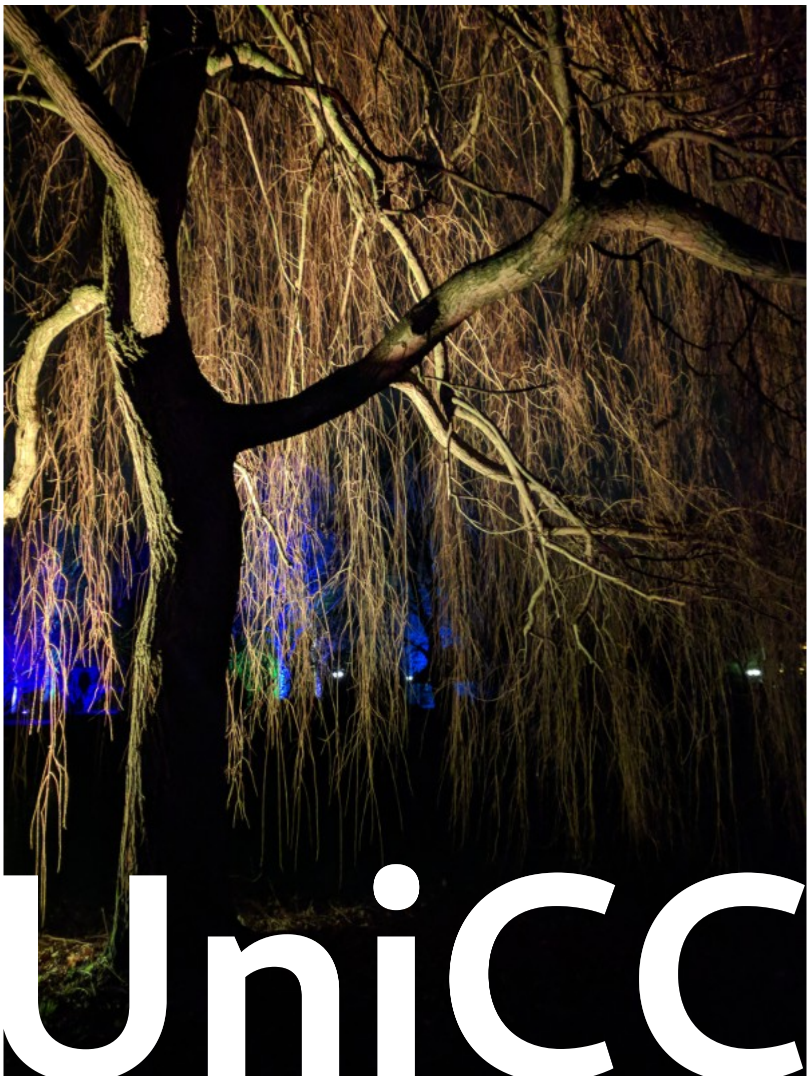


UniCC

A universal LALR(1) Parser Generator

User Manual

for UniCC Version 1.4 and higher

Phorward
Software Technologies

UniCC: A universal LALR(1) Parser Generator
User Manual

Manual-Version: **1.4.0**
Release Date: **September 10, 2018**

Written and published by

Phorward Software Technologies
Jan Max Meyer – Software Developments
Castroper Str. 65
44357 Dortmund
Germany

jmm<at>phorward<dot>de
https://phorward.info

Many thanks to AGS for corrections and typo fixes on all Phorward Software documentations!

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1. Introducing the UniCC parser generator

1.1. Welcome to UniCC!

UniCC, standing as an acronym for *universal compiler-compiler*, is a parser generator and language development system for computer professionals. Its design-goal is to serve as an all-round tool for both design and building. It should assist compiler-writers in any parsing-related task, including production-quality compiler construction and the implementation of domains specific languages.

Flexible grammar prototyping and parser definition

UniCC unions both a generator for lexical analyzers and a parser generator into one handy software solution. The programming interface of UniCC is an easy-to-use, extended Backus-Naur-Form-based grammar definition language. This language gives the compiler-developer's task much more comfort and simplicity in implementing parsers than ever before.

This grammar definition language comes with useful features for grammar prototyping, parser optimization, semantic augmentation and semantic programming. Lexical symbols can be directly defined within productions, right-hand side items can be referenced by meaningful names within semantic actions, instead of only their offsets. Features like virtual- and embedded-productions finally help to rapidly build-up iterative and optional grammatical structures. Standard features like automatic conflict resolution, terminal and production precedence association, state compression as well as parser trace and behavior modification trough semantic actions are self-evident.

Lexical analysis on a context-free level

By default, UniCC constructs whitespace-sensitive parsers. This paradigm is a speciality of UniCC and causes an internal revision of the grammar according to rules that match whitespace only under certain circumstances. In this case, the whitespace handling will be performed by the parser rather than the lexer, enabling to use the entire power of a context-free grammar for creating complex token structures or whitespace constructs. If this feature is not required by a grammar, it can be switched off, resulting in smaller parse tables and faster parsers.

One parser generator, multiple targets

By design, UniCC serves as a target language independent parser generator. Therefore, it is not bound or limited to emit parsers written in one special kind of target programming language. For now, UniCC fully supports C, C++ and Python as target languages. More targets are planned. Support for new or different target programming languages can be done with minimal effort, because every code generation target is specified in an XML-based template file which can easily be adapted for other languages.

Next to emitting parsers in a particular programming language, UniCC also offers the possibility to export all the grammar information, the parse tables, lexical recognizers, semantic actions and settings extracted from a UniCC parser definition into an independent, XML-based output format that can be handled by any individual subsequent tasks or module, like optimizing code-generators, analyzers, or direct parser interpreters.

1.2. The intention behind UniCC

The UniCC parser generator has its origin at *Phorward Software Technologies* and is initially written and maintained by Jan Max Meyer. Phorward Software Technologies is a one-man business from Dortmund (Germany), providing expertise in the area of parsing, compiler construction and systems programming.

The initial intention behind UniCC was the development of a high-portable software solution for parser implementations that is flexible enough to fit many different demands and that constructs parsers that can be compiled and run on almost any platform.

The previously planned reference implementation for a UniCC-compiled parser was *RapidBATCH* version 6. RapidBATCH (https://rapidbatch.com) is a scripting language - formerly for Windows operating systems only - that was quite popular between 2001 and 2007. Due to the lack of time and some personal differences, RapidBATCH 6 sadly never came to an end, but it was reborn as an open source project in 2017. Nevertheless, UniCC and its underlying toolkit library *libphorward* was continued over the years, and is used in several proprietary software projects spreading over a wide array of platforms, including Linux, AIX, Solaris and Windows.

To get more information on Phorward Software Technologies and our projects and products, visit https://phorward.info.

1.3. Some words on this manual

Parsing and compiler-construction is on of the most complex, on the first view tight-looking, but even fascinating and challenging topic of computer science. The knowledge and experience teached by this topic can be integrated into many software development issues of a programmer's everyday life and opens new possibilities and faster ways to success.

Some readers of this manual would have already taken some experience on this complex topic, maybe from a study on computer sciences, a business-related project that used some kind of programming or definition language or a private approach of writing a compiler for an domain-specific programming task. Rather others don't have any knowledge yet, but want to know how things work or require knowledge for an upcoming project.

