UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS164 **P. N. Hilfinger**
Spring 2011

The Horn Compiler Framework (version 1.0)

HORN[1] is a tool for producing `C++` parsers, lexical analyzers, and abstract-tree generators. It accepts as input a dialect of the BISON parser-generator language, and produces `C++` as indicated here:

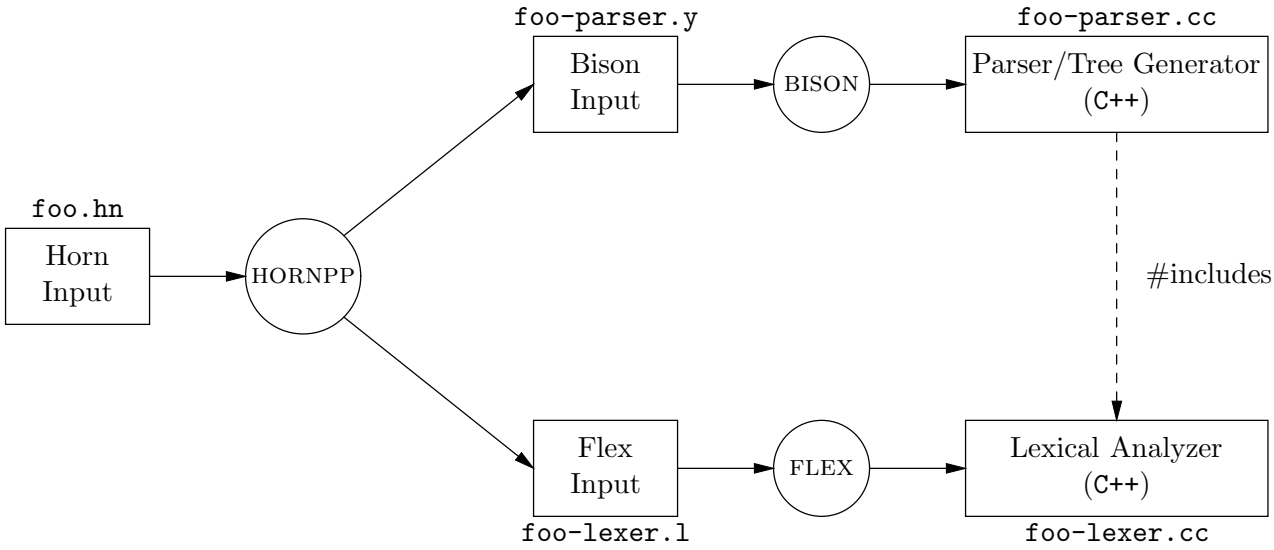

Figure 1: Diagram of how the HORN script processes source files. First, HORNPP, the HORN pre-processor, converts an input file, `foo.hn`, into two input files for the programs BISON and FLEX. The script next invokes these two programs to produce `C++` output files. Compiling the file `foo-parser.cc` produces the parser object or executable file (there is no need to compile `foo-lexer.cc`, because `foo-parser.cc` includes that file in its compilation).

The '`-parser.cc`' file is a "bottom-up" parser (either LR(1), IELR(1), LALR(1), or GLR)—that is, given a grammar rule such as

```
x : a b c ;
```

[1]The name not an acronym, but rather a take-off on ANTLR, a popular compiler framework from across the Bay that is the source of much of Horn's notation.

```
%define semantic_type "type name"

%{
#include statements

Supporting type declarations

Forward declarations of functions

%}

Token and other Bison declarations

%%

Grammar and lexical rules

%%

Definitions of functions, global variables, etc.
```

Figure 2: General layout of a Horn input file.

the program waits until it has processed an `a`, `b`, and `c` before considering whether to apply this rule to produce an `x`. Roughly speaking, top-down parsers (such as ANTLR) *predict* that they will produce an `x` based only on seeing the beginning symbols of an `a`. In theory, bottom-up parsers, since they act on more information, are the more powerful. For example, they are not bothered by left-recursive rules. However, in practice both techniques work well with modern programming languages.

1 Horn input files

An input file to horn has three parts: a prologue, a set of grammar rules, and an epilogue. The prologue declares grammar symbols, types of output values, `C++` entities used in the grammar's actions, and characteristics of the generated parser. The epilogue contains arbitrary `C++` code, such as a main program or functions that use the generated parser and are exported to other modules. The general format is as shown in Figure 2.

2 Basic Grammar and Lexical Rules

The rule sets that HORN and its underlying engine, BISON, handle are called *context-free grammars (CFG).* The notation used is a variety of *Backus-Naur Form (BNF).* Each rule has

the form

$$s_0 : s_1 \ldots s_n;$$

($n \geq 0$), where the s_i are *grammar symbols,* each of which ultimately stands for some set of possible strings of characters, which we'll denote $L(s_i)$, the *language* denoted by s_i. The generic rule pictured here means "The set of strings $L(s_0)$ includes (is a superset of) all those that can be formed by concatenating a string from each of $L(s_1)$, $L(s_2)$, ..., and $L(s_n)$." We refer to s_0 as the *left-hand side* in this particular rule, and $s_1 \ldots s_n$ as the *right-hand side.*

2.1 Context-free grammar

A subset of grammar symbols, called *terminal symbols* (or *terminals* for short) form the base cases in this recursive definition. They are defined by a set of *lexical rules* (described in §2.4), and in HORN, are denoted by identifiers that start with upper-case letters (such as `ID`), by literal strings in double quotes (e.g, `"while"`), or by single-character strings in single quotes (e.g., `';'`). The other grammar symbols, all beginning with lower-case letters in HORN, are called *nonterminal symbols.* One particular nonterminal symbol is called the *start symbol,* conventionally taken to be the symbol defined by the first grammar rule. The language denoted by the start symbol is the language defined by the grammar as a whole. This language is taken to be the *minimal* language that satisfies all the grammar rules[2].

For example, the following grammar describes simple arithmetic expressions (see §2.4 for how we define the terminal symbol `NUM`):

```
expr : term;

expr : expr "+" term;

expr : expr "-" term;

term : factor;

term : term "*" factor;

term : term "/" factor;

factor : NUM;

factor : "(" expr ")";
```

Usually, we abbreviate rules by grouping those for the same left-hand side, like this

[2]This is a typical sort of definition in mathematics. Each individual rule with a given nonterminal s as its left-hand side defines a subset of $L(s)$, but doesn't say what else might be in $L(s)$. So we make this additional provision of minimality, which in effect says that the only strings in $L(s)$ are those that are *required* to be there by some rule.

```
expr : term  |  expr "+" term  | expr "-" term;

term : factor  |  term "*" factor  |  term "/" factor;

factor : NUM |  "(" expr ")";
```

Assuming that we define `NUM` to describe ordinary integer numerals in Java, this grammar describes a language containing such strings as "`2*(3+9)-42`," as you can see from the following *derivation:*

```
expr ⟶ expr - term ⟶ expr - factor ⟶ expr - NUM ⟶ term - NUM
⟶ term * factor - NUM ⟶ term * ( expr ) - NUM
⟶ term * ( expr + term ) - NUM ⟶ term * ( expr + factor ) - NUM
⟶ term * ( expr + NUM ) - NUM ⟶ term * ( term + NUM ) - NUM
⟶ term * ( factor + NUM ) - NUM ⟶ term * ( NUM + NUM ) - NUM
⟶ factor * ( NUM + NUM ) - NUM ⟶ NUM * ( NUM + NUM ) - NUM
```

This derivation consists of a sequence of *sentential forms* (separated by arrows), starting with the start symbol and ending with a character string (once the `NUM`s are replaced by numerals, anyway). At each step we apply one rule, replacing one nonterminal symbol with the right-hand side of one rule for that nonterminal. The parsers generated by Horn and Bison actually perform such derivations in reverse, *reducing* the input to the start symbol.

Here, the language $L($`expr`$)$ is the set of all sentential forms that contain only terminal symbols and that appear at the end of some derivation that starts from `expr`. At each point in a derivation, there is typically more than one possible rule by which to replace any given nonterminal symbol. Any of these rules might be chosen, regardless of what symbols surround the nonterminal—hence the adjective *context-free.* Horn always chooses to apply a rule to the rightmost nonterminal symbol at each stage—a *rightmost derivation.* Since it does so in reverse, we say that it produces *reverse rightmost* (also called *canonical*) derivations.

