Translating Prefix/Postfix Notations to Lisp/“rpnLisp”

Recall:

- ***Prefix*** notation = ***Lisp*** *notation* *without parentheses*.
- ***Postfix*** notation = *“**rpnLisp**” notation* *without parentheses*.

Lisp:	$(*_3\ x\ (-_2\ (+_3\ 2\ 3\ y\ )\ (*_2\ w\ x)\ )\ 5\ )$
Prefix notation:	$*_3\ x\ -_2\ +_3\ 2\ 3\ y\ *_2\ w\ x\ 5$
rpnLisp:	$(\ x\ (\ (2\ 3\ y\ +_3)\ (w\ x\ *_2)\ -_2)\ 5\ *_3)$
Postfix notation:	$x\ 2\ 3\ y\ +_3\ w\ x\ *_2\ -_2\ 5\ *_3$

Q. Given a prefix / postfix expression, how can we insert parentheses to produce an equivalent Lisp / rpnLisp expression?

A. We can use variants of the stack-based algorithms for evaluating prefix / postfix expressions.

Notation: We will write $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$ and $\boxed{e_1\ \ldots\ e_k\ \textbf{op}}$ for the Lisp and rpnLisp expressions $(\textbf{op}\ e_1\ \ldots\ e_k)$ and $(e_1\ \ldots\ e_k\ \textbf{op})$.

Translating Prefix/Postfix Notations to Lisp/“rpnLisp”

Q. Given a prefix / postfix expression, how can we insert parentheses to produce an equivalent Lisp / rpnLisp expression?

A. We can use variants of the stack-based algorithms for evaluating prefix / postfix expressions.

Notation: We will write $\boxed{\mathbf{op}\ e_1\ \ldots\ e_k}$ and $\boxed{e_1\ \ldots\ e_k\ \mathbf{op}}$ for the Lisp and rpnLisp expressions ($\mathbf{op}\ e_1\ \ldots\ e_k$) and ($e_1\ \ldots\ e_k\ \mathbf{op}$).

The Lisp expression $(*_3\ x\ (-_2\ (+_3\ 2\ 3\ y)\ (*_2\ w\ x))\ 5)$

will be written $\boxed{*_3\ x\ \boxed{-_2\ \boxed{+_3\ 2\ 3\ y}\ \boxed{*_2\ w\ x}}\ 5}$

The rpnLisp expression $(x\ ((2\ 3\ y\ +_3)\ (w\ x\ *_2)\ -_2)\ 5\ *_3)$

will be written $\boxed{x\ \boxed{\boxed{2\ 3\ y\ +_3}\ \boxed{w\ x\ *_2}\ -_2}\ 5\ *_3}$

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions $e_k, \ldots, e_1$.
  - *Push* the rpnLisp expr $\boxed{e_1 \ldots e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the "rpnLisp" equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: **x 2 3 $+_3$ y $-_2$ u x 5 $*_2$ $*_3$ $-_1$**

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

***UNREAD* INPUT:** **x 2 3 $+_3$ y $-_2$ u x 5 $*_2$ $*_3$ $-_1$**

STACK:

We can use a stack as follows to translate a postfix expression to "rpnLisp":

- Read the expression *from left to right*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
  - *Pop* off k expressions $e_k, \ldots, e_1$.
  - *Push* the rpnLisp expr $\boxed{e_1 \ \ldots \ e_k \ \mathbf{op}}$.

After the entire expression has been processed in this way, the "rpnLisp" equivalent of the postfix expression will be the only thing on the stack.

Example Translate the following postfix expression into rpnLisp: $\mathbf{x \quad 2 \quad 3 \quad +_3 \quad y \quad -_2 \quad u \quad x \quad 5 \quad *_2 \quad *_3 \quad -_1}$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

rpnLisp equivalent of the expression

$$\boxed{\boxed{\boxed{\boxed{\mathbf{x \quad 2 \quad 3 \quad +_3}} \quad \mathbf{y \quad -_2}} \quad \mathbf{u} \quad \boxed{\mathbf{x \quad 5 \quad *_2}} \quad \mathbf{*_3}} \quad \mathbf{-_1}}$$

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from* *right to left*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
    - *Pop* off k expressions e_1, ..., e_k.
    - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$.

After the entire expression has been processed in this way, the Lisp equivalent of the prefix expression will be the only thing on the stack.

