1 Objective

This document is not an introduction to parser generators. Readers should rather have a good understanding of them, especially of LALR(1) parser generators. This document merely describes the main aspects of the implementation of a LALR(1) parser generator named \textit{wisent}. It is divided into a concise description of the programming interface and an overview over the data structures and specifics of the implementation of the \textit{wisent} parser generator.

2 Motivation

The parser generator \textit{wisent} is a tool like \texttt{yacc}, \texttt{byacc}, \texttt{bison} or \texttt{bison++} for the automated generation of an LALR(1) parser from a grammar specification. Due to the apparent abundance of available parser generators (not necessarily LALR(1)), there is no obvious need for yet another implementation. The trigger for the implementation of \textit{wisent} were problems with the C++ support of \texttt{bison++} that were encountered in the DITO project featuring a processor simulator supporting dynamically reconfigurable architectures [Pre\textsuperscript{+}04]. The alternative of somehow patching decent C++ support into \texttt{byacc} or \texttt{bison} was not as appealing as implementing \textit{wisent} not only supporting C++ but itself written in C++.

The well-structured object-oriented design of \textit{wisent} increases overview, encapsulation and maintainability. Due to the simple replacement and addition of functionality, \textit{wisent} lends itself to the application in academic research or student projects, especially when working on, rather than just with, parser generators. In the current state of development, \textit{wisent} is not yet well-suited for every production application. For example, the lack of table compression requires memory not to be a critical resource. Also there is no sophisticated error recovery from syntax errors built into the generated parsers.

3 Overview

\textit{wisent} is an implementation of the LALR(1) parser generation algorithm as described in [Aho\textsuperscript{+}86]. Studying the chapter on parser generators in this textbook will be a great aid to people planning to dig into the internals of \textit{wisent}. Users of \textit{wisent} should either have experience with the use of LALR(1) parser generators like \texttt{yacc} or \texttt{bison}, or they ought to get familiar with LALR(1) grammars, for example, by consulting the \texttt{bison} online reference [DoSt03].
Since wisent aims at solving the same task as yacc and bison++, adopting much from their grammar specifications was a natural choice. Thus, people familiar with a yacc-like parser generator should not have any problems using wisent.

The syntax of the grammar specification file processed by wisent is itself defined by a wisent grammar specification. This unique feature does not only allow a simple customization of the grammar specification but also implies a C++-API that may serve an arbitrary application defining customized means of input to utilize the wisent LALR(1) parser generator. A parser generated from the standard specification is included in the wisent distribution so that it can be compiled without already requiring the binary.

The C++ sources of the generated parser are comprised of a header and an implementation file. The only global symbol introduced by this parser is its class name defined in its surrounding namespace. All other types, functions or data are members of this class and do not clobber any other namespace.

The implementation language of wisent is C++. The encapsulation into objects allows an intuitive approach to the internal data structures and the operations defined on them. Quite a few of the classes are very thin and resource efficient. The sole purpose of their existence is an intuitive typing and convenient algorithmic handling.

The following sections will give a detailed introduction to the usage of wisent (Section 4) and to the internal design of the wisent parser generator (Section 5).

4 Usage

The standard way to generate a parser by wisent is the processing of a grammar specification file. Assuming the name of this file is parser.ypp, wisent will generate three output files:

1. parser.hpp – the header defining the generated parser class.
2. parser.cpp – the implementation of the generated parser class.
3. parser.log – the log of the parser generation including:
   - the internal representation of grammar symbols,
   - the echo of the grammar rules,
   - the reach and first relations of the grammar,
   - the sets of items representing the states of the LALR(1) parser,
   - the conflicts of the grammar and how they were resolved.