The UniCC User Manual is - as it stands for - a user manual for the UniCC parser generator, and not a general textbook on compiler-construction. So this manual immediately starts into the topic of practical parsing, with the assistance of examples and the learning-by-doing principle. Deep knowledge on what's going on behind the grammars, how the parser internally works in detail and how it is constructed is not required or even covered here. But if there's more interest in these topics, e.g. language theories, parsing concepts, machine code generation and their optimization, or if generally deeper information on the topic of parsing and compiler construction is wanted, it is heavily recommended to read some adequate textbooks covering all these topics. This manual does only focus on the usage of UniCC itself, and how parsers are implemented with it.

Although the UniCC parser generator comes with entirely target-language independent facilities, all programming-related examples in this manual are in the C programming language. The C standard template for the UniCC program module generator, which is delivered with the UniCC program package, is currently the only well-tested and proven parser framework for UniCC so far. In future, other implementation languages and frameworks will be made available by Phorward Software Technologies or by third-parties.

The UniCC User Manual is divided into four sections:

- The first section contains a quick start guide into parser development in assistance with UniCC. It is written in the style of a tutorial, and is advised to quickly begin using UniCC and to become familiar with the UniCC parser definition language. Authoring of this section has been started already a few years ago during earlier development stages of UniCC, so this is the reason why it is not up-to-date with all the technical possibilities UniCC provides right now. But it's a good place to start. The goal of this quick start guide is the implementation of a small programming language compiler, called *xpl*.
- The second section is the UniCC reference guide relating to the installation or build, the use of the UniCC command-line interface and the general features of UniCC.
- The third section directs to the grammar definition language and all its features in a detailed way for topic-based lookup. This chapter also includes practical examples and snippets on feature-related problems and their solutions.
- The fourth section provides information about the Standard C Parser Template delivered with the UniCC software package. It should be mostly attended by C programmers who want to develop parsers with UniCC targeting the C programming language. There will be more sections or separate manuals like this one for other target languages, e.g. C++ or Java, as soon as they are implemented and well tested.

Hopefully, this manual answers most of the questions coming up when UniCC shall become the workhorse of your upcoming project requiring a compiler or parser to be implemented. If not, don't be afraid to drop a mail or issue on https://github.com/phorward/unicc.

1.4. UniCC is free software!

Since 2016 and version 1.1, UniCC, the parser generator, its build tools and its supported target language templates are released and distributed under the terms and conditions of the 3-clause BSD license. Everybody is permitted to obtain, use, copy, modify and share the program and its sourcecode freely and without any charges. Since 2017, the main development of UniCC is done on GitHub, where the sourcecode, a bug tracker, a wiki and more resources remain. The official URL is **https://github.com/phorward/unicc**.

With version 1.2 in 2017, all parts of UniCC, including the parser templates and the minimal parser generator *min_lalr1*, that is used for bootstrapping UniCC, had been unified into one single source repository. So there is no more scattering of all the UniCC sources anymore into several repositories, which has been previously done due different licensing terms and origins.

The UniCC parser generator makes heavy use of the toolchain provided by the *Phorward Toolkit*, shortly also called *libphorward*. It is the base library and backend toolchain implementing much of the functionality[1] provided by UniCC. It is also written in C, and especially the computation of the lexical analysis part is done here. UniCC does all the stuff related to the parse table computation, grammar revision, code generation and finally the implementation of parsers working on the generated tables. libphorward is also released and distributed under the 3-clause BSD license, but developed apart from UniCC, by different reasons. Source code, help, resources and support can also be found on GitHub at **https://github.com/phorward/phorward**.

Later in this manual, there is also an how to guide to build UniCC from source. Any contribution to UniCC or libphorward, be it code, bugfixes, parser templates, ideas, enhancements, proposals or documentation is always welcome. Help is also appreciated in form of support, advertising or simply by using it.

Thank you!

---

[1] *libphorward* does also provide newer tools for parser construction and abstract syntax tree processing. These tools are currently in an imprecise development state, but will sooner or later provide as the development base for an upcoming UniCC version 2 in future. For more about that, please visit https://github.com/phorward/unicc/wiki/UniCC-v2.

2. Constructing parsers with UniCC

2.1. Parsing - a developers everyday task

Parsing is required in many situations. Information is fed to a program and must be analyzed, sometimes with a more or less logical structure. Information can be a data dump in a character separated file, the command-line parameters of a program, a stream of data in various formattings, an XML-file or another type of file with a logical, meaningful structure, and input syntax - maybe even a program source code written in a programming language.

This syntax can be analyzed and verified for correctness using a special type of program task: A parser. This parser is used to analyze a sequence of input data and produces a logical interpretation of this data according to an underlying grammar, which describes the data's valid syntax - rules that express this data in its logical way. If the syntax a parser follows is not matched, the information is useless or error-prone and cannot be processed by subsequent tasks that rely on the correctness of the parsed information.

Every compiler, as the most common example, has a parser for the language it compiles - the language must follow the correct syntax to let the compiler produce valid output, like an assembly program or a program in a lower programming language. In nearly all business and trough the whole bunch of areas where information technology is used, languages are created to describe data structures, configuration files, workflows, ways of how information could be accessed - all such tasks may require an underlying parser that fetches the information expressed in a domain-specific language approach. Serving another example, a program like an appointment assistant, allows to import appointments from a file and accepts date and time-values in different, human-readable formats, for example "May 5, 2008" or "5.5.08" - a parser is required to analyze the correct format, based on a syntax which describes all possible formats a date can expressed. As you can see, there are so many different applications requiring a parser, that all of them never can be mentioned or grouped. The limit is only the creativity of every individual, and his or her skills and ideas.

The UniCC parser generator supports the programmer dealing with any type of parsing issue in two important steps: A new (or existing) language can quickly be prototyped and tested. This prototype then can be used to implement the parser that constructs an abstract syntax tree out of the given input, which is in turn used for the further processing of this fetched information.

2.2. Defining grammars

The ways of how data is analyzed, their syntax rules, can be defined using a so called *grammar* - the grammar of the language the parser accepts. Every kind of computer language - even simple text matching patterns - can be defined in some kind of grammar.

Grammars for computer languages, so called *context-free languages*, are expressed in a special notation, which is called the *Backus-Naur-Form*, or shortened just *BNF*. It was invented in 1959 by John Backus and Peter Naur in the course of the ALGOL programming language.

Grammars expressed in Backus-Naur-Form exists of three fundamental elements: The terminal-symbols, the nonterminal-symbols and the productions. Because all of these elements integrate together, it is not possible to explain them separately. Moreover, lets first define what the purpose of these elements is.

- **Terminal symbols**, or simply called *terminals*, are language atomics, which are directly read from the input stream. A terminal symbol can be a single character, a character from a set of possible (allowed) characters, a string sequence or a regular expression that matches a classifying pattern. It is on the language designer's choice how terminal symbols are made up in the particular implementation. Some examples for widely used terminals in programming languages are identifiers for variables and functions, operators, brackets, keywords like *IF* or *WHILE*, floating point or integer numbers. The parser will expect these terminals in a valid order according to the position it is during the parse - which is in turn defined by the underlying grammatical rules it follows.
- **Nonterminal symbols**, or simply called *nonterminals*, can be seen as "variables" or "function calls" within a grammar, although they aren't. They reference to one or a bunch of the so called *productions*, which means that each production is always associated with one (and only one!) nonterminal; but one nonterminal may exist of several productions.
- **Productions**, sometimes even called *grammar rules* or just *rules*, finally describe the syntax. Its better to say, that productions define a syntactical part of the grammar - which can be replaced by the specific nonterminal each production is associated with. This syntactical description is done by defining a sequence in which terminals and nonterminals may occur to form a valid sentence. This includes, that a nonterminal can reference itself recursively in its own productions, which is a very important aspect in non-regular languages.

