Today
  • Grammars (cont)

Assignments
  • EX-2 out
  • HW-2 out
Concatenation

\[ S \rightarrow ab \]

- \( S \) yields the string \( a \) followed by the string \( b \)

\[ T \rightarrow UV \]
\[ U \rightarrow a \]
\[ V \rightarrow b \]

- here \( T \) yields the same exact string as \( S \)
Alternation

\[ S \rightarrow a \mid b \]
- \( S \) yields the string \( a \) or \( b \)

- The \( \mid \) symbol is special (meta) syntax for separate \( S \)-rules:
  \[ S \rightarrow a \]
  \[ S \rightarrow b \]

The empty string

\[ S \rightarrow a \mid \epsilon \]
- \( \epsilon \) denotes the special “empty” terminal
- \( S \) yields either the string \( a \) or "" (empty string)

Kleene Star (Closure)

\[ S \rightarrow a^* \]
- \( S \) yields the strings with zero or more \( a \)'s
- e.g., "", a, aa, aaa, and in general \( a^n \) for \( n \geq 0 \)

\[ S \rightarrow a^*b^* \]
- \( S \) yields strings with zero or more \( a \)'s followed by zero or more \( b \)'s
- e.g., "", a, b, aa, ab, bb, and in general \( a^n b^m \) for \( n, m \geq 0 \)
Recursion

Either directly when used in same rule, or indirectly ...

Direct Example:  \[ S \rightarrow \{S\} \mid \{\} \]
- \( S \) yields the strings of “balanced” open-close curly braces
- note not possible to express using \( \ast \) (Kleene star)
- however, \( \ast \) can be implemented using recursion (w/ the empty string ...)

Indirect Example:
\[
\begin{align*}
S & \rightarrow T \mid \{\} \\
T & \rightarrow \{S\}
\end{align*}
\]

Q: How can we represent \( S \rightarrow aa^* \) using recursion?
\[
S \rightarrow a \mid aS
\]
- sometimes denoted as \( S \rightarrow a^+ \)
**MyPL Constants**

Using grammar rules to define constant values

Note that we use BNF below ...

- where ::= used instead of →
- and non-terminals as <name>
- we also use our token types (which is not BNF)

```plaintext
BOOL_VAL ::= 'true' | 'false'
INT_VAL ::= <pdigit> <digit>* | '0'
DOUBLE_VAL ::= INT_VAL '.' <digit> <digit>*
STRING_VAL ::= "" <character>* ""
ID ::= <letter> (<letter> | <digit> | '_')*
  <letter> ::= 'a' | ... | 'z' | 'A' | ... | 'Z'
  <pdigit> ::= '1' | ... | '9'
  <digit> ::= '0' | <pdigit>
```

... where <character> is any symbol (letter, number, etc.) except '""
**Parsing: An example grammar**

Simple list of assignment statements

\[
\begin{align*}
\langle \text{stmt\_list} \rangle & \ ::= \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle \,' \langle \text{stmt\_list} \rangle \\
\langle \text{stmt} \rangle & \ ::= \langle \text{var} \rangle \,'=\langle \text{expr} \rangle \\
\langle \text{var} \rangle & \ ::= \ 'A' \mid \ 'B' \mid \ 'C' \\
\langle \text{expr} \rangle & \ ::= \langle \text{var} \rangle \mid \langle \text{var} \rangle \,'+\langle \text{var} \rangle \mid \langle \text{var} \rangle \,'-\langle \text{var} \rangle
\end{align*}
\]

– Note: many possible grammars for this language!

We can use grammars to generate strings (derivations)

1. choose a rule (e.g., with start symbol on left-hand side)
2. replace with right-hand side (of rule)
3. pick a non-terminal \( N \) and rule with \( N \) on left side
4. replace \( N \) with rule’s right-hand side
5. repeat from 3 until only terminals remain

 Whereas \( \rightarrow \) (or \( ::= \)) denotes a rule, \( \Rightarrow \) denotes a derivation
Example derivation of “A = B + C; B = A”

<stmt_list> ⇒ <stmt> ; <stmt_list>
  ⇒ <var> = expr ; <stmt_list>
  ⇒ A = expr ; <stmt_list>
  ⇒ A = <var> + <var> ; <stmt>
  ⇒ A = B + <var> ; <stmt_list>
  ⇒ A = B + C ; <stmt_list>
  ⇒ A = B + C ; <stmt>
  ⇒ A = B + C ; <var> = <expr>
  ⇒ A = B + C ; B = <expr>
  ⇒ A = B + C ; B = <var>
  ⇒ A = B + C ; B = C

• This is a “left-most” derivation
  – derived the string by replacing left-most non-terminals

• The opposite is a "right-most" derivation
  <stmt_list> ⇒ <stmt> ; <stmt_list>
    ⇒ <stmt> ; <stmt>
    ⇒ <stmt> ; <var> = <expr>
    ⇒ <stmt> ; <var> = <var>
    ⇒ <stmt> ; <var> = B
    ⇒ ...