The grammar specification does, unlike traditional yacc grammars, not require any specific indentation or line breaks. As an extreme, the specification of a complete grammar may be contained within a single line. The usage of the percent sign for marking directives has been adopted from yacc. The %% directive separates the three sections of the grammar specification:

1. the definition of the parser class,
2. the definition of named tokens and the start symbol,
3. the actual grammar specification.
%header{
namespace wisent {
    class Builder;
}
#include "GScanner.hpp"
#include <string>
}
%class wisent::GParser {
private:
    GScanner scanner;
    Builder& bld;

private:
    GParser(std::istream& in, Builder& b) : scanner(in), bld(b) {}
    ~GParser() {}

public:
    static void parse(std::istream& in, Builder& bld) {
        GParser(in, bld).parse();
    }

private:
    unsigned nextToken(YYSVal& sval);
    void error(std::string const& msg) const;
%
sval std::string
%
impl{
#include "GParser.hpp"
#include "Builder.hpp"
using namespace wisent;

unsigned GParser::nextToken(YYSVal& sval) {
    int tok = scanner.nextToken();
    if(tok < 0) error(scanner.getText());

    switch(tok) {
    case LIT:
    case IDENT:
    case KLASS:
    case BLOCK:
        sval = scanner.getText();
    }
    return tok;
}

void GParser::error(std::string const& msg) const {
    std::cerr << msg << " in line " << scanner.lineno() << std::endl;
    exit(1);
}
%
}

Figure 1: Sample Parser Class Definition
4.1 The Definition of the Parser Class

Figure 1 contains the definition part of wisent's own parser class. The %header directive precedes a block containing all declarations and definitions that need to be present for defining the parser class. Usually these consist of forward declarations and includes of other headers.

The actual class definition is opened by the %class directive, which is followed by the name of the parser class optionally qualified by a surrounding namespace. The following block contains a class definition in C++ manner. In this case, two data members, a scanner and a builder, are defined.

The construction and destruction of parser objects is made private in this example in favor of a public static parse method, which constructs, calls and destroys a parser object. This approach is probably advisable for most parsers as there rarely is a sensible state to keep between distinct parses.

Finally, the two mandatory methods nextToken and error are declared. The nextToken member must take a reference argument of type YYSVal and return an integral type castable to unsigned short. YYSVal is just a synonym for the type of the semantic value defined by the %sval directive. The error member must accept an argument of type char const * containing the error message and must not return or alternatively throw an exception.

The semantic value type is defined by the %sval directive. This type is required to be default- and copy-constructable as well as assignable. Thus, any primitive data type and appropriate classes are supported. The use of polymorphic classes, which requires pointers in some form, requires more attention. Being LALR(1), wisent can guarantee that every semantic value is shifted only once onto the stack and only used once by a single reduction. Nevertheless, a reference counting smart pointer as semantic value ought to be preferred over a complex ownership scheme. Such an implementation would additionally be exception-safe.

Last but not least, the non-inline custom members need to be defined in the %impl block. This block must also include all headers required by some semantic action as defined in the grammar specification.

4.2 The Definition of Named Terminals and the Start Symbol

Besides the direct use of 8-bit single-character literals, wisent supports named terminals. These are declared in the second section of the grammar specification file. Each of these terminals will be represented by a public class member of the specified name and of an integral type. This member can be returned by nextToken and referred to by an external scanner.

Terminals are defined by a whitespace separated list of C identifiers, which is preceded by one of the directives %token, %nonassoc, %left or %right. Analogously to yacc, this directive identifies the associativity of the defined tokens. The order of the token definition matters in the same way as it does for yacc as each new directive also increases the precedence level of the defined tokens. All tokens defined after the last directive precede all those defined before and will therefore bind stronger than the others when used as infix operator.

For the purpose of defining the precedence relation among tokens, the token definition lists may also contain single-character literals quoted by single quotes. This will, however, not result in the generation of some named class member. Arbitrary literals may be used within productions. Those not explicitly defined will be assigned no precedence. Any shift/reduce such a literal is involved in will be resolved in favor of the shift. The standard C escape sequences are supported for literals.