Let's see an example to get more familiar with these new terms. We want to define a language that allows for the detection of integer numbers. We have one terminal, which is a character-class that exists of a digit from "0" to "9". This means, that the characters 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 are our valid characters forming this specific terminal, and we generalize them by defining a character-class by saying `'0-9'`. Characters and character-classes are enclosed by single quotation marks to identify them as terminals.

Then we have one nonterminal in our grammar, let's call it *integer*. Nonterminals are directly identified by writing their name, like `integer`.

Using these two elements, we can now define productions for them. We begin with the requirement that *integer* should only exist of a single digit. Then we write

```
integer -> '0-9' ;
```

This single line defines all three elements described above: One *nonterminal* called `integer`, one *terminal* defined as `0-9`, and one *production* that is associated with the nonterminal `integer`. This production defines that one digit is the valid input to match the rule that forms an integer.

The arrow-symbol `->` separates the nonterminal-name from its production definitions, semicolon `;` closes the definition block. Because of this syntactical structuring of grammar elements (note that BNF *is* even a language with its own grammar to express grammars!) the nonterminal-name is sometimes called as the "left-hand side", where the associated productions resist on the "right-hand side". Using this definition, our grammar allows for integer numbers with one digit.

If we want to extend the grammar now to allow for one or multiple digits forming an integer, we have to add a new production which calls nonterminal `integer` recursively, besides its one-digit rule.

Adding such a new production is done by separating the first production from the second, using a pipe-character `|`. Adding more productions requires even more pipes to separate them.

```
integer -> '0-9' | integer '0-9' ;
```

This is now a finished and valid grammar to detect multiple-digit integer numbers.

The concept becomes increasingly clear when we see how the parser moves along the input for a number, e. g. "321", and how it consumes productions each digit. Let `$` act as an end of input marker.

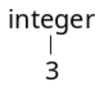
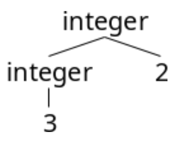
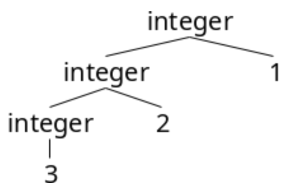
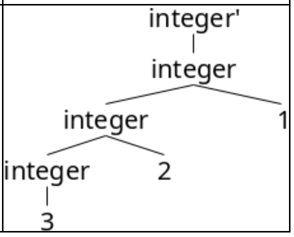
Input	`321$`	`21$`	`1$`	`$`
Productions	`integer -> '0-9'`	`integer -> integer '0-9'`	`integer -> integer '0-9'`	`integer' -> integer`
Parse tree				

Fig. 1: Construction of parse tree from the string `321`

From this parse-process and the structure, how the input string is absorbed by the parser, the following visualized recursion tree is constructed.

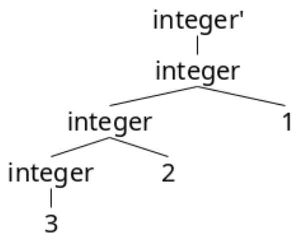


Fig. 2: Parse tree derived from the input "321" (simplified)

This structure is called the *parse tree* of the input, where the input is broken down into its grammatical structure created by the parser. Due the left-recursion of our simple grammar, the tree grows left-leaning, because the leftmost, recursive call to nonterminal *integer* results in a new branch each level. As you can see, all leafs of the parse tree are terminal symbols. It is also possible to write right-leaning trees, but in the LALR parse algorithm, right-recursive grammars require much more parse stack usage than leftmost ones while the parser analyzes the input. Parser stack usage should always be kept low, and this is the case in leftmost grammars.

A tree of this kind is automatically generated for each input a parser analyzes. But its only a fictitious tree, a data structure that is constructed virtually from the natural and logical flow of the parse process and the underlying grammar. It is only a visualization of how the parser walks along the grammar and maps the input into this structure. Like with the above grammar, each call to a nonterminal in a production allows the parser to parse this nonterminal's entire underlying structure, which includes all its productions, the productions of the nonterminals in these productions and so on, in any possibility the syntax allows for. This can cause branches to giant structures in this virtual parse tree! Just imagine what happens when we parse a thirty-thousand digits huge integer number using the above grammar; It won't be a problem for the computer to parse this input, but it results in a giant, logical structure, that can be visually mapped into such a parse tree. And a thirty-thousand digit integer can never be stored to a normal computer variable for further calculation.

To have a more abstract view on such a parse tree, compiler-writers are rather dealing with another kind of tree-structure, which is called the *abstract syntax tree*. Abstract syntax trees are derived from the parse-tree, and represent only the logical structure of information from the parse-tree, by hiding or merging syntactical details which are not mandatory to keep-up the parsed structure.

By using these virtually constructed trees, compiler-writers can perform several actions to be executed on the particular production, e. g. generating output code or building-up data structures to be used by subsequent compiler-related actions - this is the way how compilers are written along the parser. Unconsciously, compiler-writers do excessively make use of abstract tree structures along the parse tree when writing compilers. But this will be discussed in the next chapters. We now only rely on the definition of the grammars itself.

This example was really very simple. What about, if we go back to the idea of the appointment assistant mentioned in the above chapter - a simple grammar to detect dates in various formats? First, we want to parse dates like `August 18, 2008` which follows the basic syntax `<Name of Month> <Day>, <Year>`.

Semantic checks, e. g. if a valid day for the month or year is given, will be ignored to simplify matters. For the day and year number, we can re-use the integer-grammar from above here to integrate with this new "date grammar".

The date grammar definition will simply be:

```
#whitespaces    ' \t';
#lexeme         integer;

date$           -> month integer ',' integer ;

month           -> "January"
                | "February"
                | "March"
                | "April"
                | "May"
                | "June"
                | "July"
                | "August"
                | "September"
                | "October"
                | "November"
                | "December"
                ;

integer         -> '0-9' | integer '0-9' ;
```

This is a valid UniCC grammar definition. Instead of the first, simple grammar definition, this grammar can be fed to UniCC as it is, and produces a working parser.

In the first three lines, some parser configuration is performed. An *end-of-input symbol* is defined as a line break (by default, this is the zero-byte), and a character-class acting as whitespace symbol is defined. How this special kind of symbol is handled will be discussed later. You currently just have to know what its meaning is.

The third line defines nonterminal `integer` to be a *lexeme*. This must be done to disallow whitespace between the terminals of `integer` itself, to let it act as a syntactical coherent unit. If this is not done, the input "12 34" would also be valid for one single integer, although it would be stored as "1234" internally - but this isn't our goal, nor a syntax that we allow for a valid date.

Nonterminal `date` defines the basic syntax of a date. The dollar-symbol `$` behind `date` defines `date` to be the *goal symbol*. In terms of LALR-parsing, the goal symbol is the last symbol identified by the parser to ensure that the parse is finished, complete and valid. This is due the bottom-up approach of LALR-parsers: The parse tree is constructed from the leafs (the terminal symbols) up to all nonterminal symbols, to finally match the goal symbol. The nonterminal marked as goal is always the root from which all subsequent branches in the yielding parse tree will start.