2.2 The end of file

The Horn system actually inserts its own start symbol into the grammar, effectively defining it like this:

horn_start_symbol `: your_start_symbol` *EOF* `;`

where `EOF` indicates the end of the input (End Of File). The symbols written here in italics are internally generated; you don't have access to them. A lexical rule (§2.4) can return an end of file token by using 0 as the syntactic category (see §5.3), but this is not generally necessary unless you include specific actions in your lexer for end-of-file (see `_EOF` in §2.5).

2.3 Extended BNF

Certain grammatical constructs crop up repeatedly. For example, as part of describing an S-expression in Lisp, we need to describe a sequence of S-expressions[3]:

[3] Horn uses C-style comments, so "`/* empty */`" is ignored. I use it for human readers as a convention for indicating a right-hand side with no symbols—an empty string.

```
sexpr : atom | "(" sexpr_list ")";

sexpr_list : /* empty */ | sexpr_list sexpr;
```

As a shorthand, we can write this instead as

```
sexpr : atom | "(" sexpr* ")";
```

The trailing '`*`' (the *Kleene star*) means "zero or more repetitions of." Similarly, a trailing '`+`', as in

```
stmt_list : stmt+
```

means "one or more repetitions of," and a trailing '`?`', as in

```
relation : expr "not"? "in" expr;
```

means "optional," or "zero or one occurrences of."

HORN also permits grouping using parentheses, as in ordinary algebraic expressions. Thus, instead of

```
expr : expr "+" term | expr "-" term;
```

you may write

```
expr : expr ("+" | "-") term;
```

In combination with the other notations, you can describe even more complex constructs succinctly, such as:

```
argument_list : "(" ( expr ( "," expr )* )? ")";
```

to describe the parenthesized part of a function call.

These extensions to the plain BNF presented in §2.1 give what is called *extended BNF*. All of them can be translated into plain BNF (which, in fact, is how the HORN processor deals with them).

2.4 Lexical rules

Lexical rules define terminal symbols, also known as *(lexical) tokens* or *lexemes*. The HORN script uses an open-source tool called FLEX to produce a program (called a *lexical analyzer*) that processes them, splitting the input text into its constituent tokens and giving these to the parser. Each different kind of token has a unique *syntactic category,* encoded as a non-negative integer. In the `C++` programs it produces, HORN defines the upper-cased terminal symbols used in your grammar as constants that other parts of your program can use.

Lexical rules look very much like ordinary context-free grammar rules that define non-terminal symbols ("CFG rules" from here on), but with a few restrictions and extensions. Lexical rules may not contain nonterminal symbols or other named terminal symbols—just double-quoted strings, single-quoted characters, and *auxiliary lexical symbols,* defined below. Like CFG rules, lexical rules may use parentheses and the operators '`*`', '`+`', '`?`', and '`|`'.

Lexical rules may also use sets of characters. First, the notation

'C_1' .. 'C_n'

is a synonym for

'C_1' | 'C_2' | ⋯ | 'C_n'

where $C_2, \ldots, C_{n-1}$ are all characters between C_1 and C_n in the ASCII collating sequence. Thus,

```
'A' .. 'Z'
```

denotes "any upper-case letter." Ranges and single-quoted characters are the simplest sets of characters. The '`|`' operator, when applied to two sets of characters, yields a set of character. Finally, the operator '`-`', when applied to two sets of characters, yields the set difference between those sets: the set containing all characters in the first operand that are not in the second. For example, the set of lower-case consonants might be denoted

```
'b' .. 'z' - ('e' | 'i' | 'o' | 'u')
```

An *auxiliary lexical symbol* starts with an underscore, and is defined by an *auxiliary lexical rule* having the same form as other lexical rules. These rules have two additional restrictions: an auxiliary lexical symbol must be defined in a single rule, and the right-hand side of an auxiliary lexical rule may only contain auxiliary lexical symbols that are defined *before* that rule. For example, we can define

```
_UpperCase : 'A' .. 'Z';
_LowerCase : 'a' .. 'z';
_Digit : '0' .. '9';
_Letter : _UpperCase | _LowerCase
_Alphanum : _Letter | _Digit
ID : _Letter _Alphanum*
NUM : _Digit+
```

but it would be illegal to put the definition of `_Alphanum` first, since it would then reference auxiliary symbols defined later, and it would be illegal to write a rule such as

```
_Chars : _Char | _Char _Chars
```

since it mentions `_Chars` on the right-hand side, but that is not defined in a *previous* rule.

The collection of all lexical rules (and auxiliary rules) together define a *regular language,* the set of all terminals. This collection is interpreted differently from the CFG rules. There is no one start symbol. Instead, each time a terminal symbol is needed, the lexical analyzer produced by HORN in effect tries each of the lexical grammar rules to see if it matches the beginning of the remaining input text. The analyzer delivers the terminal symbol of whichever rule matches the *longest* prefix of the remaining text, with ties going to the first of the rules matching the most text. For example, consider

```
WITH : "with";
ID : _Letter _Alphanum*;
```

If the remaining input starts with the characters "`withdraw $10`," then both of these rules will match a prefix of the input, but the rule for `ID` matches the longer prefix, so the lexer produces `ID` as the next terminal symbol. It never matters what terminal symbols are allowed by the CFG grammar; the lexical analyzer will try all lexical rules against the remaining input.

With a few exceptions (see §2.2 and §2.6), lexical rules that match the empty string are ignored, in order to guarantee that the lexical analyzer always makes progress. For example, consider

```
NUM: ('0' .. '9')*;
```

If the next input character is something other than a digit, then the definition of '`*`' indicates that this rule can match an empty string, which would be the longest possible match for `NUM` in that case. However, even if no other lexical rule matches, the match for `NUM` will be ignored. Instead, the lexical analyzer will fall back to a last-resort default in which it delivers the next character in the input as a token (the same token denoted by a single-quoted one-character string in rules).

Horn automatically turns terminal symbols represented by strings or character constants in the CFG grammar into lexical rules that precede any lexical rules supplied by the user. Thus,

```
expr : expr "in" expr;
```

becomes something like

```
TOK_3 : "in";
expr : expr TOK3 expr;
```

where `TOK_3` is some automatically generated symbol. You never need to know about these generated symbols. And because the implicit definition of `TOK_3` would come before any (user-written) rule for *ID* (such as that above), the `TOK_3` rule will have precedence (as desired), even though `ID` also matches the same string.

2.5 Special lexical symbols

Several auxiliary symbols are pre-defined. Each matches the empty string, but only under certain circumstances. At the moment, none may be mentioned in an auxiliary rule.

_ANY Is a primitive character set denoting any of the allowable characters (everything other than ASCII NUL ('`\000`') and '`\377`' (an 8-bit character used internally to indicate end-of-input).

_BOL Matches the empty string at the beginning of a line: that is, at the beginning of the file or just after a line terminator sequence). It may only occur as the first symbol in a lexical rule.

_EOL Matches the empty string at the end of a line: that is, immediately before a line terminator sequence (defined as an optional carriage return followed by a newline character).

It does *not* match at the end of file, so if the last line of your input is not properly terminated, you may not get the results you expect. It may only occur as the last symbol in a lexical rule.

Warning: There is a slight glitch here. For the purposes of determining a longest match, _EOL counts as if it matched the newline sequence (i.e., as if it matched a 1- or 2-character string rather than a 0-character string). Usually, this doesn't matter, but it is easy to contrive cases where it does.

_EOF Matches the empty string at the end of file. It must appear at the end of its rule. You will not often need to use this; the HORN lexical analyzer will by default return an end-of-file indication at the appropriate point, and as described in §2.2, the CFG grammar is automatically set up to handle it. `_EOF` is a special case in that any rule it appears in *can* match the empty string. Once `_EOF` matches, it continues to do so until the lexer switches to another input file, so be careful to avoid an infinite loop when using such rules. A common way to do so is to have your rule explicitly return the end-of-file category (0) when you reach real end of file (Horn usually does this for you automatically if you don't provide an explicit rule that matches `_EOF`.)

2.6 Preferred lexical rules and empty matches

Normally, the lexical analyzer returns the longest non-empty match possible from among its rules, preferring the first-appearing rule when there are ties. By including the special declarative symbol `%prefer` at the end of a lexical rule (just before the lexical action, if any), you can specify that a rule should be chosen in preference to rules not so marked, regardless of the length of text matched, and that it may match an empty string. Among preferred rules, the usual precedence rules apply.