Example Translate the following prefix expression into Lisp.

$$*_3 \quad x \quad -_2 \quad +_3 \quad 2 \quad 3 \quad y \quad *_2 \quad w \quad x \quad 5$$

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:** $*_3 \quad x \quad -_2 \quad +_3 \quad 2 \quad 3 \quad y \quad *_2 \quad w \quad x \quad 5$

STACK:

We can use a stack as follows to translate a **prefix** expression to **Lisp**:

- Read the expression *from right to left*.
- *Push* each variable or constant that is seen.
- Whenever a k-ary operator **op** is seen:
    - *Pop* off k expressions e_1, ..., e_k.
    - *Push* the Lisp expression $\boxed{\textbf{op}\ e_1\ \ldots\ e_k}$.

After the entire expression has been processed in this way, the Lisp equivalent of the prefix expression will be the only thing on the stack.

Example Translate the following prefix expression into Lisp.

$$*_3\ \ x\ \ -_2\ \ +_3\ \ 2\ \ 3\ \ y\ \ *_2\ \ w\ \ x\ \ 5$$

$+_3$ and $*_3$ are 3-ary operators; $*_2$ and $-_2$ are binary operators.

***UNREAD* INPUT:**

STACK: $\boxed{*_3\ \ x\ \ \boxed{-_2\ \ \boxed{+_3\ \ 2\ \ 3\ \ y}\ \ \boxed{*_2\ \ w\ \ x}}\ \ 5}$

Lisp equivalent of the expression

Note that:

- The structure of the Lisp / rpnLisp equivalent of a prefix / postfix expression does ***not*** depend on the names and semantics of the operators, but only depends on the *arities* of the operators.

For example, the problem

Translate the following postfix expression into rpnLisp:

x 2 3 $@_3$ y $\#_2$ u x 5 $\hat{}_2$ $!_3$ $\sim_1$

Here $@_3$ and $!_3$ are 3-ary, $\hat{}_2$ and $\#_2$ are binary, and $\sim_1$ is unary.

is essentially equivalent to the problem

Translate the following postfix expression into rpnLisp:

x 2 3 $+_3$ y $-_2$ u x 5 $*_2$ $*_3$ $-_1$

Here $+_3$ and $*_3$ are 3-ary, $*_2$ and $-_2$ are binary, and $-_1$ is unary.

that we solved above: Substituting $@_3$, $!_3$, $\hat{}_2$, $\#_2$, and $\sim_1$ for $+_3$, $*_3$, $*_2$, $-_2$, and $-_1$ in our solution to the latter problem gives a solution to the former problem.

THIS IS THE “CUTOFF POINT” FOR EXAM 1

Questions on Exam 1 may assume knowledge of material presented on the preceding slides, but no question on Exam 1 will assume knowledge of material presented on the rest of today’s slides.

Context-Free Syntax of Programming Languages

Context-Free Grammars

Grammars were invented by Chomsky in the mid-1950s for describing natural languages. In the late 1950s, a notation equivalent to one of Chomsky's types of grammar (his ***Type 2*** or ***context-free*** grammars) was proposed by Backus as a way to specify the syntax of the new language Algol.

Backus's notation was improved by Naur and used in the **Algol 60 Report** (edited by Naur), an influential document that did an excellent job of specifying Algol 60.

The grammar notation used in the Algol 60 Report is now called "Backus Naur Form" or **BNF**.

Like many authors (but unlike Sethi), we use the term ***BNF*** more loosely, to simply mean "*a commonly used notation for writing context-free grammars*"; we refer to grammars written in such a notation as ***BNF specifications***.

$$
\begin{aligned}
\langle \textit{expression} \rangle \; &::= \; \langle \textit{expression} \rangle + \langle \textit{term} \rangle \\
&\;\;\mid \;\; \langle \textit{expression} \rangle - \langle \textit{term} \rangle \\
&\;\;\mid \;\; \langle \textit{term} \rangle \\
\langle \textit{term} \rangle \; &::= \; \langle \textit{term} \rangle * \langle \textit{factor} \rangle \\
&\;\;\mid \;\; \langle \textit{term} \rangle \,/\, \langle \textit{factor} \rangle \\
&\;\;\mid \;\; \langle \textit{factor} \rangle \\
\langle \textit{factor} \rangle \; &::= \; \textbf{number} \\
&\;\;\mid \;\; \textbf{name} \\
&\;\;\mid \;\; (\, \langle \textit{expression} \rangle \,)
\end{aligned}
$$

A grammar written in BNF notation on p. 46 of Sethi (p. 47 in the course reader).

Figure 2.10 BNF syntactic rules for arithmetic expressions.