Next to `date`, nonterminal `month` defines all month names in its productions, where each month name is represented by a so called *string*. String terminals are enclosed by double quotation-marks, and are a kind of terminal symbol. In comparison to characters and character classes, strings require that the character sequence from which they are made up exactly match to the input characters coming up next in the input stream. More on this topic will even be discussed later.

Finally, we use the same nonterminal, which is `integer`, for day- and year-number. The parse tree of the input string "August 17, 2008" using this grammar will be

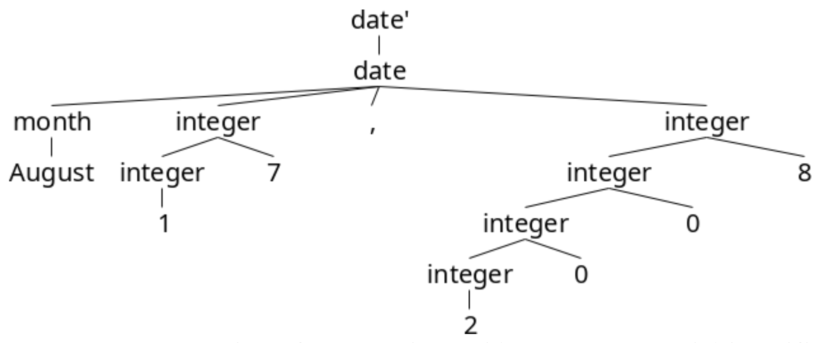


Fig. 3: Parse tree derived from the input 'August 17, 2008' (simplified)

2.3. Grammar modularity

Grammars are modular. They can simply be extended and rewritten, by re-using already defined grammatical constructs as part of other grammatical constructs. Due the build-up, that one nonterminal can be derived from one or multiple productions, grammars are defined in a modular structure, where existing elements can be replaced or enhanced without rewriting the whole grammar.

This will be demonstrated by the next example: What about extending the appointment assistant to even accept other date formats, e. g. the format `<Day>.<Month>.<Year>` and `<Month>/<Day>/<Year>`?

In this case, we can reuse the nonterminals of the above grammar, and extend the grammar to fit our requirements simply by adding some productions. Extending nonterminal `date` will tune up our grammar to recognize more than one date format.

```
date$           -> month integer ',' integer
                | integer '.' integer '.' integer
                | integer '/' integer '/' integer
                ;
```

Fine! And now, we want to extend our appointment assistant grammar to allow for adding an appointment message using the format `<Date> "<Title>"`, or why not even `"<Title>" <Date>`?

Here, the grammar's goal symbol, which was `date` until now, must be replace with another goal symbol, the `appointment`, which is a more meaningful goal for the input we want match. The final grammar fulfilling this request is shown next.

```
#whitespaces    ' \t';
#lexeme         integer title;

appointment$    -> date title | title date ;

title           -> '"' !'"'* '"' ;

date            -> month integer ',' integer
                | integer '.' integer '.' integer
                | integer '/' integer '/' integer
                ;

month           -> "January"
                | "February"
                | "March"
                | "April"
                | "May"
                | "June"
                | "July"
                | "August"
                | "September"
                | "October"
                | "November"
                | "December"
                ;

integer         -> '0-9' | integer '0-9' ;
```

You see, that we have quickly extended the original grammar for simple dates to match a small language, that allows for entering appointments with a date in several formats.

Maybe you already noticed some new elements used in this grammar, e. g. the production

```
title           -> '"' !'"'* '"'
                ;
```

This production is enabling a title that is enclosed as a string literal. The terminal definition `!'"'*` defines a terminal that exists of all possible characters instead of the double quotation mark (which is specified by the prefixed exclamation mark), and this with an automatic repetition of zero or multiple characters (invoked by the postfixed star-character).

The first of these two new symbols, the exclamation mark, is a negation which belongs to the character class it precedes.

Secondary, the star is a modifying operator that causes a Kleene-closure of zero or multiple of the preceding symbol, in this case the negated quotation mark character class. These modifier operators can be used everywhere in the grammar definition behind nonterminal and terminal symbols on the right-hand side and invokes the so-called *virtual production* feature of UniCC. For each of these virtual productions, UniCC automatically inserts a virtual nonterminal that implements the desired syntactical element in its correct, well-formed grammatical structure.

UniCC provides the following three virtual production operators, which are well-known from regular expressions.

- ***** for a "kleene closure" (multiple or none repetitions)
- **+** for a "positive closure" (one or multiple repetitions)
- **?** for an "optional closure" (one or none repetitions)

For the above shortcut, the contract-out version would be

```
title           -> '"' virtual1 '"'
                ;

virtual1        -> virtual2
                |
                ;

virtual2        -> !'"'
                | virtual2 !'"'
                ;
```

As you can see, there is much lesser effort in writing the grammar by reaching the same effect.

2.4. Building parsers with UniCC

UniCC is a parser generator. This means that is generates parsers from such as the above shown grammar definitions into an adequate, simplified parser representation. UniCC parses grammar definitions, performs some revisions on it, constructs the parse tables and lexical analyzers, which are required by the underlying parser driver to match the defined grammar and produces some output. In terms of a compiler-writer, UniCC is nothing else than a compiler to compile grammars into parsers. So this is also the reason why UniCC is used to compile its own grammar during its build.

Since the first line of UniCC was written, one of its major design goals had been to be universal, related to the parsers it outputs: UniCC is not bounded to one or a special set of programming language a UniCC-compiled parser can be written in. UniCC is *target-language independent*, which means that the parsers it outputs could be implemented in any programming language a parser template or code generator is provided for.

UniCC provides two different parser generation target approaches. The first approach is a static, build-in code-generator working with a so called *parser code template*. This code-generator is a build-in part of UniCC and allows to directly turn a UniCC grammar definition into a program written in the syntax of a particular programming language. For now, UniCC ships with support for C, C++ and Python as target languages, but more targets can easily be adapted, actually without modifying a single line of UniCCs own sourcecode.

As a total approach for target-language independency, UniCC provides a secondary way for emitting a parser. This is a parser description file. Such a file is expressed in an XML-based document and includes much information on the grammar, its symbols, the parsing states and lexical recognizers and the parser behavior as well. Using this type of output, third-party code-generators or other related software can be used to build or analyze parsers according to their specialized task using UniCC as an universal front-end.

In the upcoming and the following chapters, we only rely on the build-in code-generator. Because the C parser template, which is delivered with UniCC is the most stable and widely used one, we implement parsers in this guide in C. You can easily adapt the knowledge from here to other languages when using other target languages. The examples we will deal with are hold simple, so even those of you who have no experience in C will understand them with ease. What is needed to run the examples in this manual is an installed C or C++ compiler, e. g. gcc on Linux.

The general build order for parsers implemented using UniCC is the following, once your grammar is written and stored into a file.

- Generate the parser using UniCC. This step is nothing else than compiling the grammar into a program source code.
- Compile or run the generated parser with the particular compiler your parser is written for. This step is usually not required if your target language is interpreted, like with Python.
- Run the resulting program, either in its binary form or via the interpreter.

Using specialized code-generation tools e.g. which use the parser definition file feature may use another approach, in order they call UniCC or use a subsequent call to a generator which builds the output file for a particular programming language or solution.

Experienced programmers usually will put the above actions into a Makefile or other kind of build system, but we will run them manually for now.