As you might guess, this feature is rather specialized. In general, you should rely on HORN's usual rules for precedence. Indeed, the only use I've found so far for `%prefer` is to handle Python's indentation rules:

```
*: _BOL (' ' | '\t')* %prefer { ... }
```

When a preferred rule matches the empty string, no further preferred rules are applied until at least one more token is read using non-preferred rules (to avoid infinite loops in which the lexical analyzer keeps returning empty strings).

3 Grammar Conflicts

HORN parsers belong to a category known as *shift-reduce* parsers. These attempt to reconstruct, in reverse, the sequence of grammar rules needed to derive the input from the start symbol, as described in §2.1. The parser consumes the input and maintains a sequence of grammar symbols (terminals and nonterminals) called the *parsing stack*[4] such that the con-

[4] Abstractly, it is a sequence, but because of the way shift-reduce parsing works, we almost invariably refer to it as a stack, since it is always the most recently added symbols (those at the "top") that are manipulated at each step.

catenation of the parsing stack and the remaining input (as a sequence of tokens) forms one of the sentential forms in a derivation (again, see §2.1).

At each step, the parser either *shifts* a token from the remaining input onto the end (top) of the parsing stack, or it *reduces* zero or more symbols on top of the parsing stack into a nonterminal, using one of the grammar rules. Since multiple grammar rules might seem applicable to the top of the parsing stack, the parser examines the next (unshifted) input token and a summary of the contents of the parsing stack (the *parser state*) to decide what rule (if any) to apply. Sometimes, the choice is unclear, causing HORN (or more precisely, BISON, which does the real work) to report a *grammar conflict,* of which there are two varieties: *shift-reduce* conflicts and *reduce-reduce* conflicts.

3.1 Shift-reduce conflicts

A shift-reduce conflict results when the top of the stack contains the right-hand symbols of some grammar rule (suggesting a reduction), but it might also be valid to shift the next token so as to later get a different reduction. For example, if you were to write

```
expr : expr '-' expr
     | ID
     ;
```

and try to parse an input such as '`a-b-c`,' the parser would eventually find itself in this situation:

```
expr '-' expr ⋈ '-' ID
```

where '⋈' marks the start of the remaining input. At this point, the parser could take either of two routes: either

```
expr ⋈ '-' ID              (Reduce)
expr '-' ID ⋈              (Shift twice)
expr '-' expr ⋈            (Reduce)
expr ⋈                     (Reduce)
```

or else

```
expr '-' expr '-' ID ⋈     (Shift twice)
expr '-' expr '-' expr ⋈   (Reduce)
expr '-' expr ⋈            (Reduce)
expr ⋈                     (Reduce)
```

corresponding to interpreting this expression as either '`(a-b)-c`' or '`a-(b-c)`'. In this example, the conflict results (as it often does) from an essential ambiguity in the grammar. The programmer simply hasn't said which interpretation to choose.

3.2 Reduce-reduce conflicts

A reduce-reduce conflict results when the top symbols of the stack might reasonably be reduced according to either of two different rules. For example, given a grammar containing

```
expr : '(' type ')' expr          (C-style cast)
     | '(' expr ')'               (parenthesized expression)
     | ID
     | ....
     ;
type : ID
     ;
```

and the input '`(a) b`', the parser will eventually see this situation:

```
'(' ID ⋈ ')' ID
```

It might convert `ID` either into a type or an expr. In this case, if it were to look beyond the '`)`', it would see that choosing to reduce to `expr` would not work, but since the parser looks only at the next unshifted token of the input, it does not see this and therefore reports a conflict.

This example notwithstanding, most reduce-reduce conflicts are due to errors in your grammar. You should treat warnings about reduce-reduce conflicts as error messages and resolve them. The parser-generator will arbitrarily resolve these conflicts in favor of the earlier rule, but it is extremely risky to rely on this resolution, since it usually just papers over a real problem. (This is in contrast to lexical analysis, which also resolves conflicts in favor of the earlier rule, but where doing so is usually the right thing.)

3.3 Dealing with shift-reduce conflicts

Sometimes, conflicts result from accidental introduction of ambiguity. For example, there's a good chance you'll eventually make this mistake:

```
expr : expr '+' term
       term                       (Left off the |)
     ;
```

or this one:

```
expr :
     | expr '+' term              (Extra |)
     | term
     ;
```

Either of these can result in a flood of conflicts in the rest of the grammar. All I can say about accidental conflicts is "Try not to introduce them."

Sometimes, however, a conflicted grammar is actually clearer than an unconflicted one, the principal example being expression grammars. You'd like to be able to say

```
expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     ...
```

together with some way of indicating, as in informal English descriptions, that the operators group to the left, with '`*`' having precedence over '`+`' and '`-`'. The usual alternative uses a cascade of definitions, like this:

```
expr : term | expr '+' term | expr '-' term ;
term : factor | term '*' factor ;
  ...
```

(see if you can figure out why this approach avoids ambiguity). This works, but is a bit verbose.

HORN uses a mechanism provided by BISON to allow you to declare precedences for operators, so that an expression grammar can look like this:

```
%left '='
%left '+' '-'
%left '*' '/'
%right "**"
...

%%

...

expr: ID
    | expr '+' expr
    | expr '-' expr
    | expr '=' expr
    | expr "**" expr
    ...
```

Here, '`%left`' and '`%right`' are declarations that go in the prologue of your grammar file. They list operators from lowest to highest precedence, and indicate whether they group to the left or right. Operators in the same declaration have the same precedence.

The idea is pretty simple: HORN assigns each rule the precedence of the operator token it contains (assuming there is only one token given a precedence), tweaking the precedence slightly up if the operator is left associative, and slightly down if it is right associative. Now, consider a conflict like that illustrated in §3.1:

```
expr '-' expr ⋈ '*' ID
```

Either we can reduce the '`expr '-' expr`' or shift the '`*`'. Because the rule has the precedence of '`-`', which is declared to be lower than that of '`*`', shifting wins out here, and the parser will eventually end up reducing the multiplication before reducing the subtraction. With

```
expr '-' expr ⋈ '-' ID
```

since '`-`' has been declared to be left associative, the subtraction rule has (slightly) higher precedence than the '`-`' symbol, and the parser will reduce the first '`-`' first, grouping the first two terms together as desired.

This is all very convenient, but I strongly recommend using this feature *only* for simple operator precedence such as in these examples. The consequences of forcibly "resolving" conflicts that actually indicate problems are surprising and usually undesirable.

3.4 Precedence and extended BNF

Given the facilities in §3.3, it is natural to want to write something like this:

```
expr : expr ('+' | '-' | '*' | '/' | "**" | ...) expr ;
```

but you will quickly find this doesn't work. HORN converts this to some weird-looking rule like[5]:

```
expr : expr __0 expr ;
__0 : '+' | '-' | '*' | '/' | "**" | ... ;
```

Whereas before, the parser would face situations like this:

```
expr '-' expr ⋈ '-' ID
```

where the two operators in question are both available for inspection, with the new grammar, it sees only

```
expr __0 expr ⋈ '-' ID
```

and the identity of the left operator is lost.

Fortunately, there is a convenient, if moderately obscure feature that addresses just this problem. We can write our rule as follows:

```
expr : expr ('+' | '-' | '*' | '/' | "**" | ...) expr %expand
```

The effect of the `%expand` directive is to convert this rule differently, so that it reads

```
expr : expr '+' expr | expr '-' expr | expr '*' expr | ... ;
```

In this form, precedence rules work properly. You get to write the more concise rule and have it expanded for you into the long-winded form.

[5]Symbols such as `__0` are internally generated, and are not lexical symbols.

3.5 GLR parsing

Sometimes, as in the example from §3.2, a conflict results from the fact that the parser is required to make a decision before it has all the necessary information. You can generally resolve this with judicious rewriting, but it is sometimes clearer to use "brute force." HORN provides an alternative parsing algorithm called *Generalized LR (GLR)*[6]. When confronted with a conflict at parsing time, the GLR parser will (in effect) split into multiple parsers, each pursuing a different choice of shifts and reductions. As some of these choices turn out to be unfeasible, their parsers die off. Assuming that only one parser makes it to the end, all is well. While the parser is split, it does not execute any actions, but instead saves them up until the surviving parse is determined.