The second section is ended by the definition of the start symbol of the grammar behind the %start directive. Like the named terminals and all other variables used within the grammar, this must be a unique and valid C identifier.
The wise\textit{nt} \textit{Parser Generator}

\begin{verbatim}
%%
%token BREAK
%token HEADER SVAL CLASS IMPL
%token TOKEN LEFT RIGHT NONASSOC
%token START
%token LIT IDENT KLIASS BLOCK

%start gram
%%
gram : opt BREAK decl BREAK spec ;

opt : header klassd svald impl;
header : HEADER BLOCK { bld.setPreHead($2); }
klassd : CLASS klass BLOCK {
   bld.setClass ($2);
   bld.setClsHead($3);
}
svald : SVAL sval { bld.setSVal($2); }
sval : klass {
   $$ = $1; }
| sval '*' {
   $$ = $1 + '*'; }
| klass '<' svall '>' {
   $$ = $1 + '<' + $3 + '>'; }
svall : svall ',' sval {
   $$ = $1 + ', ' + $3; }
| sval {
   $$ = $1; }

Figure 2: Sample Token and Rule Definition

4.3 The Grammar Specification

The last section of the grammar specification file defines the actual grammar in terms of derivation-rules for non-terminals. The left hand side of a grammar rule is always an identifier representing a non-terminal of the grammar, i.e. this identifier may not have been defined as named terminal in the previous section. The right hand side is a list of arbitrary grammar symbols, terminal or non-terminal. It is followed by an optional block of C++ code describing the semantic action to be executed whenever the reduction defined by this rule is performed.

Although the syntax for the rule definition closely resembles the one used for \textit{yacc}, there are a few differences. As \textit{wise\textit{nt}} does not rely on the source layout to determine the begin and end of a single rule, the unambiguous closing of rules with no semantic actions is required. Thus, the last alternative derivation of a variable must be closed with a semicolon if its semantic action is omitted. Otherwise a semicolon is optional.

Figure 2 shows a short example of the token and rule definition sections. As illustrated there, the semantic values of the right-hand side of the grammar rule can be accessed by the semantic action using the \$n syntax known from \textit{yacc}.

Custom class members, token names and local identifiers as used in the semantic action blocks should avoid the prefixes \texttt{yy} and \texttt{YY} as such identifiers are used by the grammar-independent part of the parser algorithm. Also the use of the names \texttt{parse}, \texttt{nextToken} and \texttt{error} should be carefully watched due to their pre-defined semantics for certain method signatures.

A complete example of a grammar specification is the parser used by \textit{wise\textit{nt}} for the parsing of grammar specification files. It is contained in the file \texttt{GP.ypp} of the \textit{wise\textit{nt}} distribution. Running \textit{wise\textit{nt}} on this grammar specification will not overwrite the readily generated parser distributed with \textit{wise\textit{nt}} which is named \texttt{GParser}. 

5 Internal Design

As mentioned before, wisent does not only generate C++ parsers but is implemented in C++ itself. Before digging into the implementation details, some of the used terminology needs to be clarified. Then the data structures and, finally, the process flow used by wisent will be explained.

5.1 Terminology

The LALR(1) grammar that serves as input to wisent is defined as $G = (\Sigma, V, P, A)$ with

1. $\Sigma$, the set of terminals, i.e. the alphabet;
2. $V$, the set of non-terminals;
3. $P$, the set of productions defining possible derivations from non-terminals;
4. $A \in V$, the start symbol.

For the purpose of this document, the subset of 8-bit, single-character terminals from the set of terminals shall be distinguished as literals. Furthermore, the terms variable and rule will be used as synonyms for non-terminal and production, respectively. The join of the set of terminals and the set of variables will be referred to as the set $S = \Sigma \cup V$ of grammar symbols or plainly symbols. The empty word will be denoted by $\varepsilon$.

5.2 Grammar Representation

This section describes the concepts and data structures used internally by wisent to represent the user-specified grammar. It closes with a brief description of the construction process, by which the grammar specification is transformed into this internal representation.