On p. 42, Sethi gives this equivalent grammar that is written in a similar notation. ***We will consider this notation to be BNF***, even though it isn't exactly the same as the notation used in the Algol 60 Report and so Sethi does ***not*** call it BNF.

$$
\begin{aligned}
E \; &::= \; E + T \mid E - T \mid T \\
T \; &::= \; T * F \mid T \,/\, F \mid F \\
F \; &::= \; \textbf{number} \mid \textbf{name} \mid (\, E \,)
\end{aligned}
$$

Figure 2.6 A grammar for arithmetic expressions.

We will use the term ***grammar*** to mean "context-free grammar"; we will not consider other types of grammar.

- A grammar is a relatively concise way to precisely specify certain (possibly infinite) *sets of finite sequences of symbols*; those symbols are referred to as ***terminals*** of the grammar.
- Each of the specified *sets of finite sequences of terminals* is denoted by a ***nonterminal*** of the grammar.
- One of the nonterminals is regarded as the "most important": It is called the ***starting nonterminal*** (or *start symbol* or *sentence symbol*); the set of sequences of terminals it denotes is called the ***language*** *generated by* (or *language of*) the grammar.
- We commonly think of the other nonterminals as auxiliary nonterminals that are defined for use in defining the starting nonterminal.

⟨*real-number*⟩ ::= ⟨*integer-part*⟩ . ⟨*fraction*⟩
⟨*integer-part*⟩ ::= ⟨*digit*⟩ | ⟨*integer-part*⟩ ⟨*digit*⟩
⟨*fraction*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*fraction*⟩
⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

In the above grammar:

The following characters are the 11 ***terminals***:

. 0 1 2 3 4 5 6 7 8 9

A ***terminal*** of a grammar is a constant symbol that is ***not*** defined by the grammar.

The following are the 4 ***nonterminals***:

<real-number> *<integer-part>* *<fraction>* *<digit>*

A ***nonterminal*** of a grammar is a variable that denotes a *set of finite sequences of terminals*. For example, *<digit>* denotes the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

⟨*real-number*⟩ ::= ⟨*integer-part*⟩ . ⟨*fraction*⟩
⟨*integer-part*⟩ ::= ⟨*digit*⟩ | ⟨*integer-part*⟩ ⟨*digit*⟩
⟨*fraction*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*fraction*⟩
⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

In the above grammar:

There are 15 rules called ***productions***. Each production:

- has a left side that is a *single nonterminal*, and
- has a right side that is a *sequence of 0 or more terminals and/or nonterminals*.

The "vertical bar" symbol | means:

The left side of this production is the same as the left side of the ***previous*** *production.*

Example: The **3rd** production of the above grammar is

<integer-part> ::= *<integer-part> <digit>*

⟨*real-number*⟩ ::= ⟨*integer-part*⟩ . ⟨*fraction*⟩
⟨*integer-part*⟩ ::= ⟨*digit*⟩ | ⟨*integer-part*⟩ ⟨*digit*⟩
⟨*fraction*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*fraction*⟩
⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

Grammar notation is "free format": We can insert whitespace characters, ***including newlines***, between symbols without changing the specified grammar!

For example, the 2nd and 3rd productions

<integer-part> ::= *<digit>* | *<integer-part> <digit>*

of the above grammar could be ***rewritten*** as:

```
<integer-part> ::=  <digit>
                 |  <integer-part> <digit>
```

Intuitively, a production *N* ::= ... means "any ... is an *N*".

$$
\begin{array}{rcl}
\langle \textit{real-number} \rangle & ::= & \langle \textit{integer-part} \rangle \;.\; \langle \textit{fraction} \rangle \\
\langle \textit{integer-part} \rangle & ::= & \langle \textit{digit} \rangle \mid \langle \textit{integer-part} \rangle \, \langle \textit{digit} \rangle \\
\langle \textit{fraction} \rangle & ::= & \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \, \langle \textit{fraction} \rangle \\
\langle \textit{digit} \rangle & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

Figure 2.3 BNF rules for real numbers.

<*real-number*> is the ***starting nonterminal*** of the above grammar.

In this course, we use the convention that ***unless otherwise indicated***, *the starting nonterminal of a grammar is the nonterminal on the left side of the* ***first*** *production*:

If you write a grammar and want *some other* nonterminal to be its starting nonterminal, then you must ***explicitly indicate*** which nonterminal is the starting nonterminal!

⟨*real-number*⟩ ::= ⟨*integer-part*⟩ . ⟨*fraction*⟩
⟨*integer-part*⟩ ::= ⟨*digit*⟩ | ⟨*integer-part*⟩ ⟨*digit*⟩
⟨*fraction*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*fraction*⟩
⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

<*empty*> denotes the empty string; other people write ϵ or λ to denote the empty string.