The first step is the same on all platforms and with all versions of UniCC. Store the grammar you want to compile into a format-free text file, let's use *dates.par* as filename for the above grammar. To invoke UniCC from a shell or Makefile, simply type

```
unicc -w dates.par
```

If UniCC does output nothing at all, the grammar is valid, consistent and has been successfully compiled and generated without any errors and warnings.

If UniCC reports *errors*, these must be fixed. If errors are reported, this always causes that no output (respective a parser) is generated. Errors arise if UniCC comes into a situation where a valid result is not possible to generate, or the algorithm on generating the parser is initiated with missing or incomplete data. This can be e. g. a parse error in the input, where no valid grammar can be recognized, or wrong use of left-hand side items within a productions, which avoids replacing a placeholder within executable code with a valid item access actions.

Some *warnings* are normal in the daily use of UniCC. Warnings can normally be ignored, because the reference to automatically fixed grammar ambiguities or default mechanisms taking place if they where not explicitly defined. If you want to suppress all warnings, run UniCC without the `-w` or `--warnings` option. Using the `-v` or `--verbose` option, UniCC outputs some processing informations and grammar statistics.

```
$ unicc -v dates.par
UniCC version: 1.2
Parsing grammar...Done
Parser construction mode: sensitive
Goal symbol detection...Succeeded
Setting up single goal symbol...Done
Rewriting grammar...Done
Fixing precedences...Done
Computing FIRST-sets...Done
Validating rule integrity...Done
Building parse tables...Done
Terminal anomaly detection...Done
Constructing lexical analyzer...Done
Detecting default rules...Done
Code generation target: C (default)
Invoking code generator...[UniCC Standard C Template, v1.1]...Done

dates.par produced 36 states (0 errors, 0 warnings), 2 files
$
```

To get an overview about all supported command-line options, run `unicc` without any parameters. There is also a section about command-line parameters in the reference manual.

In case of the standard C parser template delivered with UniCC, above call will generate two output files, which are the program source file `dates.c` and `dates.h`, a header file, containing some definitions. This output can immediately be compiled with a suiting C compiler. It is not required to write a separate main() function because the default C target uses a predefined main()-function if no individual code for the parser's footer is specified. Note, that this feature is always target related. Especially third-party drivers and own modifications may not support this feature.

Building, compiling and playing a little bit around with the parser looks like this.

```
$ unicc -w dates.par
$ cc -o dates dates.c
$ ./dates -el

ok
August 24, 2010 "An important meeting!"

ok
23.5.2011 "Birthday of Mr. X"

ok
"Holidays!" 24.7.
line 1: syntax error on token '&eof'

ok
29.10.2010 Hello
line 1: syntax error on token 'H'

ok
line 1: syntax error on token 'e'

ok
line 1: syntax error on token 'l'

ok
line 1: syntax error on token 'l'

ok
line 1: syntax error on token 'o'

ok
line 1: syntax error on token '&eof'

ok
```

2.5. Parsing semantics

With above examples, we only created parsers that match valid and reject invalid input. Based upon the above grammar for parsing dates, it can now be simply turned it into a real compiler.

2.5.1. Using semantic actions

This example is not a compiler in terms of a programming language, but some kind of converter to compile an input date into an other output format. For this purpose, the parser is augmented with semantic operations to be performed during the parse. A semantic action is a piece of program code that is executed on a part of the parse tree when a production rule has been entirely matched. This causes an internal reduction of the rule to its left-hand side (the nonterminal it belongs to), which is then a part of another production again, or at least the goal symbol.

For this purpose, UniCC allows to store values into the different symbols used in each production definition - this includes all symbols on the right-hand side of the particular production and the left-hand side symbol (the nonterminal!) the production belongs to. Storing a value to the left-hand side means that it is taken over to the right-hand side of the next rule reduction.

Before we drift now into a too complex textual clutter, lets assign some semantic actions to the original, simple date grammar from above!

```
#whitespaces    ' \t';
#lexeme         integer;
#default action [* @@ = @1; *];

//Defining the grammar
date$           -> month:month integer:day ',' integer:year
                        [* printf( "%02d.%02d.%04d\n",
                                @day, @month, @year );
                        *]
                ;

month           -> "January"    [* @@ = 1; *]
                | "February"    [* @@ = 2; *]
                | "March"       [* @@ = 3; *]
                | "April"       [* @@ = 4; *]
                | "May"         [* @@ = 5; *]
                | "June"        [* @@ = 6; *]
                | "July"        [* @@ = 7; *]
                | "August"      [* @@ = 8; *]
                | "September"   [* @@ = 9; *]
                | "October"     [* @@ = 10; *]
                | "November"    [* @@ = 11; *]
                | "December"    [* @@ = 12; *]
                ;

integer         -> '0-9'        [* @@ = @1 - '0'; *]
                | integer '0-9' [* @@ = @1 * 10 + @2 - '0'; *]
                ;
```

That's everything required for a simple date-format compiler, parsing an input date in the format `Name-of-Month Day, Year` and compiling it into the format `Day.Month.Year`.

In comparison to the grammar-draft from above, this augmented version contains programmed actions, which define what the parser should do on the different grammatical elements during the parse, and how values are passed trough the parse tree. The following, visualized parse tree shows how the input-string `August 17, 2008` is parsed, including the semantic values stored into every node, which is an instance of a nonterminal symbol.

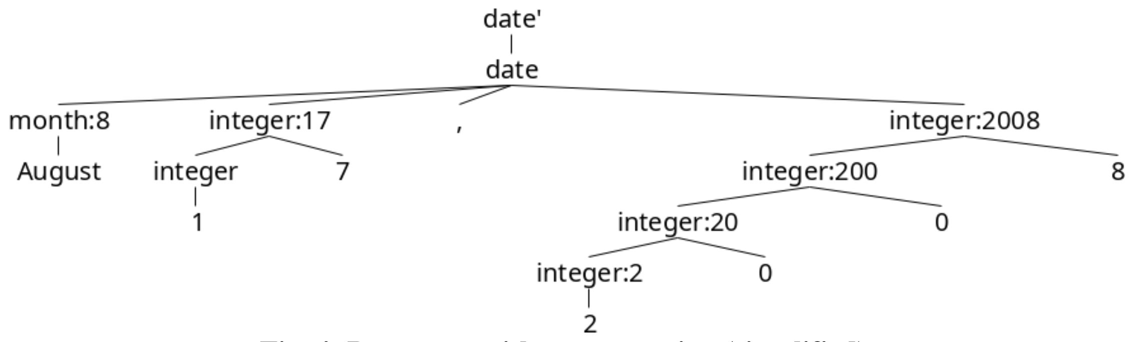


Fig. 4: Parse tree with augmentation (simplified).

Every code-segment, which is enclosed between `[*` and `*]`, is executed when the parser successfully matches a production. This is why the semantic-code invoked on a rule's reduction is even called as the *reduction action* in LR and LALR-parsers.

In such a reduction action, the compiler-writer is able to access all values of the current production and "return" values to the higher nodes of the parse tree, which are based on these semantic values. It is also possible to perform code-generation within these reduction codes, or mixed semantic checks, symbol table management, and more. These are all the things to be done in a real compiler, and go beyond the task of parsing.

