–contents of Syntax Tree–

–requires–

Building Syntax Trees

Building a syntax tree from a shift/reduce sequence requires a simple algorithm:

- Stack each shifted token.
- Build a node from the top of the stack to match each reduce.

Here is the IOG for *Proposition*. It describes the constant formulas of the propositional calculus. It has 10 rules. The rule numbers are used as output symbols. The *chicken-foot* symbol $\perp$ is used for end-of-file.

Proposition	$\leftarrow$ *Disjunction* $\perp$	0
Disjunction	$\leftarrow$ *Disjunction* $\vee$ *Conjunction*	1
Disjunction	$\leftarrow$ *Conjunction*	2
Conjunction	$\leftarrow$ *Conjunction* $\wedge$ *Negation*	3
Conjunction	$\leftarrow$ *Negation*	4
Negation	$\leftarrow$ $\neg$ *Boolean*	5
Negation	$\leftarrow$ *Boolean*	6
Boolean	$\leftarrow$ t	7
Boolean	$\leftarrow$ f	8
Boolean	$\leftarrow$ (*Disjunction*)	9

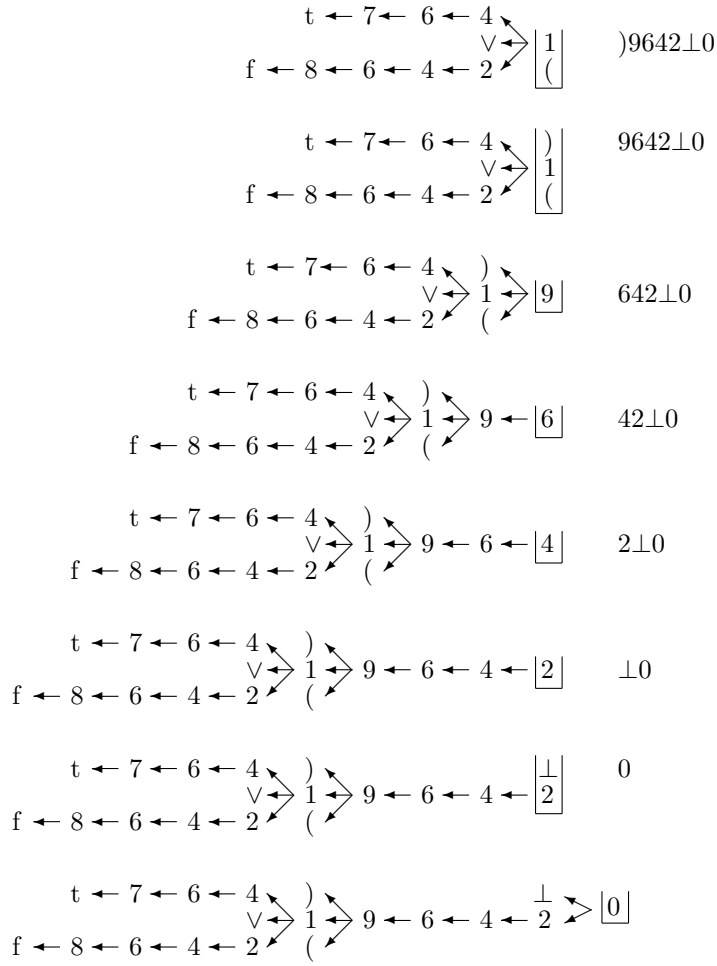
Applying the two-rule algorithm to the shift/reduce sequence leads to a syntax tree. The nodes are labelled with the rule numbers. The input symbols are at the leaves.

An Example

In the display of the steps below, the input text is on the right and the successive states of the stack are on the left. The entries in the stack are tree nodes. The tree is pushed out to the left (horizontal tree).

The shift/reduce sequence for
    (f$\vee$t)$\perp$
is
    (f8642$\vee$t7641)9642$\perp$0

treelets	*stack*	*input*
	$\lfloor\ \rfloor$	(f8642∨t7641)9642⊥0
	$\lfloor (\rfloor$	f8642∨t7641)9642⊥0
	$\lfloor \begin{matrix} \mathrm{f} \\ (\end{matrix} \rfloor$	8642∨t7641)9642⊥0
f ←	$\lfloor \begin{matrix} 8 \\ (\end{matrix} \rfloor$	642∨t7641)9642⊥0
f ← 8 ←	$\lfloor \begin{matrix} 6 \\ (\end{matrix} \rfloor$	42∨t7641)9642⊥0
f ← 8 ← 6 ←	$\lfloor \begin{matrix} 4 \\ (\end{matrix} \rfloor$	2∨t7641)9642⊥0
f ← 8 ← 6 ← 4 ←	$\lfloor \begin{matrix} 2 \\ (\end{matrix} \rfloor$	∨t7641)9642⊥0
f ← 8 ← 6 ← 4 ←	$\lfloor \begin{matrix} \vee \\ 2 \\ (\end{matrix} \rfloor$	t7641)9642⊥0
f ← 8 ← 6 ← 4 ←	$\lfloor \begin{matrix} \mathrm{t} \\ \vee \\ 2 \\ (\end{matrix} \rfloor$	7641)9642⊥0
t ←	$\lfloor \begin{matrix} 7 \\ \vee \\ 2 \\ (\end{matrix} \rfloor$	641)9642⊥0
f ← 8 ← 6 ← 4 ←		
t ← 7 ←	$\lfloor \begin{matrix} 6 \\ \vee \\ 2 \\ (\end{matrix} \rfloor$	41)9642⊥0
f ← 8 ← 6 ← 4 ←		
t ← 7 ← 6 ←	$\lfloor \begin{matrix} 4 \\ \vee \\ 2 \\ (\end{matrix} \rfloor$	1)9642⊥0
f ← 8 ← 6 ← 4 ←		



Tree Transformations

While xcom treats trees as read-only, there is a whole class of useful transformations that can be applied to trees to improve the quality of the resulting compiled code. Some of these are described under the heading of **Optimization**.

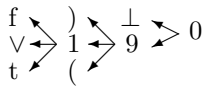
Tree transformations lead to the creation of new nodes and the abandonment of others. The tree mechanism needs to support these dynamic demands. Some tree transformations lead to trees that could not represent a program as written. This is ok until a diagnostic needs to be issued for a tree construct that has no image in the source text. All of these complexities and more have been solved in most modern compilers.

Intermediate Representation

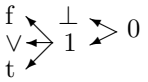
The result of transformations often takes the data structure far from its origins as a syntax tree. In many compilers the information passed from front-end to back-end is called the *intermediate representation* or IR.

Abstract Syntax Tree

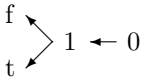
Most of the syntax tree is occupied by the empty transitions from one phrase name to the next. The value of this form of grammar is that it expresses associativity and operator precedence. The existence of long chains of steps which, in themselves, convey no meaning, is a disadvantage of using unmodified syntax trees. Compiler writing tools, generally speaking, eliminate the useless chains by one means or another to give an Abstract Syntax Tree. Here is what the tree would look like for the example.



If there is no semantics associated with parentheses, they too can be eliminated



And, finally, the operators are implicit in the rules, so they too can be eliminated.



The resulting tree is (obviously) much more concise and still contains the information needed to compile code. One disadvantage in the last step of eliminating the operator symbols is that their position in the source text goes with them. This somewhat complicates the formulation of diagnostics involving the operators.