5.2.1 Symbols

Although both terminals and variables are grammar symbols this relation is not mapped to an equivalent inheritance hierarchy. However, both terminals and variables define a conversion operator yielding an object of type Symbol. The Symbol class itself has the very small footprint of a single signed short member named no denoting the symbol number. The categorization into terminals, variables and epsilon is performed using the members isTerminal(), isVariable() and isEpsilon(). These are implemented inline and very efficiently by a simple comparison of no with zero:

- no < 0: variable #no
- no = 0: $\varepsilon$
- no > 0: terminal #no

5.2.2 Pools

A central concept of the internal grammar representation are pools. They store the elementary components of the grammar like identifiers, terminals, variables and rules. The pool concept ensures efficient memory management and prevents memory fragmentation due to an abundance of small objects.

The elements of a pool are uniquely identified by their index. Using the pool’s method get(), this index can be transformed into a thin wrapper object offering a simple, intuitive interface hiding
the internal pool representation. The lifetime of such a wrapper is limited to that of the pool. Neither
is the validity of the wrapper guaranteed to stretch across the insertion of another element into the
pool as this operation may require the growth and displacement of the allocated data storage.

All pools support the iteration over their contents by a public inner Iterator class. The interface
of these iterators is Java-like and is comprised of the methods hasNext() and next().

5.2.3 Identifiers

Within the grammar specification, identifiers are used to represent both variables and named termi-
nals. These identifiers are pooled internally and can thus be unambiguously represented by their pool
index.

The pool of identifiers is implemented by the class IdentPool. An IdentPool consists of a
dynamically growing data array of char and a dynamically growing index. The data array holds the
concatenation of all defined identifiers preserving their termination by ‘\0’. The index serves for a
fast mapping from identifier indices to their textual representation. It simply holds pointers into the
data array.

The index assigned to the first identifier defined is one (1) as the index entry index[0] is re-
served for a pointer past the end of the used data area. The last valid index entry is followed by
a terminating null pointer, which simplifies the IdentPool’s Iterator implementation. This
structure is illustrated in figure 3.

5.2.4 Terminals

Also all terminal symbols are stored within a single pool containing a dynamically growing data
area. An extra index was not implemented in this pool as all terminals are represented by four
unsigned short values so that the offset into the data array can be easily computed from the
terminal index.

All terminals have an internal as well as an external code. The dense internal code is a natural
number greater than the zero (0), which represents the terminal’s symbol number. This code is for
the internal use by wisent and by the generated parser.

The external code is the one to be returned by nextToken() and thus, most likely, used by a
scanner. It is generally not dense. The external code of a literal is equivalent to its 8-bit representation
whereas named terminals are assigned subsequent external codes starting from 256.

![Figure 3: Structure of the Pool of Identifiers](image-url)
Each terminal is attributed with a precedence and an associativity. Named terminals additionally contain an index into the pool of identifiers to reference their textual representation. The concrete layout of a terminal inside the TermPool is revealed in figure 4. Note that by reserving the two least significant bits for the associativity, it can be stored jointly with the precedence within a single short value.

Figure 4 also describes the wrapper class Terminal. Its only member is a pointer into the data area of its owning pool. Its interface encapsulate the knowledge about the internal data layout and provides an intuitive access to the terminal’s properties.

An Iterator over the TermPool is an equally thin object just containing a pointer into the data area of the pool. A constant increment of four array elements steps to the next terminal. The end of the terminal list is marked by a terminating zero (0). Note that this terminator can be clearly distinguished from terminal data which would begin with a valid terminal index greater than zero.

### 5.2.5 Variables

The representation of grammar variables within the VarPool is even simpler than the presentation of terminals. Variables are simply defined by a symbol number, the equivalent to the internal terminal code, and an index into the pool of identifiers signifying their textual representation.

The dynamically growing data area of the VarPool is an array of signed short. Each defined variable occupies to consecutive array elements: one holding the negative symbol number and the other the index into the identifier pool. The implementation of the thin Variable and Iterator classes are equally simple as they are for the TermPool. A sketch is given in figure 5.