For all elements of the reduced production, the right-hand side, values are accessed using an `@` character followed by the number of the desired tokens position, which begins with token number 1 from the left. It is also possible to assign meaningful names to right-hand side symbols, simply by separating the symbol from the identifying name using a colon `:`. This is done in the production

```
date$           -> month integer:day ',' integer:year
                        [* printf( "%02d.%02d.%04d\n",
                                @day, @month, @year );
                        *]
                ;
```

from above, so we are able to access `month`, `day` and `year` over their meaningful names instead of their position offsets. There is no label specified for `month` because UniCC automatically associates the nonterminal's name with an right-hand side value specifier. This default can be overridden by defining it manually, as its done with `day` and `year`.

By using the position offsets, the same result could be reached by writing

```
date$           -> month integer ',' integer
                        [* printf( "%02d.%02d.%04d\n",
                                @2, @1, @4 );
                        *]
                ;
```

If names are given, the symbols of the right-hand side can be accesses both via offset or by name. The advantage of using identifying names is that no changes in the semantic action code is required if the production symbol order changes, e. g. when a new separation symbol is introduced between two symbols.

To assign a value to the left-hand side, which is the upper node in the parse tree, an `@@` placeholder is used. `@@` never contains a value (it is initialized to zero), and is only used to pass a result from one successfully recognized production to another (at the time the production is reduced) uncompleted, upper lying production calling the current production's nonterminal it is associated with. Here, this value can be accessed again by a reduction action to compute or output a result from it.

All values stored to nonterminals are written there by reduction actions. The atomic values from the input-stream, the terminals, are the base for these values, and obtain their values directly from the input, or an lexical analyzer, which is introduced later.

In the standard C parser template delivered with UniCC, every character terminal gets the character-code of the character it matches in the input. Therefore, the semantic value that is constructed using nonterminal `integer` is

```
integer         -> '0-9'         [* @@ = @1 - '0'; *]
                | integer '0-9' [* @@ = @1 * 10 + @2 - '0'; *]
                ;
```

to result in a true, decimal number that is stored into memory as an integer data type.

This looks a little bit tricky for those who are not familiar with C. For the parse of the number "17", for example, the first scanned digit (which can be a digit between 0 and 9) in the reduction code of the first production only exists in its character-coded form from the input, which is code 0x31 (decimal 049) for the digit "1" in the ASCII character map. To easily get an integer number 1 from this coded representation of the character, we have to subtract the value of the character-code of digit "0", which is 0x30 (decimal 048), so the operation 0x31-0x30 returns the correct value 0x1, which is then passed to the left-hand side.

In the second step of our parse of the input sequence "17", we first have to multiply the first digit by its base, 10 (to derive 10 from the 1), and then perform the same procedure as in the first production, but with the difference that is will be added to this 10. The result is 17, as a true, decimal number to be stored to an **int** data typed variable.

This latter step can then be performed for every digit in the integer; The already parsed value is multiplied with 10 (to be moved one digit to the left) again and then added by the next digit. Note the recursion of nonterminal `integer`: It can parse one single digit or a chain of digits as one unit.

As a beginner, you can now say that coding parsers in UniCC is hard to understand. Well, you might be a little right. But this is only the first impression on it. The more you learn about the techniques, the more practical experience you will have, things will make sense, and the parse trees you want to climb will grow in your mind. Don't give up, even if you feel so - it wouldn't be worthwhile!

If you're already familiar with parsing, this "low-level"-looking way of extracting atomic integer numbers from the input will look unconventional to most of you. This is done because UniCC allows (but not relies) to perform lexical analysis by the grammar rather than an upstreamed lexer. To let the reader become more familiar with grammar definitions in UniCC first, this approach was chosen here. All possibilities and facilities on lexical analysis will be introduced later on. The advantage of this approach is, that full LALR power is available on this character-based parsing method, so even recursive "lexemes" can be parsed as true context-free grammars, not only regular ones. Terminal symbol based on regular-expressions are even possible in UniCC, but this will be discussed later on.

2.5.2. Precedences

In the last chapters, we did some experience on how grammars are written and how a simple parser is furnished and attributed with semantic actions. A real compiler, compiling a programming language (or similar!) into another representation (e.g. assembly code) is nothing else than this - but much more effort in writing the grammar and especially the semantic code is required. To build grammars for such higher and complex targets, you need experience, time and the most important thing: Patience.

We will now write a grammar that parses mathematical expressions, a four function calculator. Nearly every high-level programming language supports mathematical expressions, so why not to begin here?

Mathematical expressions have special demands to their grammar. Multiplicative operations (multiplication and division) take precedence over additive operations (addition and subtraction), but this order can be broken by bracketing terms to take same precedence as operands.

Let's first implement a grammar for additive calculations. Simple terms of the syntax

```
integer + integer
integer - integer
```

shall be valid and parsed. A first draft of the new grammar would be

```
//Some grammar-related directives
#whitespaces  ' \t';
#lexeme       integer;

//Defining the grammar
expression$   -> integer '+' integer
              | integer '-' integer
              ;

integer       -> '0-9'
              | integer '0-9'
              ;

//End of definition
```

Compiled and run, this works for simple terms with always two operands, but terms with just one single operand, or even terms with three operand to be added or subtracted, are punished with a syntax error. Extending nonterminal `expression` to read as

```
expression$   -> expression '+' integer
              | expression '-' integer
              | integer
              ;
```

will allow for a recursion, which is the correct method to enable terms with variable length.

But how to add precedence now to this grammar, enabling multiplicative operators? There are two methods in UniCC to make this possible. The first is obvious, and the second is for lazy people. We select the obvious method first, requiring more efforts in writing the grammar and even more parse-states in the resulting parser. The lazy method is described below, when the expression parser is finished.

Our obvious approach is simple: Why not copy the definition of nonterminal `expression` to fit the demands of operators with higher precedence, and then call this higher-level nonterminal from `expression`? The resulting grammar of this idea is

```
//Some grammar-related directives
#whitespaces  ' \t';
#lexeme       integer;

//Defining the grammar
expression$   -> expression '+' term
              | expression '-' term
              | term
              ;

term          -> term '*' integer
              | term '/' integer
              | integer
              ;

integer       -> '0-9'
              | integer '0-9'
              ;

//End of definition
```

The new nonterminal `term` matches this idea, and is called by `expression` where `integer` was called before. The compiled and run version of this grammar allows to enter any desired expression with mixed additive and multiplicative operators. One demand is missing: Overwriting precedence rules with bracketing. Because brackets may appear in the same positions where our operand-nonterminal `integer` currently appears, a replacement for `integer` must be added to allow for both ways in this uppermost precedence level.

Rewriting the grammar with a new decider between `integer` and a call to a new `expression` enclosed with brackets results in a terminal `factor`, which is then called by `term`.

```
factor        -> integer
              | '(' expression ')'
              ;
```

Finally, we have the complete grammar to parse expressions the correct way.