For example, going back to the example from §3.2:

```
expr : '(' type ')' expr          (C-style cast)
     | '(' expr ')'               (parenthesized expression)
     | ID
     ;
type : ID
     ;
```

HORN will report that there is a reduce-reduce conflict when the parser has just shifted '`(`' and '`ID`' and is looking at '`)`'. If the parser were to look one symbol beyond the '`)`', it would know which reduction would work. For this grammar, including the declaration

```
%glr-parser
```

in the prologue will cause the parser to pursue both possibilities, one of which will get pruned.

This is a very powerful mechanism (and not fully described here). However, there is one problem: HORN will still report conflicts in the grammar, since it cannot in general analyze whether the parser is guaranteed to accept only one parse. You will have to analyze your grammar carefully (and test it extensively) in order to make sure you are getting the proper results.

4 Semantic Actions

So far, we've been concerned entirely with syntax. The HORN parsers illustrated so far will read an input text and either determine that it obeys the grammar rules and do nothing, or determine that it does not obey the grammar rules and produce an error message. The main point of defining a grammar and breaking it down into rules is to implement *syntax-directed translation* of the input, in which the particular derivation (sequence of rules) used to parse an input triggers a corresponding sequence of actions that translates or otherwise processes the text. In HORN, these actions take the form of arbitrary `C++` code enclosed in curly braces and placed at the end of a rule. Being arbitrary `C++` code, it can do anything. For example, given the HORN program:

[6] "LR" is the name of HORN's standard parsing algorithm. The initials stand for "Left-to-right (reverse) Rightmost derivation."

```
expr : term              { printf ("term <- expr\n"); };
expr : expr "+" term     { printf ("expr + term <- expr\n"); };
expr : expr "-" term     { printf ("expr - term <- expr\n"); };
term : factor            { printf ("factor <- term\n"); };
term : term "*" factor   { printf ("term * factor <- term\n"); };
term : term "/" factor   { printf ("term / factor <- term\n"); };
factor : NUM              { printf ("NUM <- factor\n"); };
factor : "(" expr ")"    { printf ("( expr ) <- factor\n"); };
```

and the input string "`2*(3+9)-42`," we get the following output from the compiled and executed program:

```
NUM <- factor
factor <- term
NUM <- factor
factor <- term
term <- expr
NUM <- factor
factor <- term
expr + term <- expr
( expr ) <- factor
term * factor <- term
term <- expr
NUM <- factor
factor <- term
expr - term <- expr
```

Follow this output from bottom to top and compare it to the sample derivation of the same string in §2.1. You should see that each output line shows what changed between one derivation step and the next.

4.1 Semantic Values

Such pure side-effect-producing programs are not the usual case. More commonly, semantic actions are mostly concerned with computing *semantic values* for nonterminal symbols. Consider two steps from the derivation of `2*(3+9)-42` in our running example (see §2.1):

$$\cdots \longrightarrow \texttt{term - NUM} \longrightarrow \texttt{term * factor - NUM} \longrightarrow \cdots$$

First, let's reverse their order to match the order in which they actually get processed:

$$\cdots \longleftarrow \texttt{term * factor - NUM} \longleftarrow \texttt{term - NUM} \longleftarrow \cdots$$

The `term*factor` part corresponds to '2*(3+9)' in the input. Suppose we attach to each of these symbols the numeric value of the numeric expression it represents:

$$\cdots \longleftarrow \texttt{term}_2 \texttt{ * factor}_{12} \texttt{ - NUM}_{42} \longleftarrow \texttt{term}_{24} \texttt{ - NUM}_{42} \longleftarrow \cdots$$

The subscripts are the semantic values attached to the instances of the grammar symbols in this example. We can write the action for the rule that specifies this derivation step so that it computes the value for the newly reduced `term`, like this:

```
term : term "*" factor   { $$ = $term.value() * $factor.value(); }
```

In this action, we refer to the grammar symbols in the rule using the '`$`' notation, which the HORN preprocessor will convert to the (rather arcane) expressions that actually access those symbols. We can refer to a symbol's value by its name (as in `$factor`) if it is unique. Alternatively, we can attach labels to the symbols and refer to those:

```
term : L=term "*" R=factor { $$ = $L.value() * $R.value(); }
```

This latter notation is necessary when the given grammar symbol is used multiple times in a right-hand side.

These '`$`·' symbols have various operations defined on them, one of which is `.value()`, a method that extracts a semantic value of some user-defined type. Assigning a semantic value to the left-hand side symbol (as in "`$$ =`") effectively defines the semantic value of that symbol. Filling out the example:

```
expr : term;
expr : expr "+" term     { $$ = $expr.value() + $term.value(); };
expr : expr "-" term     { $$ = $expr.value() - $term.value(); };
term : factor;
term : term "*" factor   { $$ = $term.value() * $factor.value(); };
term : term "/" factor   { $$ = $term.value() / $factor.value(); };
factor : NUM;
factor : "(" expr ")"    { $$ = $expr; };
```

When we don't provide an action, as in three of the rules above, the default action copies the value of the first right-hand symbol, as if we had written:

```
expr : term                  { $$ = $term; };
```

4.2 Inner actions

Occasionally, it is useful to take an action part way through collecting the symbols on the right-hand side of a rule. This is not often necessary, but is sometimes useful in cases where global variables affect subsequent processing. Roughly, a rule such as

```
init : "{" { initializing=true; } init_list "}" ;
```

is shorthand for

```
init : "{" gensym_1 init_list "}" ;
gensym_1 : /* empty */ { initializing = true; } ;
```

where `gensym_1` represents a new, unique, automatically generated symbol. I said "roughly" because these inner rules *are* allowed to reference the semantic values of preceding grammar symbols (but not `$$`), as in

```
values : expr { collect ($expr); } "=" expr;
```

even though "`$expr`" would not be defined in the compiler-generated rule.

Because inner actions introduce a new rule that the parser has to reduce before seeing the subsequent parts of the right-hand side, they can introduce conflicts, where the parser does not have enough information immediately available to decide what alternative path to follow (see §3).

4.3 Collecting actions

Especially when you are using extended BNF, you will need to collect lists of semantic values. Let's go back to a previous example:

```
sexpr : atom | "(" sexpr* ")";
```

If you add actions to this rule, you can reference `$sexpr` easily enough, but the question becomes "which of the sequence of zero or more `sexprs` did you mean?" Fortunately, we have a solution, illustrated here:

```
sexpr : atom                 { $$ = process($atom.value()); }
      | "(" (L+=sexpr)* ")"  { $$ = process($L.list_value()); }
```

(the parentheses around '`L+=...`' aren't actually necessary, but they make the meaning clearer). The '`+=`' notation tells HORN to add each `sexpr`'s value into a list of some user-selected type (such as `list<`*T*`>*`, where `list` is the generic list type in the `C++` Standard Template Library (STL) and *T* is the type of semantic values).

4.4 Methods on grammar symbols

As you've seen in previous sections, the objects represented by quantities such as `$atom` can contain semantic values or lists of values. They also carry other information, which you can access by means of additional methods. Here is the list:

.value() The semantic value of this symbol, if it is a simple value as opposed to a list. Yields the default value if the value is missing.

.list_value() The value of this symbol as a list of semantic values. Yields an empty list if the value is missing.

.missing() True if the semantic value of this symbol is missing (which happens in cases such as these:

```
primary : atom suffix?                  { ... }
secondary : (atom suffix | prefix atom) { ... }
```

In the first case, `$suffix.missing()` will be true if the optional suffix is not present. In the second, either `$suffix.missing()` or `$prefix.missing()` will be true depending on which alternative applies.)

.text() The source text associated with this symbol as a C++ string. Generally, this is empty for symbols other than tokens (lexical symbols), although the programmer can arrange to associate a text value with all semantic values.

.c_text() The source text associated with this symbol as a C `const char*` pointer. Unlike most C strings, however, this pointer is not NUL terminated (use `.text_size()` to get its length). Generally, this is NULL for symbols other than tokens (lexical symbols), although the programmer can arrange to associate a text value with all semantic values.

.text_size() The length of text in `.c_text()`.

.loc() The location of this symbol (its type is `const char*`, but that should be immaterial; it is intended for use with `yyprinted\_location`, `yylocation_line`, and `yylocation_source`.) See also §8.

.set_loc(L**)** Set the location (the value of `.loc()`) associated with this symbol to L. If semantic values of symbols carry locations, this will also set the location of the semantic value of this symbol. See also §8.