### 5.2.6 Grammar Rules

The structure of the RulePool containing the grammar rules is slightly more complicated due to the fact that rules may be of arbitrary lengths. The basic principles of the pool architecture remain, however, the same. The pool manages a dynamically growing data area, and Rule instances merely contain a pointer into it. The fast lookup of rules is implemented using an explicit index as was done for the IdentPool.
The individual rules may occupy differently-sized blocks inside the data area of the RulePool. A header of four signed short values is common to all rules. These values contain the rule number, the rule precedence, the rule length and the variable derived by this rule, i.e., its left-hand side. The rule definition is completed by the right-hand side of the production and a closing zero (0). All variables and terminals in this definition are represented by their symbol numbers, i.e., their pool indices. Positive ones refer to terminals, negative ones to variables. Note that derivations to ε are represented by a rule header directly followed by the closing zero, which can be validly interpreted as ε.

The user-specified rules are assigned indices starting at one (1). There is, however, always a rule with index zero (0) which derives the user-defined start symbol from the internal start symbol $. This added production turns the original grammar into an augmented one where the acceptance of the input is recognized as reduction by this additional rule. The left-hand side of this rule ($) is coded as zero (0), an otherwise illegal value for variables.

The layout of the RulePool’s data area is exemplified in figure 6 for the following user-specified grammar:

\[
A \rightarrow A 'a' \\
| 'b'
\]

Although, all symbols are actually only represented by their number, their textual equivalent was added here for clarity. Note also that the last entry is a closing negative one (-1). This value was chosen since it must be distinct from all valid rule numbers and zero was thus not an option. An extra RulePool member named eol is maintained to point at this end-of-list marker.

The main purpose of the eol member is to assist the construction of new rules from the grammar specification file. To avoid an extra intermediate representation of a complete production, the internal representation is synthesized step by step directly within the rule pool but beyond the current end-of-list position. When the construction of an additional rule is initiated only its left-hand side needs to be provided. The right-hand side is built up by repeatedly appending symbols. The length of the rule is automatically incremented during this process. The assignment of the rule number as well as the adjustment of the end-of-list marker are not performed until the rule construction is explicitly finished. At this point also the rule precedence needs to be provided.
5.2.7 Construction

The construction of the internal grammar representation is performed by the Builder class. A Builder instance is initialized with the references to the pools to be filled. It is then passed to the parser parsing the grammar specification, which then uses the Builder’s methods as callback stubs.

The Builder maintains some state holding information only required during the construction process. This state includes mappings from identifiers and literals to symbols, a counter of directives declaring tokens and the associativity type of the last of these directives as well as the precedence of the last terminal read and, finally, the left-hand side of the current production.

The mappings to grammar symbols speed up the identification of symbols already defined as this task would be rather expensive when each look-up required the scanning of possibly both the TermPool and the VarPool. These mappings are not required for the further processing of the grammar and the parser generation and are, therefore, kept local to the Builder.

The current value of the counter of token-declaring directives determines the precedence of declared tokens. It is incremented each time a new %token,%left,%right or %nonassoc directive is found. Any such directive will also properly adjust the current associativity type applying to...
all following token declarations until re-adjusted.

The left-hand side of the current production is saved in the local state so that alternative derivations of the same variable may be specified as such in the grammar without requiring each of them to be a stand-alone rule. The pipe symbol (|) serves for this purpose in the grammar specification. Whenever one derivation is finished the corresponding rule inside the RulePool will be closed. Its assigned precedence is determined by the precedence of the last terminal read.

Beside the core grammar elements, the Builder needs to process the user-specified implementation fragments to be integrated into the generated parser. Most of this information is stored inside a Context object including the name to be given to the generated parser, the type of the semantic value, the user-specified header and class layout as well as the custom parts of the implementation. Only the specified semantic actions to be executed upon a reduction step are stored in a separate ActionPool. Unlike the pools of the grammar components, this pool does not implement some dynamically growing data area but is implemented using an STL\(^1\) vector of strings mapping rule numbers to their corresponding actions.