Example: Changing the 2nd production above from *<integer-part> ::= <digit>* to *<integer-part> ::= <empty>* will allow a number with ***no digits before the point*** (e.g., .213) to belong to the language of the grammar.

Note that <*empty*> is ***neither*** a terminal ***nor*** a nonterminal!

Parse Trees

Q. Exactly which sequences of terminals belong to the set of sequences of terminals that is denoted by a given nonterminal ***N*** of a grammar?

A. A sequence of terminals $t_1 \ldots t_k$ belongs to the set of sequences denoted by a nonterminal ***N*** *if and only if* there is a **parse tree with root *N* that generates** $t_1 \ldots t_k$.

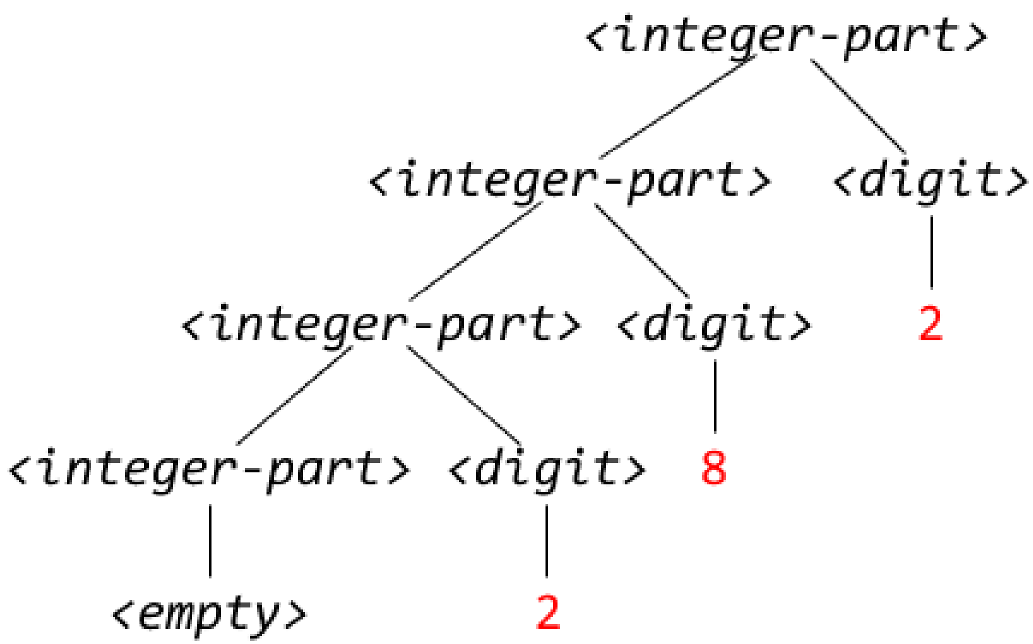
Unless otherwise indicated, the term ***parse tree*** means *parse tree <u>whose root is the starting nonterminal</u>.*

So we can say that a sequence of terminals $t_1 \ldots t_k$ belongs to the language of a grammar *if and only if* there is a **parse tree that generates** $t_1 \ldots t_k$.

Comment: Instead of using parse trees, we can also answer the above question using the concept of a ***<u>derivation</u>*** that is introduced on pp. 40 – 41 of Sethi.

Below is a parse tree, whose root is *<integer-part>*, that shows 282 belongs to the set of sequences denoted by *<integer-part>* in the following grammar:

<real-number> ::= *<integer-part>* . *<fraction>*
<integer-part> ::= *<empty>* | *<integer-part>* *<digit>*
<fraction> ::= *<digit>* | *<digit>* *<fraction>*
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Note: This is just a picture to show what parse trees look like.

A precise definition of a parse tree will be given below.

Given a nonterminal N, a ***parse tree with root N*** is an ordered rooted tree with the following properties:

1. The **root** is the nonterminal N.
2. Each **leaf** either is a terminal or is <*empty*>; moreover, a leaf that is <*empty*> has no sibling.
3. Each **internal node** is a nonterminal.
4. The left-to-right sequence of children of any internal node M is the right side of a production whose left side is the nonterminal M.

Unless otherwise indicated, the term ***parse tree*** means *parse tree whose root is the starting nonterminal*.

Given terminals t_1, ..., t_k, a ***parse tree with root N that generates*** t_1 ... t_k (or ***parse tree with root N for*** t_1 ... t_k or ***parse tree with root N of*** t_1 ... t_k) is a parse tree with root N for which the left-to-right sequence of leaves that are not <*empty*> is t_1, ..., t_k.