5 Lexical Actions

Lexical rules can also have actions, but they differ considerably from actions on CFG rules. For one thing, they are much more limited: inner actions are not allowed; and a lexical action may not reference the values of the individual right-hand side items—only the complete text matched by the rule. Within a lexical action, the symbol `$TEXT` is a `char*` pointer to the text matched by the rule and `$TEXT_SIZE` is the length of this text. When you compute a semantic value to attach to the token produced by a lexical rule, you can return it as you do for CFG rules:

`$$ =` *semantic value for token*`;`

For example, if your semantic values are integers, you might need a rule like this for decimal literals[7]:

```
NUM : ("-" | "+")? ('0' .. '9')+    { $$ = atoi($TEXT); }
```

By default, the HORN framework will set `$$` if you do not, using a user-supplied function. For values other than trees, this will be a function with the header:

semantic_value_type `make_token (int syntax, const char* text, size_t len);`

where `syntax` is the syntactic category of the token (e.g., `NUM` in the last example).

The values that `$TEXT` takes on are persistent: you may safely store them and expect that the characters they point at will not change. However, although within the text of a lexical action, the string is NUL terminated (as per the standard C convention), it need not be so terminated later, so if you need to keep the text around, you will need to either copy the characters into a NUL-terminated string or `C++ string`, or keep its length around as well.

[7] Actually, most such rules won't allow a sign in order to avoid conflicts with unary negation, for example, but I thought I'd take the opportunity to illustrate the '`?`' operator.

5.1 Specifying actions for implicit tokens

In context-free rules, one normally indicates a literal token (such as a keyword or punctuation mark) with a quoted string. HORN generates lexical rules for these without your having to write anything, and normally generates an appropriate lexical action. You can specify explicit lexical actions for these symbols by using them on the left side of a lexical rule whose right side consists of a single lexical action. For example,

```
"(" : { bracket_count += 1; }
    ;
```

increments a variable once for each left parenthesis. The actual pattern matched by this rule is always the same as the left-hand side; you never actually write it.

5.2 Ignoring tokens

In many cases, the parser would just as soon not see some of the text. For example, in most programming languages, whitespace (blanks, tabs, and sometimes line terminators) take no part in the grammar of a language and would be a nuisance to deal with there. Similarly for comments. The HORN system provides a way to specify tokens that should be ignored, and never seen by the parser. To do this, simply include a `YYIGNORE` statement (it's actually a macro) in the lexical action. A typical example:

```
WS : (' ' | '\t' | '\n' | '\r' | '\f')+   { YYIGNORE; }
```

The generated lexical analyzer will skip all `WS` tokens and will suppress the default creation of a semantic value for them. These tokens will still serve to delimit other tokens (such as identifiers and keywords), as usually required in most applications.

5.3 Explicit syntactic categories

As indicated in §2.4, the parser (outside of actions) depends only on the syntactic categories of the tokens that the lexical analyzer feeds to it. In HORN, these categories are represented as integers. By default, the syntactic category returned by a rule is that named on its left side, but there are cases where it is more convenient to decide on a category in lexical actions. The statement

`YYSET_TOKEN(`*category*`);`

does just this. For example, we could write a rule like:

```
UPPER_ID : ('A' .. 'Z') _Alphanum* ;
LOWER_ID : ('a' .. 'z') _Alphanum* ;
```

or like this:

```
UPPER_ID : _Letter _Alphanum* {
    if (islower ($TEXT[0])) YYSET_TOKEN(LOWER_ID); }
```

Of course, it is a little confusing for the reader to have the syntactic category returned by a rule differ from that on the left-hand side like this, so we also allow rules with no specified syntactic category:

```
* : _Letter _Alphanum* {
    YYSET_TOKEN(islower ($TEXT[0]) ? LOWER_ID : UPPER_ID); }
```

In the absence of `YYSET_TOKEN`, these rules are ignored, so we could also rewrite the whitespace rule as

```
* : (' ' | '\t' | '\n' | '\r' | '\f')+
```

5.4 Declaring syntactic categories

When using `YYSET_TOKEN`, you must be careful that the names you use as syntactic categories are defined. HORN does this automatically for names that appear on the left sides of lexical rules, but not for other names you might want to use. However, you can introduce new names by means of a *token declaration,* which appears in the prologue mentioned in §1. For example:

```
%token UPPER_ID LOWER_ID
```

introduces the syntactic categories in the example above without requiring that you use them on the left-hand side of a lexical rule. New syntactic categories are particularly useful when used with the HORN tree-building framework (see §7.1 and §7.3), which uses them to identify types of tree nodes.

You can also attach symbolic names to tokens denoted by string literals. For example,

```
%token EXPO "**"
```

Allows you to use the name EXPO in program text to name the syntactic category associated with the '`**`' token (which would otherwise be anonymous).

6 Defining Semantic Types

In order to use the `.value()` and `.list_value()` methods (see §4.1), you must inform HORN what types of value they return and provide some information about these types. The simplest declaration is just

```
%define semantic_type "Type"
```

which indicates the type of semantic values and creates a list type for use with '`+=`' operators. One may supply any POD type[8] for *Type*, with the result that the expression `$`*X*`.value()` will yield values of that type and lists returned by `$`*X*`.list_value()` will yield a type derived from the standard `C++` library type `list<`*Type*`>`.

To get the operations required by the tree-building features described in §7, use the declaration

[8] POD stands for "Plain Old Data" and refers to standard C types, in particular excluding types with constructors or destructors. The standard collection types in the `C++` library, in particular are *not* POD types. However, since pointers are POD types, you can generally get anything you want for a semantic type by using a level of indirection.

`%define semantic_tree_type` *YOUR_TREE_TYPE*

in place of `%define semantic_type`.

Figure 3 shows a fleshed-out example.

6.1 Automatic storage deallocation

`C++` does not require garbage collection of dynamically allocated storage (i.e., storage allocated using the **new** operator)—indeed, several features of the language make automatic garbage collection quite difficult. The HORN framework provides a limited amount of garbage collection, but requires cooperation from the programmer.

The underlying technique is known as *reference counting.* The idea is that instead of using pointers to objects it controls, the framework uses a kind of augmented pointer: rather than `T*`, it uses `Pointer<T>`. Thanks to the definitional facilities of `C++`, you can (largely) use this type as a direct substitute for `T*`; if `x` has type `Pointer<T>`, then `x->aField`, `x->aMethod()`, `*x`, `x = new T(...)`, and `x == NULL` all mean the same things as in vanilla `C++`. However, behind the scenes, the framework keeps track of the total number of `Pointer<T>` values that reference any give `T` object, automatically modifying this count when new `Pointer` values get created, assigned, or deallocated. When the count reaches 0, the object is deleted. The pointed-to objects will typically have destructors defined for them whose effect is to release all instance variables in the object, including those of type `Pointer<...>`, so that, for example, releasing the last pointer to a reference-counted list releases the entire list. Together, these features will delete non-circular structures as they become unreachable.

You must be careful to take certain precautions: in particular, you must be careful never to deallocate all `Pointer<T>` to an object while still manipulating that object by means of ordinary pointers. Fortunately, you aren't normally tempted to do so.

7 Building Abstract Syntax Trees

One very common application of parser frameworks is the production of *abstract syntax trees* (ASTs), which are essentially tree representations of a program that elide certain syntactic or lexical details. HORN includes a set of notations that allow you to specify transformations from textual representations of programs to ASTs, and provides some basic AST classes that you can extend to suit your application.

