stmt}\ \mathit{Stmts}\}) &= \{\\
\mathsf{First}(;\ \mathit{Stmt}\ \mathit{Stmts}) &= ;\\
\mathsf{First}(\epsilon) &=
\end{aligned}$$

(b)

Nonterminal	if	exp	then	else	while	do	skip	id	;	{	}
Prog	1				1		1	1		1	
Stmt	2				3		4	5		6	
Stmts									7		8

(c) Yes.

* 5. Consider now the following grammar:

$$\begin{array}{rcll}
\mathit{Prog} & ::= & \mathit{Stmt}\ \mathit{Stmts} & (1)\\
\mathit{Stmt} & ::= & \mathsf{if}\ \mathsf{bexp}\ \mathsf{then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt} & (2)\\
 & | & \mathsf{while}\ \mathsf{bexp}\ \mathsf{do}\ \mathit{Stmt} & (3)\\
 & | & \mathsf{skip} & (4)\\
 & | & \mathsf{id} \leftarrow \mathit{aexp} & (5)\\
 & | & \mathit{Stmt}\ \mathit{Stmts} & (6)\\
\mathit{Stmts} & ::= & ;\ \mathit{Stmt}\ \mathit{Stmts} & (7)\\
 & | & \epsilon & (8)
\end{array}$$

This grammar is the same as the previous one, except that braces have been removed in rule 6. The Nullable, First and Follow maps for this grammar can be tabulated as follows.

Nonterminal	Nullable?	First	Follow
Prog	×	if, while, skip, id	
Stmt	×	if, while, skip, id	;, else
Stmts	✓	;	;, else

(a) For each rule $X ::= \alpha$ numbered (1) – (8), compute $\mathsf{First}(\alpha)$, the set of terminal symbols that can start any string derivable from the rule right-hand side α.

(b) Construct the parsing table for this grammar.

(c) Is the grammar LL(1)?

Solution

(a)

(1)	First(*Stmt Stmts*)	=	if, while, skip, id
(2)	First(if exp then *Stmt* else *Stmt*)	=	if
(3)	First(while exp do *Stmt*)	=	while
(4)	First(skip)	=	skip
(5)	First(id ← exp)	=	id
(6)	First(*Stmt Stmts*)	=	if, while, skip, id
(7)	First(; *Stmt Stmts*)	=	;
(8)	First(ϵ)	=	

(b) The parsing table is as follows:

Nonterminal	$	if	bexp	then	else	while	do	skip	id	aexp	;
Prog		1				1		1	1		
Stmt		2, 6				3, 6		4, 6	5, 6		
Stmts	8				8						7, 8

(c) No.

** 6. Give CFGs for the following languages. The later parts are more difficult than 2-star.

(a) All odd length strings over $\{a, b\}$.

(b) All strings over $\{a, b\}$ that contain aab as a substring.

(c) The set of strings over $\{a, b\}$ with more a than b. Hint: every string w with at least as many a as b (possibly the same number of a as b) can be characterised inductively as follows. Either:

- w is just a
- or, w is of shape avb with v containing at least as many a as b
- or, w is of shape bva with v containing at least as many a as b
- or, w is of shape v_1v_2 with v_1 and v_2 each separately containing at least as many a as b
- or, w is the empty string

(d) The complement of the language $\{a^n b^n \mid n \geq 0\}$ over $\{a, b\}$. Hint: express "not of shape $a^n b^n$ for some n" into one or more positive (i.e. without using *not* or similar) conditions.

(e) $\{v\#w \mid v, w \in \{a, b\}^*$ and the reverse of v is a substring of $w\}$, over $\{a, b, \#\}$.

Solution

(a)

$$\begin{array}{rcl} S & ::= & XSX \mid X \\ X & ::= & a \mid b \end{array}$$

(b)

$$\begin{array}{rcl} S & ::= & XSX \mid aab \\ X & ::= & a \mid b \mid \epsilon \end{array}$$

(c) Here we dedicate a nonterminal T to describe the inductive characterisation given in the hint and then use S to force an extra a.

$$\begin{array}{rcl} S & ::= & TaT \\ T & ::= & a \mid aTb \mid bTa \mid TT \mid \epsilon \end{array}$$

(d) When you are asked to give the complement, or otherwise to describe something according to a negated condition, it is best to translate it immediately into one or more positive conditions because CFG have no direct way to express negation. In this case, not of shape $a^n b^n$ means either:

- The word has all the a before any b but there are strictly more a than b (nonterminal R),
- or, the word has all the a before any b, but there are more b than a (nonterminal T),
- or, the word has a b occurring before an a (non-terminal U).

$$\begin{array}{rcl} S & ::= & R \mid T \mid U \\ R & ::= & AaRb \mid a \\ A & ::= & aA \mid \epsilon \\ T & ::= & aTbB \mid b \\ B & ::= & bB \mid \epsilon \\ U & ::= & XbXaX \\ X & ::= & aX \mid bX \mid \epsilon \end{array}$$

(e) Strings in this language have the following shape: $v\#w_1v^Rw_2$ where w_1 and w_2 are completely arbitrary. We need to match v to v^R letter by letter, so this is best done by a single nonterminal (T). The "middle" of the string generated by this non-terminal (between the matching pieces of v and its reverse) will contain the # and, to the right of it, any arbitrary substring w_1. Finally, the whole string has an arbitrary suffix w_2. We can describe this as follows:

$$\begin{array}{rcl} S & ::= & TX \\ X & ::= & aX \mid bX \mid \epsilon \\ T & ::= & aTa \mid bTb \mid \#X \end{array}$$