The only disadvantage: We still see no results, again! So there's just a little bit of augmentation required to this grammar to make it a working expression calculator.

```
//Some grammar-related directives
#whitespaces     ' \t';
#lexeme          integer;
#default action  [* @@ = @1; *];


//Defining the grammar
calculator$   -> expression           [* printf( "= %d\n",
                                              @expression ); *]
              ;

expression    -> expression '+' term  [* @@ = @1 + @3; *]
              | expression '-' term   [* @@ = @1 - @3; *]
              | term
              ;
```

```
term          -> term '*' factor      [* @@ = @1 * @3; *]
              | term '/' factor       [* @@ = @1 / @3; *]
              | factor
              ;

factor        -> integer
              | '(' expression ')'    [* @@ = @expression; *]
              ;

integer       -> '0-9'                [* @@ = @1 - '0'; *]
              | integer '0-9'         [* @@ = @integer * 10 +
                                              @2 - '0'; *]
              ;

//End of definition
```

That's the complete program code which calculates any desired expression for you the correct way! Simple, isn't it? Maybe you recognized the line with

```
#default action [* @@ = @1; *];
```

which was already present in our first example using semantic actions. This parser directive is required to define a default action that should be performed at an (nonempty productions) reduction code if no code has been provided by the grammar. For example in

```
factor        -> integer
              | '(' expression ')'    [* @@ = @2; *]
              ;
```

The first production uses this default action code to assign the return-value of the `integer` nonterminal automatically to the value of `factor`. Just keep this in mind, if you feel that your parser is loosing values because you'd forgotten to add this directive to your grammar. Because UniCC is a target-language independent parser generator, it was decided to let the grammar designer choose the way of how values are passed by default.

Some sentences above, we mentioned that there are two ways of implementing precedences within UniCC grammar definitions. We first chose to implement the obvious method as described above, but there is also a version for those lazy people among you! "Lazy" means, that you write lesser grammar code but take the same precedences effect as you will get with writing an obvious grammars as the one above. Another advantage is, that UniCC produces lesser states for the same parsing behavior - even the same semantic actions can be used.

The key elements for lazy grammar writers are the parser configuration directives `#left`, `#right` and `#nonassoc`, whereas for our case of the expression language, we only require the `#left` directive. These directives furnish grammar symbols with precedence- and associativity-weighting to influence the parse table generator and to resolve parse table conflicts, which come up with ambiguous grammars.

Such an grammar would be the following, when we try to compile it.

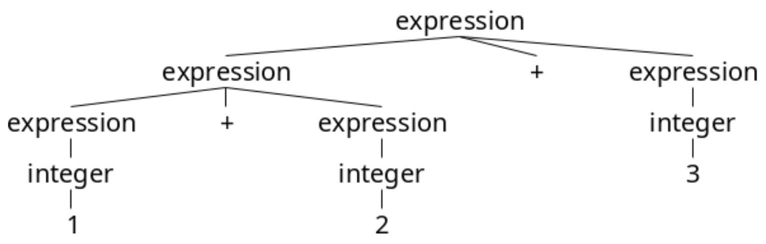
```
//Some grammar-related directives
#whitespaces    ' \t';
#lexeme         integer;

//Defining the grammar
calculator$     -> expression
```

```
                ;

expression      -> expression '+' expression
                | expression '-' expression
                | expression '*' expression
                | expression '/' expression
                | '(' expression ')'
                | integer
                ;

integer         -> '0-9'
                | integer '0-9'
                ;

//End of definition
```

When this grammar is fed to UniCC using the `-w` command line switch, we get many lines with the following warnings:

```
$ unicc -w conflicts.par
unicc: warning: state 16: Shift-reduce conflict on lookahead: '+'
    (1) expression -> expression .+' expression
    (1) expression -> expression +' expression .      { &eof '+' '-' '*' '/' ')' }
    (2) expression -> expression .-' expression
    (3) expression -> expression .*' expression
    (4) expression -> expression ./' expression
unicc: warning: state 16: Shift-reduce conflict on lookahead: '-'
    (1) expression -> expression .+' expression
    (1) expression -> expression +' expression .      { &eof '+' '-' '*' '/' ')' }
    (2) expression -> expression .-' expression
    (3) expression -> expression .*' expression
    (4) expression -> expression ./' expression
unicc: warning: state 16: Shift-reduce conflict on lookahead: '*'
    (1) expression -> expression .+' expression
    (1) expression -> expression +' expression .      { &eof '+' '-' '*' '/' ')' }
    (2) expression -> expression .-' expression
    (3) expression -> expression .*' expression
    (4) expression -> expression ./' expression
unicc: warning: state 16: Shift-reduce conflict on lookahead: '/'
    (1) expression -> expression .+' expression
    (1) expression -> expression +' expression .      { &eof '+' '-' '*' '/' ')' }
    (2) expression -> expression .-' expression
    (3) expression -> expression .*' expression
    (4) expression -> expression ./' expression
unicc: warning: state 17: Shift-reduce conflict on lookahead: '+'
    (1) expression -> expression .+' expression
    (2) expression -> expression .-' expression
    (2) expression -> expression -' expression .      { &eof '+' '-' '*' '/' ')' }
    (3) expression -> expression .*' expression
    (4) expression -> expression ./' expression
unicc: warning: state 17: Shift-reduce conflict on lookahead: '-'
    (1) expression -> expression .+' expression
    (2) expression -> expression .-' expression
    (2) expression -> expression -' expression .      { &eof '+' '-' '*' '/' ')' }
    (3) expression -> expression .*' expression
    (4) expression -> expression ./' expression
unicc: warning: state 17: Shift-reduce conflict on lookahead: '*'
    (1) expression -> expression .+' expression
    (2) expression -> expression .-' expression
    (2) expression -> expression -' expression .      { &eof '+' '-' '*' '/' ')' }
    (3) expression -> expression .*' expression
    (4) expression -> expression ./' expression
unicc: warning: state 17: Shift-reduce conflict on lookahead: '/'
    (1) expression -> expression .+' expression
    (2) expression -> expression .-' expression
    (2) expression -> expression -' expression .      { &eof '+' '-' '*' '/' ')' }
    (3) expression -> expression .*' expression
    (4) expression -> expression ./' expression
unicc: warning: state 18: Shift-reduce conflict on lookahead: '+'
    (1) expression -> expression .+' expression
    (2) expression -> expression .-' expression
    (3) expression -> expression .*' expression
    (3) expression -> expression *' expression .      { &eof '+' '-' '*' '/' ')' }
    (4) expression -> expression ./' expression
unicc: warning: state 18: Shift-reduce conflict on lookahead: '-'
    (1) expression -> expression .+' expression
    (2) expression -> expression .-' expression
    (3) expression -> expression .*' expression
    (3) expression -> expression *' expression .      { &eof '+' '-' '*' '/' ')' }
    (4) expression -> expression ./' expression
unicc: warning: state 18: Shift-reduce conflict on lookahead: '*'
    (1) expression -> expression .+' expression
    (2) expression -> expression .-' expression
    (3) expression -> expression .*' expression
    (3) expression -> expression *' expression .      { &eof '+' '-' '*' '/' ')' }
    (4) expression -> expression ./' expression
unicc: warning: state 18: Shift-reduce conflict on lookahead: '/'
    (1) expression -> expression .+' expression
    (2) expression -> expression .-' expression
    (3) expression -> expression .*' expression
    (3) expression -> expression *' expression .      { &eof '+' '-' '*' '/' ')' }
    (4) expression -> expression ./' expression
unicc: warning: state 19: Shift-reduce conflict on lookahead: '+'
    (1) expression -> expression .+' expression
    (2) expression -> expression .-' expression
    (3) expression -> expression .*' expression
    (4) expression -> expression ./' expression
    (4) expression -> expression /' expression .      { &eof '+' '-' '*' '/' ')' }
unicc: warning: state 19: Shift-reduce conflict on lookahead: '-'
    (1) expression -> expression .+' expression
    (2) expression -> expression .-' expression
    (3) expression -> expression .*' expression
    (4) expression -> expression ./' expression
    (4) expression -> expression /' expression .      { &eof '+' '-' '*' '/' ')' }
unicc: warning: state 19: Shift-reduce conflict on lookahead: '*'
    (1) expression -> expression .+' expression
    (2) expression -> expression .-' expression
    (3) expression -> expression .*' expression
    (4) expression -> expression ./' expression
    (4) expression -> expression /' expression .      { &eof '+' '-' '*' '/' ')' }
unicc: warning: state 19: Shift-reduce conflict on lookahead: '/'
    (1) expression -> expression .+' expression
    (2) expression -> expression .-' expression
    (3) expression -> expression .*' expression
    (4) expression -> expression ./' expression
    (4) expression -> expression /' expression .      { &eof '+' '-' '*' '/' ')' }
$
```