This framework provides trees in which each node is labeled by a token and has an arbitrary number of children. For example, consider again a language of arithmetic expressions, and suppose that the translation we're after takes each expression, $E = E_1 \oplus E_2$ (where '$\oplus$' is a binary operator) and produces a tree, $T(E)$, labeled with the token for $\oplus$ and having two children representing the translations of E_1 and E_2 (or in Lisp-like prefix notation, $(\oplus\ T(E_1)\ T(E_2))$). We could re-work the calculator example in Figure 3 to do this by modifying the actions:

```
expr : L=expr op="+" R=expr { $$ = make_tree ($op.value(), $L.value(), $R.value(); };
expr : L=expr op="-" R=expr { $$ = make_tree ($op.value(), $L.value(), $R.value(); };
expr : L=expr op="*" R=expr { $$ = make_tree ($op.value(), $L.value(), $R.value(); };
```

```
%{
#  include <cstdlib>
#  include <cstdio>
#  include <math.h>

   extern double make_token (int syntax, const char* text, size_t len);

   using namespace std;

%}

%define semantic_type double
%interactive

%left "+" "-"
%left "*" "/"
%right "**"

%%

prog : (expr ";" { printf ("=%g\n", $expr.value()); })* ;

expr : L=expr "+" R=expr     { $$ = $L.value() + $R.value(); };
expr : L=expr "-" R=expr     { $$ = $L.value() - $R.value(); };
expr : L=expr "*" R=expr     { $$ = $L.value() * $R.value(); };
expr : L=expr "/" R=expr     { $$ = $L.value() / $R.value(); };
expr : L=expr "**" R=expr    { $$ = pow($L.value(), $R.value()); };
expr : NUM;
expr : "(" expr ")"          { $$ = $expr; };

_DIG : '0' .. '9' ;
NUM : _DIG+ ("." _DIG*)? (("e"|"E") ("+"|"-")? _DIG+)? ;
* : ' ' | '\t' | '\n' | '\r';
%%

double
make_token (int syntax, const char* text, size_t len) {
   return strtod (text, NULL);
}

main () {
  yypush_lexer (stdin, "<stdin>");
  yyparse ();
}
```

Figure 3: Full calculator example, showing specification of a simple domain of semantic value (in this case, `double`).

```
expr : L=expr op="/" R=expr { $$ = make_tree ($op.value(), $L.value(), $R.value(); };
expr : L=expr op="**" R=expr { $$ = make_tree ($op.value(), $L.value(), $R.value(); };
expr : NUM;
expr : "(" expr ")"         { $$ = $expr; };

_DIG : '0' .. '9' ;
NUM : _DIG+ ("." _DIG*)? (("e"|"E") ("+"|"-")? _DIG+)?
```

As you can see, this leads to a rather tedious and repetitive definition. You can be considerably more clear and concise by using HORN's tree-forming operators, which allow the following specification:

```
%right "**"
%left "*" "/"
%left "+" "-"

%token EXPO "**"
%%

expr : expr "+"^ expr;
expr : expr "-"^ expr;
expr : expr "*"^ expr;
expr : expr "/"^ expr;
expr : expr "**"^ expr;
expr : NUM;
expr : "("! expr ")"!;

_DIG : '0' .. '9' ;
NUM : _DIG+ ("." _DIG*)? (("e"|"E") ("+"|"-")? _DIG+)?;
```

This produces the same definition as before. The '`^`' symbols mark the operators, and the '`!`' symbols mark tokens that are to be ignored and not included in the tree. All defaulted lexical rules that are supposed to return tokens use a call to a `make_token` operator, as in the previous version. (We've also defined the symbolic name `EXPO` as a synonym for the '`**`' token. We won't really need it, however, until §7.3.)

More precisely, consider a general grammar rule of the form

$$x0 : a_1 \ \cdots \ a_k \ \ b_1\texttt{\^{}} \ \ a_{k+1} \ \cdots \ a_{k'} \ \ b_2\texttt{\^{}} \ \ a_{k'+1} \ \cdots;$$

where all the a_i and b_i are grammar symbols. We eliminate any symbols followed by `!`, and then proceed from left to right, adding the value of each a_i to the "current node". Initially, the current node is a special kind of tree node that acts as a list (it has a null operator), so that in the absence of any b_j`^` clauses, the default action will just produce a list of the values of the a_i. Each time a b_j`^` is encountered, the framework creates a new node with b_j as its operator and the current node as its child. This new node now becomes the current node.

Adding a list node, L, as a child of another node, N, "unpacks" L; that is, its children become the (direct) children of N, so that lists *per se* are never children of other nodes (including other lists). This is similar to Perl, in which there are no lists of lists, since lists are always flattened into single-level structures. Therefore, a rule such as

```
thing : ID ID  "<>"^ NUM NUM
```

gives trees of the form

```
("<>" ID ID NUM NUM)
```

rather than something like

```
("<>" (ID ID) NUM NUM)
```

Likewise, the rules

```
thing : ids "<>"^ nums ;
ids : ID ID ;
nums : NUM NUM ;
```

yield the same trees as the first form (`ids` yields a list of two `ID` nodes, since there is no `^` operator present.

When combined with extended BNF operators, you can get some nice effects. For example,

```
arg_list : "("! (expr (","! expr)*)? ")"! ;
```

turns input "`(e1, e2, e3)`" into a list of three expression trees, discarding the commas and parentheses. The same rule matches input "`()`," yielding an empty list. As another example,

```
expr : NUM (op^ NUM)+ ;
op : "+" | "-" ;
```

would yield a left-associated tree such as

```
(+ (- (+ NUM NUM) NUM) NUM)
```

from input text "`NUM + NUM - NUM + NUM`."

7.1 Explicit tree formation

Sometimes, the convenient and concise tree-formation operators '`^`' doesn't quite fit the grammar. For example, to translate a function call with a syntax such as

```
expr : expr "("! arg_list ")"!
```

you'll most likely want an operator with a name such as `CALL`, defined with

```
%token CALL
```

in the prelude (see §5.4). (You *could* instead use '`(`' as an operator, as in

```
expr : expr "("^ arg_list ")"!    /* ?? */
```

but this seems a bit artificial.) There's nothing for it but to set `$$` explicitly. Fortunately, there are a few shortcuts. In actions, the symbol `$^` is shorthand for the name of the tree-forming function. Its first argument, the operator, can either be a token from the right-hand side of the rule, or it can be the name of a terminal symbol from the grammar. So, for example,

```
expr : expr "("! arg_list ")"!   { $$ = $^(CALL, $expr, $arg_list); }
```

For even more brevity, you can refer to the entire list of tree operands (if there is at least one) with '`$*`':

```
expr : expr "("! arg_list ")"!   { $$ = $^(CALL, $*); }
```

7.2 Defining tree types

The HORN framework includes a generic tree type that serves as the base class of user-defined trees. This provides for simple tree formation, and for accessors for children and operators. Any particular tree type used in your program will be derived from the generic type, and will add whatever additional methods and other members needed for your application. The simplest possible definition, giving only the basics, looks like this:

```
%define semantic_tree_type Node

%{
class Token;
class Tree;

class Node : public CommonNode<Node, Token, Tree> {
   /* The predefined class CommonNode defines the type NodePtr as a
    * synonym for Pointer<Node>.  CommonNodes (and  thus Nodes) are
    * reference counted (see §6.1).  */
};

class Tree : public CommonTree<Node, Token, Tree> {
public:
   /** An internal node with operator OPER (which must be a token),
     *  and the children between iterators BEGIN (inclusive) and END
     *  (exclusive). */
    template <class InputIterator>
    Tree (Node::NodePtr oper, InputIterator begin, InputIterator end)
        : CommonTree<Node, Token, Tree>(oper, begin, end) { }

};
```

```
class Token : public CommonToken<Node, Token, Tree> {
public:
    Token (int syntax, const char* text, size_t len, bool owner = false)
        : CommonToken<Node, Token, Tree>
            (syntax, text, len, owner) { }
    Token (int syntax, const std::string& text, bool owner)
        : CommonToken<Node, Token, Tree>
            (syntax, text, owner) { }
};
%}
```

The rather convoluted definitions of `Node`, `Tree`, and `Token` address a problem with the static typing of `C++`. First, we want to have a common type that defines operations on all tree nodes, with two derived types covering tokens (a type of leaf) and inner nodes. So far, so easy: we just define

```
class CommonNode {
    ...
    CommonNode* child (int k) const { ... }
    ...
};
class CommonToken : public CommonNode { ... }
class CommonTree : public CommonNode { ... }
```

Unfortunately, what we really want is for the user to be able to extend these three types. However, when you derive `YourNode` from `CommonNode`, the new type is no longer a supertype of `CommonToken` and `CommonTree`, so that types you derive from those latter two types will not be subtypes of `YourNode`. Therefore, we define our base node types as taking the types you want to define as parameters. The real definitions look more like this:

```
template <class YourNode, class YourToken, class YourTree>
class CommonNode {
public:
    ...
    virtual YourNode* child (int k) const { ... }
    ...
};

template <class YourNode, class YourToken, class YourTree>
class CommonToken : public YourNode {
   ...
};

template <class YourNode, class YourToken, class YourTree>
class CommonTree : public YourNode {
```

```
    ...
};
```

It looks strange, but when these are instantiated (so that `Node`, is substituted for `YourNode`, `Token` for `YourToken`, and `Tree` for `YourTree`), the subtyping relations will all be right: `Token` and `Tree` will be subtypes of `Node`, as desired.