5.3 Parser Generation

This section describes the grammar analyzation and parser construction as performed by wisent. Following a brief overview on these processing steps, their implementation will be detailed and the used data structures introduced.

5.3.1 Overview

Once the internal grammar representation has been constructed from the grammar specification, it is analyzed for the construction of the LALR(1) parser. This analyzation is centered around an Analyzer object, which supervises the construction of a Collection. This Collection embodies the collection of sets of items representing the states of the parser to be generated. This Collection is used to generate the two parser tables action and goto common to all shift-reduce parsers. These form together with some other administrative tables and the fixed parser algorithm the final output of the generated parser which is divided into a header and a C++ source file.

5.3.2 Grammar Analyzation

The first step of the grammar analyzation is the pre-calculation of two essential relations between the grammar variables and symbols. Both of these relations capture which symbols may appear leftmost in derivations of a variable. They only differ in their treatment of ε productions and their reflexiveness.

Let \( S' = S \cup \{ \varepsilon \} \). Then reach and first are relations over \( V \times S' \) and constructed as follows:

Relation reach:
1. Add \((v, s) \in V \times S\) to reach if there is a production \( p \in P \) and an \( \alpha \in S^* \) with \( p : v \rightarrow s\alpha \).
2. Add \((v, \varepsilon)\) with \( v \in V \) to reach if there is a production \( p \in P \) with \( p : v \rightarrow \varepsilon \).
3. Close reach transitively.
4. For each \( v \in V \) add \((v, v)\) to reach, i.e. make reach reflexive.

\(^1\)Standard Template Library
Relation first:

1. For each variable \( v \) that is derivable to \( \varepsilon \) add \((v, \varepsilon)\) to first.

   *This step is implemented by iteratively marking variables known to derive \( \varepsilon \). It starts by marking variables with \( \varepsilon \) productions and continues marking variables with productions whose right-hand side only contains already marked variables. This process is continued until no new marks are generated.*

2. Add \((v, s) \in V \times S\) to first if there is a \( p \in P \), an \( \alpha \in S^* \) and \( \beta \in V^* \) with \( p : v \rightarrow \beta s\alpha \) where all variables in \( \beta \) may derive \( \varepsilon \).

3. Close first transitively, however, without adding pairs \((v, \varepsilon)\).

For clarification, consider an example grammar \( G = (\Sigma, V, P, A) \) with

\[
\begin{align*}
\Sigma & = \{a, b\} \\
V & = \{A, B\} \\
P & : A & \rightarrow & B & a \\
 & B & \rightarrow & b & | \varepsilon
\end{align*}
\]

Using the above construction algorithms the following relations are obtained for this example:

<table>
<thead>
<tr>
<th>reach</th>
<th>( \varepsilon )</th>
<th>a</th>
<th>b</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>first</th>
<th>( \varepsilon )</th>
<th>a</th>
<th>b</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

In wisent each of these relations is represented by a SymbolMatrix, which is a very thin wrapper around an efficient BitMatrix implementation. Every matrix line can be accessed in whole as a SymbolSet, which is again just a thin wrapper around a BitSet. Using the unify operations provided by these sets, the transitive closure is implemented efficient.

The wrapper around BitSet to form a SymbolSet simply translates Symbols into BitSet indices to uniquely identify a bit position. This translation is performed using a mapping that assigns zero (0) to \( \varepsilon \), its positive symbol number to each terminal, and its negative symbol number added to the SymbolSet’s length to each grammar variable. This is why terminals appear in declaration order behind \( \varepsilon \), and variables in reverse declaration order at the end of the index space. Note that code using the SymbolSet must ensure that the index ranges of terminals and variables are distinct by constructing sets of sufficient sizes. No range checking is performed by the SymbolSet; it simply forwards positive symbol numbers to the underlying BitSet and adds the set’s length to negative ones.