7.3 Node Factories

One common pattern used in compilers and other language processors assigns a subtype of the tree type to each different kind (or "*phylum*") of AST—one for `if` statements, one for function calls, etc. By defining appropriate virtual methods in the base node type, you can then customize the behavior of each type of node—say by having a different overriding of a code-generating method for each.

The HORN framework helps out here by providing a static node *factory* method that allows the framework to decide what type of node to create depending on the syntactic category of the operator. By putting the appropriate boilerplate into an AST class, you can get the framework to generate an instance of it for each instance of a given operator.

Let's consider again the arithmetic-expression example from §7, which had the operators

```
"+" "-" "*" "/" "**"
```

We'll give our AST nodes an `eval` method, which yields the integer value denoted by that tree (performing whatever its operator is supposed to do on the values of its operands). Figure 4 shows the definition of the parent node, token, and tree types.

Now we can define separate classes for each of the operators. Here's addition:

```
class Add_Tree : public Arith_Tree {
public:
    int eval() {
        return child(0)->eval() + child(1)->eval();
    }

protected:

    Add_Tree* make (const Arith_Node::NodePtr& oper,
                    const Arith_Node::iterator& begin,
                    const Arith_Node::iterator& end) {
        return new Type (oper, begin, end);
    }

    Add_Tree(const Arith_Node::NodePtr& oper,
                    const Arith_Node::iterator& begin,
                    const Arith_Node::iterator& end)
            : Arith_Tree(oper, begin, end) { }
```

```
class Arith_Token;
class Arith_Tree;

class Arith_Token;
class Arith_Tree;

class Arith_Node : public CommonNode<Arith_Node, Arith_Token, Arith_Tree> {
public:
    virtual int eval () { return 0; }
};

class Arith_Tree : public CommonTree<Arith_Node, Arith_Token, Arith_Tree> {
protected:
    template <class InputIterator>
    Arith_Tree (const Arith_Tree::NodePtr& oper,
                InputIterator begin, InputIterator end)
          : CommonTree<Arith_Node, Arith_Token, Arith_Tree (oper,  begin, end)
    { }
    /** Factory constructor.  See text. */
    Arith_Tree (int syntax)
          : CommonTree<Arith_Node, Arith_Token, Arith_Tree> (syntax) { }

};

class Arith_Token : public CommonToken<Arith_Node, Arith_Token, Arith_Tree> {
public:
    Arith_Token (int syntax, const char* text, size_t len, bool owner = false)
        : CommonToken<Arith_Node, Arith_Token, Arith_Tree>
            (syntax, text, len, owner),
          _value (atoi(string(text, len).c_str()))
    { }

    Arith_Token (int syntax, const std::string& text, bool owner)
        : CommonToken<Arith_Node, Arith_Token, Arith_Tree> (syntax, text, owner) { }

    int eval () { return _value; }

private:
    int _value;
};
```

Figure 4: Parent classes for arithmetic ASTs (unabbreviated.)

```
    /** Use for factory only. */
    Add_Tree() : Arith_Tree('+') { }
    static const Add_Tree factory;
};

const Add_Tree Add_Tree::factory;
```

That's about it. The declaration of `Add_Tree::factory` (which cannot be referenced outside the `Add_Tree class`) is a C++ trick that calls the one-argument constructor defined by the `CommonTree` template class before the main program gets executed. This in turn causes the factory variable to get stored in a mapping between syntactic categories and factory nodes. The `make` method overrides a virtual `make` method in the `CommonTree` template class. To create a new node whose operator has the syntactic category '+', the HORN framework first looks up the factory for `Add_Tree` in a table indexed by syntactic category, and then calls the `make` method on that factory, which, as you see, then calls the constructor for `Add_Tree`.

For single-character tokens like `"+"`, the framework simply uses the ASCII character value as the syntactic category. For others, you'll need to use (and define) symbolic names with `%token` declarations.

7.4 Useful Abbreviations

The definitions in this section are involved and, when repeated many times for each subclass of tree, tend to clutter one's source programs. Therefore, the HORN framework provides some abbreviating macros. Figure 5 gives an abbreviated version of Figure 4 and of `Add_Tree`. The confusion of constructors is reduced to a few lines for each class. The macro `NODE_BASE_CONSTRUCTORS` defines the necessary constructors for a base class—one that is not instantiated. `NODE_CONSTRUCTORS` defines the necessary constructors and declares the factory variable for a node that will be instantiated, and `NODE_FACTORY` defines the actual factory variable.

Occasionally, you'll need to have your node constructors perform additional initialization. In the general case, you'll have to forego using the macros and write the full definitions (as shown in Figure 4). However, if the definitions are simple initializers, the HORN framework provides a couple of macros to do the job.

For example, suppose that you have a tree subclass that has two additional integer fields that must be initialized to 0. In expanded form, such a class would look like this:

```
class Fancy_Tree : public Arith_Tree {
private:
    int writes, reads;

protected:

    Fancy_Tree* make (const Arith_Node::NodePtr& oper,
                      const Arith_Node::iterator& begin,
```

```
class Arith_Token;
class Arith_Tree;

class Arith_Token;
class Arith_Tree;

class Arith_Node : public CommonNode<Arith_Node, Arith_Token, Arith_Tree> {
public:
    virtual int eval () { return 0; }
};

typedef CommonTree<Arith_Node, Arith_Token, Arith_Tree> Arith_Tree_Parent;

class Arith_Tree : public Arith_Tree_Parent {
protected:
      NODE_BASE_CONSTRUCTORS (Arith_Tree, Arith_Tree_Parent);
};

class Arith_Token ... /* as in Figure 4 */

class Add_Tree : public Arith_Tree {
public:
    int eval() {
        return child(0)->eval() + child(1)->eval();
    }

protected:

    NODE_CONSTRUCTORS (Add_Tree, Arith_Tree);
};

NODE_FACTORY (Add_Tree, '+');
```

Figure 5: Abbreviated version of arithmetic trees.

```
                        const Arith_Node::iterator& end)
    {
        return new Type (oper, begin, end);
    }

    Add_Tree(const Arith_Node::NodePtr& oper,
                    const Arith_Node::iterator& begin,
                    const Arith_Node::iterator& end)
            : Arith_Tree(oper, begin, end), writes(0), reads(0) {
    }

    ... etc.

};
```

The abbreviated version is

```
class Fancy_Tree : public Arith_Tree {
private:
    int writes, reads;

protected:

    NODE_CONSTRUCTORS_INIT (Fancy_Tree, Arith_Tree, writes(0), reads(0));
};
```

There is likewise a `NODE_BASE_CONSTRUCTORS_INIT` for base types.

8 Source Locations

When you push a file or string into a HORN lexer, it will keep track of the correspondence between the lexeme text it returns (in the form of C `char*` pointers) and positions (line numbers) that the text came from, relative to the file or string that contained it. The function `yyprinted_location`(P) (see §4.4) will convert a text pointer, P, into a string of the form $F{:}L$, where F is the supplied to `yypush_lexer` for the file or string that contains P, and L is the line number within that file or string. The functions `yylocation_line` and `yylocation_source` break out L and P individually. Thus, these `char*` pointers double as source locations.

During the parse, the function `yysource_location()` returns the lexer's current position, which is generally somewhere *after* that of the last token it found. Each terminal symbol in a rule stores its source position, which you may access using the `.loc()` method, as in `$ID.loc()`. Nonterminal nodes don't automatically track source locations and by default `.loc()` will return NULL (the unknown location) when applied to them. However, if your semantic values do contain locations (see below), then `.loc()` will work on nonterminals as well.

Semantic values may carry location information as well. In particular, the standard tree-building routines supplied in the HORN framework do so: if `x` is a node (token or tree), then `x->loc()` is its location and `x->set_loc(L)` allows you to change the location it stores. In the absence of `set_loc` operations upon it, a tree node will report its location as that of the first child that has a known location (or `NULL` if none does).

When semantic values carry locations, the operation `.loc()` on grammar symbols will consult that location and `.set_loc(`*L*`)` will set both the location maintained in the grammar symbol, but also that of the semantic value.

9 The Prologue

Throughout this document we've introduced a number of items that may appear in the prologue of a HORN program—the part preceding the first `%%` separator line. This section consolidates them for easier reference.

The BISON engine that underlies HORN supports a large number of prologue directives and declarations. For expendience, HORN just passes most of these through at the moment, but to be honest, their interactions with the HORN framework are untested and potentially problematic. It is probably best to stick to the features described here.

9.1 Inserting code

Actions in the grammar are general `C++` source text. Any functions, global variables, or types that they refer to must be defined in the prologue. You can insert arbitrary `C++` code before the grammar section by enclosing it in the delimiters '`%{`' and '`%}`', as in

```
%{
#include <iostream>

using namespace std;

static bool need_postprocessing;

static void eval (const char* expr);

%}
```

This code will appear in the midst of framework definitions generated by HORN itself. To specify that it appear as early as possible (seldom necessary, but see §9.2), use

```
%code top {
C++ code
}
```

9.2 Namespaces

Especially when you need more than one parser in your program, it is convenient encapsulate each in a `C++` *namespace* so that the global names used in each do not conflict. The declaration

```
%define api.namespace "name"
```

does this, enclosing the entire parser and lexer in

```
namespace name {
    ⋮
};
```

If you do this, you will need to put `#include` directives for all headers used in the parser that do not define names in the parser namespace in a '`%code top`' region so as to come before the namespace declaration. It doesn't matter if this results in redundant `#includes`, assuming that (like the system headers), all header files follow the C/`C++` convention of protecting their contents using conditional compilation:

```
#ifndef _THISHEADERFILENAME_H
#define _THISHEADERFILENAME_H

   contents

#endif
```

thus guaranteeing that each header's declarations get processed exactly once.

9.3 Collected directives and declarations

%define api.namespace *NAME* Place all the parser's exported definitions in namespace *NAME* (see §9.2).

%define semantic_type "*TYPE***"** Defines *TYPE* to to be the semantic type of all grammar symbols. *TYPE* may be any POD type (see §6). Lists (created by the `+=` operator) will have type `std::list<`*TYPE*`>*`, where `std::list` is the standard `C++` library list type.

%define semantic_tree_type "*TYPE***"** Defines *TYPE* to be semantic type of all grammar symbols and of all lists of symbols, and enables the `^` operator. By default, all rules will create trees as their semantic values.

%define token_factory "*FUNCTIONNAME***"** Unless you have defined `semantic_tree_type`, lexical rules by default create tokens out of the text of a token using a function named `make_token` (see §5). This definition allows you to specify a different name.

%define error_function_name "*FUNC***"** In case of syntax error, call the function *FUNC*, which you must define in a `%{ ... %}` section, passing it two arguments, both of type `const char*`: a source location, and an error message to print.

%defines "*FILENAME***"** HORN produces a header file containing definitions of token syntax values for use elsewhere in your program. By default, its name is *BASE*`-parser.hh`, where *BASE* is the base for forming the names of the `.cc` files that HORN generates. This declaration replaces the name of this header file with *FILENAME*.

%expect *N* Tells HORN not to complain if there are exactly *N* shift-reduce conflicts in the grammar. In general, you should only use this with GLR parsers, and only after having checked each of the shift-reduce errors to ensure that it is expected.

%expect-rr *N* Tells HORN not to complain if there are exactly *N* reduce-reduce conflicts in the grammar. The same considerations apply as for '`%expect`.'

%glr-parser Produce a GLR parser (see §3.5).

%interactive Produce a lexer that reads as little input as it needs to determine its next token. You'll need this when writing programs that take input from the terminal. Without it, the lexer tries to buffer as much data as it can before producing any tokens. That's generally the more efficient course, but with an interactive program, it simply doesn't work.

%start *SYMBOL* Use nonterminal symbol *SYMBOL* as the start symbol, rather than the left-hand side of the first grammar rule.

%token *NAME* **...** Define the specified *NAME*s (upper-case identifiers) as token (terminal symbol) names. This essentially introduces new integer-valued symbols that stand for the syntactic categories of terminals that may be used in grammar rules. It is unnecessary (but harmless) for names that appear on the left side of a lexical rule.

%left *TERMINAL_SYMBOL* **...** Define the specified symbols to be left-associative operators of the same precedence. Multiple `%left`, `%right`, and `%nonassoc` rules define symbols of different precedence, lowest first. See §3.3.

%right *TERMINAL_SYMBOL* **...** Define the specified symbols to be right-associative operators of the same precedence.

%nonassoc *TERMINAL_SYMBOL* **...** Define the specified symbols to be non-associative operators of the same precedence.

10 Predefined Functions, Macros, and Values

Generated parsers provide a number of definitions to support parsing and lexical analysis.

const char* yysource_location()
Returns the current position in the source file(s).

bool yyis_known_location(const char* loc)
True iff LOC is a location known to the lexer.

int yylocation_line(const char* loc)
Returns the line number within its source file or string of LOC (1-based). Returns 0 for an unknown location.

string yylocation_source(const char* loc)
Returns the name of the source file or string containing LOC. This is the second argument provided to `yypush_lexer` for that source. Returns an empty string for an unknown location.

string yyprinted_location(const char* loc)
Returns a string containing a standard Unix description of location LOC with the form *file name*:*line number*. Thus, it is the result of concatenating `yylocation_source` and `yylocation_line` separated by a colon.

yyqueue_token(int token, T value, const char* loc, const char* text, size_t text_size)

[Usually used in lexical rules.] Add an instance of the terminal symbol denoted by TOKEN (as defined by `%token` declarations or by appearing on the left side of a lexial rule) to the end of the queue of pending tokens to be delivered by the lexical analyzer, letting VALUE be its semantic value. Each time the parser requests a token, the lexer checks this queue first, before looking for an applicable rule. Set the `.loc()`, `.text()`, and `.text_size()` values of the enqueued token to LOC, TEXT, and TEXT_SIZE (which default to NULL or 0, as appropriate).

yyqueue_token(int token, S value)
[Usually used in grammar rules.] As for the first form of `yyqueue_token`, but takes a grammar symbol as the token to be pushed.

const char* yyexternal_token_name(int token)
A printable representation of TOKEN (the left side of a lexical rule or defined by `%token`).

YYMAKE_TREE(*oper, child1, child2,...***)** [Only defined when creating trees.] A macro that gives the same result as `$^` does in context-free rules, but that can be used in the epilogue as well.

YYSET_TOKEN(int token)
[Used in lexical rules only.] Set the syntactic category to be returned by the current lexical rule to TOKEN. A value of 0 indicates the end of input (normally, Horn and Flex supply it automatically upon reaching the end of input, but there are cases where you'll need to produce an "artificial" end of input yourself.) A value of -1 indicates an ignored token (see `YYIGNORE`).

YYIGNORE
[Used in lexical rules only.] Discard the token matched by the current lexical rule. This is equivalent to `YYSET_TOKEN(-1)`.

yypush_lexer (FILE* input, string name)
Start reading input from INPUT (a C file stream), and use NAME as the file name to give for source locations from INPUT. Any current input file is kept at its current location until this file is popped (see `yypop_lexer`). In general, you should use a lexical rule that matches `_EOF` to determine when you reach the end of INPUT and pop it off.

yypush_lexer (const string& input, string name)
As for previous overloading of yypush_lexer, but takes input from a string rather than a file.

yypop_lexer()
Discontinue input from the current input source (file or string) and revert to the input stream active before the call to `yypush_lexer` that started the current one.

yylex_init()
Clear out all inputs from the parser and prepare to restart it.

yyparse()
Begin parsing.

yy_set_bol(V) [Used in lexical rules only.] Indicates whether `_BOL` will match at the beginning of the next rule applcation, overriding the default behavior. The argument V may be either non-zero (true), indicating that the input is currently at the beginning of a line (even if it really isn't) or zero (false), indicating that the input is not at the beginning of line (even if it really is).

NODE_BASE_CONSTRUCTORS(Type, Parent)[Used to define AST node base types
Defines the standard constructors for type `Type`, whose immediate base type is `Parent`. See 7.3.

NODE_CONSTRUCTORS(Type, Parent) [Used to define AST node subtypes] Defines the standard constructors and declares the factory methods for type `Type`, whose immediate base type is `Parent`. These definitions allow the HORN framework to construct tree nodes appropriate for a particular syntactic category. See 7.3.

NODE_FACTORY(Type, Category) Used outside the definition of class `Type` to complete its factory, and to associate that factory with a specified syntactic category (an integer).

NODE_BASE_CONSTRUCTORS_INIT(Type, Parent, ARGS) As for `NODE_BASE_CONSTRUCTORS`, but adds `ARGS` as additional constructor initializers in the constructors.

NODE_CONSTRUCTORS_INIT(Type, Parent, ARGS) As for `NODE_CONSTRUCTORS`, but adds `ARGS` as additional constructor initializers in the constructors.