### 5.3.3 State Space Construction

Both the reach and the first relation are tightly coupled with the construction of the state space and transitions of the parser to be generated. Please, refer to [Aho + 86] for details of this construction process.

The state graph of the LALR(1) parser is generated as LR(0) state graph where each state is characterized by a set of LR(0) items. Each item represents a parse position denoted by a dot within a grammar rule. The items of an LR(0) state are all parse positions conforming with the input seen so far. As usual, each set of items is internally represented only by its core, i.e. all items starting with a dot are left out unless it is the item \( S \rightarrow \cdot A \) of the added rule #0 for the start symbol \( A \).

As illustrated in figure 7 for the small example grammar, all LR(0) state transitions described by a goto function for each state can be easily computed from the core of a state if the reach relation.
is available as follows:

1. If the core contains a production $A \rightarrow \alpha \cdot a\beta$ with $A \in V$, $\alpha, \beta \in S^*$ and $a \in \Sigma$: add $A \rightarrow \alpha a \cdot \beta$ to goto(a).

2. If the kernel contains a production $A \rightarrow \alpha \cdot B\beta$ with $A \in V$, $\alpha, \beta \in S^*$ and $B \in V$: for all rules $C \rightarrow X\gamma$ with $(B, C) \in \text{reach}$ and $X \in S$, add $C \rightarrow X \cdot \gamma$ to goto(X).

Note that for each $X$ encountered here $(B, X) \in \text{reach}$ must hold by construction of reach.

wisent discovers the relevant state space of the parser by subsequently calculating the goto functions of all new states until no more new states are found as results of goto functions. The discovery begins at the state described by the core $\$ \rightarrow \cdot A$ for the start symbol $A$. The states are assigned unique subsequent natural numbers as discovered. The initial state is numbered zero (0). Note that this construction process resembles a power set construction over the set of all items of the grammar.

The first relation assists the fast calculation of the FIRST function, which is extensively used by the lookahead calculation transforming the LR(0) state space into an LALR(1) state space. The individual lines of the relational matrix representation of FIRST are exactly the function values of the FIRST function for the variable indexing this line. As described in [Aho+86], the lookaheads are calculated in two phases starting with the identification of spontaneous lookaheads and lookahead forwarding paths and finishing with the propagation of lookahead paths along these paths.

The data structure used internally to represent the state space is the Collection of sets of items as depicted in figure 8. This collection essentially contains an STL map assigning a modifiable CoreInfo object to a specific Core. The Core is a simple set of Items, each identified by a rule index and the parse position, i.e. the dot. The CoreInfo object holds all information yet to be collected during the construction process. This includes the goto function of the corresponding state as well as the lookahead information belonging to each of the core items.

After the calculation of the LR(0) state space and the augmentation of the LR(0) items with lookahead information, the resulting collection of states is output to the .log file. Each of the states will be represented by its core and is accompanied by its goto function.

5.3.4 Table Generation

The tables controlling the generated parser are extracted directly from the calculated state graph. All conflicts that may arise are resolved, i.e. there is no conflict situation that will result in the parser generation to be aborted. The resolution of the conflict is documented in the .log file.
The conflict resolution is performed using the same rules as traditional yacc:

1. The rule with the lower index, i.e. the one declared first in the grammar specification, will win a reduce/reduce conflict.

2. Shift/reduce conflicts are resolved in favor of the shift unless both the rule as well as the lookahead have an associated precedence and the rule’s precedence is greater or it is equal to that of the lookahead, which is left associative.

All actions are stored within an ActionTable. This table takes a state number to select a row and a Symbol to identify the column therein. The table entries are of type signed short initialized to zero (0), which indicates an undefined action, i.e. an error. It is filled by traversing the state space.

Within an ActionTable, shifts are represented by a positive integer signifying the state to go to after the shift. Reductions are represented by negative integers identifying the rule to reduce by. Note that the state after a shift may neither be 0 (always: $ \rightarrow \cdot A$) nor 1 (always: $ \rightarrow A \cdot$). So these can be used to code the special values error (0) and accept (1). Latter is actually identified as the reduction by the added rule #0. The code 0 was nevertheless preferred to code the undefined entries.

An ActionTable also holds the goto table of the generated parser. The columns of this subtable are indexed by variables whereas the parser actions, shift and reduce, where indexed by the lookahead, i.e. a terminal. The goto subtable only contains positive integers coding the target state number.

The completed ActionTable outputs both the action as well as the goto table as the members yyaction and yygoto of the generated parser. Whereas these members are declared within the header, their definition, i.e. the contained data, will only appear in the implementing C++ source file.
Three additional arrays are generated in the output directly by the RulePool:

- `yyrules` contains a string representation of each of the productions. This array will only be present in the executable compiled from the parser source if the preprocessor symbol `TRACE` is defined.

- `yylength` holds the length of the right-hand side of each production. This determines the number of stack entries to pop after a reduction step.

- `yylhs` stores the bitwise negation of the variable number on the left-hand side of each production. This yields the column index into the `yygoto` table after a reduction step. The bitwise negation merely maps the integer range $-1, -2, \ldots$ to the range $0, 1, \ldots$ yielding a more intuitive indexing.

Another array named `yyintern` is generated by the TermPool. It maps the external representation of a token to its internal one. Zero (0) as the code of `EOF` is always mapped to zero. All others are defined as required. Since generally not all literals are valid terminals, there must be a value identifying undefined entries. This value is `YYINTERN`, which is one beyond the external representation of the last non-literal or 256 if there is none.

### 5.3.5 The Fixed Parser Algorithm

The fixed parser algorithm is a straight-forward implementation of a shift-reduce parser. It is parameterized by the tables described in the previous section and contains a `switch` statement composed from the semantic actions of the grammar specification. It maps the rule index to its corresponding semantic action. The parsing algorithm is otherwise fixed.

Unlike `bison`, the current implementation does not include features as early reductions without a lookahead. A lookahead is always fetched, and the traditional shift-reduce processing is preserved.

The parser stack is implemented by the inner class `YYStack` of the generated parser. It is based upon the STL `vector` container class and stores pairs containing a state number and an instance of `YYSVal`, the semantic value type specified in the grammar specification. The following operations are defined for this stack:

- `push(unsigned short state, YYSVal const& val)` pushes a new pair onto the stack. If `val` is omitted the default constructor of `YYSVal` is used to construct a semantic value.

- `pop(unsigned cnt)` pops the `cnt` topmost pairs from the stack.

- `YYSVal& operator[](unsigned idx)` returns a reference to the semantic value of the `idx`th pair from the top of stack.

- `unsigned short operator*()` returns the state stored in the pair on top of the stack.

These operations are used in the obvious manner. It should be noted that the `[]` operator is the only member accessed by the semantic actions defined by the user. In fact, it is the C++ translation of the `$n$` placeholders used in the grammar specification.

Using the STL `vector` as base for the stack ensures a simple, exception-safe implementation with a dynamically-adapting storage size and well-defined construction and destruction semantics. This is an important property not only lacking in the C-based parser generation tools but also in the standard skeleton of the C++-oriented `bison++`. 
The generated parser produces verbose output about the performed parsing actions when it is compiled with the preprocessor symbol `TRACE` defined. As line information is included in the generated source, sensible error reporting by the compiler, e.g. pointing directly to a misformatted semantic action, is possible. As supported by the compiler, this line information also allows the online debugging of the parser on the source code level.

6 Summary

The wisent parser generator for the automatic generation of C++ sources for an LALR(1) parser has been introduced. Its programming interface was described, and its internal data structures and algorithmic concepts were explained.

Wisent is planned to be released as open-source project under the GPL shortly.

References