This is caused due the ambiguity of the grammar, which comes up in the nonterminal definition of `expression`. This ambiguity can be visualized when analyzing the expression *1+2+3*. Because our grammar defines each operator production as `expression operator expression` and `expression` itself can exists of such a composition again, the grammar allows for parsing it as *(1+2)+3* and as *1+(2+3)*, which results in completely different parse trees. It is correct, that the order of the operators in additions doesn't care, but what about *1+2*3*? Here, again, the grammar from above allows both for *(1+2)*3* and *1+(2*3)*, but only latter one is a valid result we want to accept.



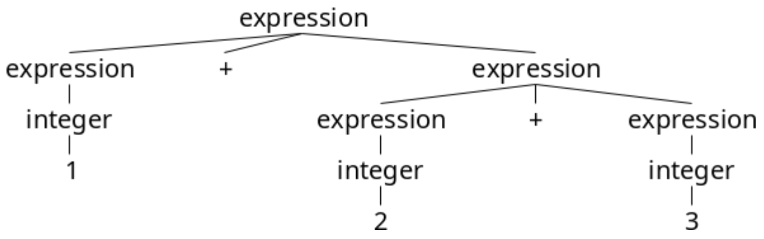


Fig. 5: Two different parse trees are yielding in ambiguous grammars (simplified).

For these cases, precedence and associativity assignments to terminals can be used to define the correct way of how UniCC should handle these conflicts.

Using a `#left` directive, we give terminal symbols a left-sided associativity, which means that the left expression of the terminal is resolved first. When we do this to addition and subtraction, these operators will resolve the left expression first, so the parse tree grows left-derivative. The input string *1+2+3* will then be parsed as *(1+2)+3*, and *1+2+3+4* will be parsed as *((1+2)+3)+4*. Because subtraction is on the same precedence level as addition, *1+2-3* will correctly be parsed as *(1+2)-3*.

But what about multiplication and division? When we assign the same, left-bounded associativity to these operators, *1+2*3* will be incorrectly parsed as *(1+2)*3*. So another, higher precedence level than to addition and subtraction must be added, which we do with yet another call to the `#left`-directive. As deeper a precedence-directive in the grammar occurs below other `#left`, `#right` and `#nonassoc` directives, so higher the precedence will be assigned.

When this secondary precedence and associativity assignment is done, the grammar will behave the correct way, even for expressions like *1+2/3+4+5*6-7* - which will then correctly be parsed as *(((1+(2/3))+4)+(5*6))-7*.

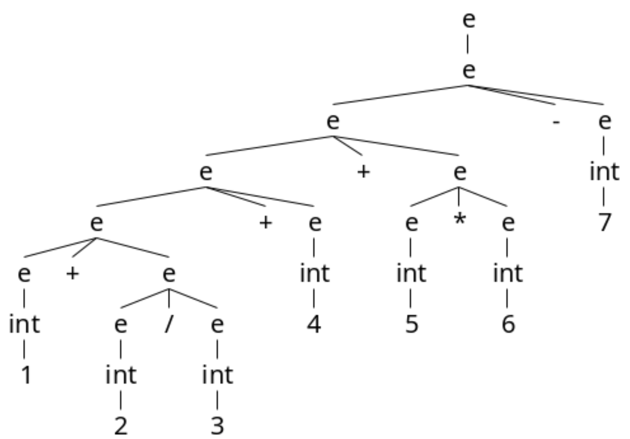


Fig. 6: The unambiguous parse tree of "1+2/3+4+5*6-7" (simplified). Symbol names had been shortened.

Adding this precedence directives to above grammar, we'll get its unambiguous pendant.

```
//Some grammar-related directives
#!language      "C";
#whitespaces    ' \t';
#lexeme         integer;
#default action [* @@ = @1; *];

#left           '+' '-';
```

```
#left           '*' '/';

//Defining the grammar
calculator$     -> expression                [* printf( "= %d\n",
                                                 @expression ); *]
                ;

expression      -> expression '+' expression [* @@ = @1 + @3; *]
                | expression '-' expression  [* @@ = @1 - @3; *]
                | expression '*' expression  [* @@ = @1 * @3; *]
                | expression '/' expression  [* @@ = @1 / @3; *]
                | '(' expression ')'         [* @@ = @2; *]
                | integer
                ;

integer         -> '0-9'                     [* @@ = @1 - '0'; *]
                | integer '0-9'              [* @@ = @integer * 10 +
                                                    @2 - '0'; *]
                ;

//End of definition
```

This will parse and calculate the correct values, including all precedence relations.

2.6. Implementing a compiler

Now that we have learned to know how to implement simple expressions using UniCC, it's time to implement a real-world example of a toy programming language. This toy language is called eXample Programming Language, shortened XPL, and is a C-styled language.

The language should match the following criteria:

- Dynamically typed language, supporting integer, floating point and string values
- Support of arithmetic and conditional expressions
- Simple control structures for conditionals and iterations
- Nested calls to build-in functions with variable arguments
- The functions should provide simple data manipulation routines and input/output facilities

We first don't care about how to execute this programming language. First it is only a draft for a syntax describing the language grammar. A simple example to rudely define the syntax of XPL shall be the 99-bottles-of-beer programming example.

```
if( ( bottles = prompt( "Enter number of bottles [default=99]" ) ) == "" )
    bottles = 99;

if( integer( bottles ) <= 0 )
{
    print( "Sorry, but the input '" + bottles + "' is invalid." );
    exit( 1 );
}

while( bottles > 0 )
{
    if( bottles > 1 )
        print( bottles + " bottles of beer on the wall, " +
                bottles + " bottles of beer." );
    else
        print( "One bottle of beer on the wall, one bottle of beer." );

    print( "Take one down, pass it around." );

    if( ( bottles = bottles - 1 ) == 0 )
        print( "No more bottles of beer on the wall." );
    else if( bottles == 1 )
        print( "One more bottle of beer on the wall." );
    else
        print( bottles + " more bottles of beer on the wall." );
}
```

2.6.1. Getting the sources

All the source code presented in the following sections is not covered in detail, but it is a working example for a programming language implemented entirely using UniCC. To obtain the latest version of the XPL source code presented here, clone the source repository:

```
git clone https://github.com/phorward/xpl.git
```

The source code printed here can also be found in the appendix.

2.6.2. Drafting the grammar

The plain grammar of the language is described below. This intermixes already known grammar constructs like lexemes with some new features that had not been covered for now.

```
//Meta information
#parser             "XPL";
#description        "eXample Programming Language (Draft)";
#copyright          "In the public domain, 2011";
#version            "0.1";
#prefix             "xpl";

//Precedence and associativity
#left               "=";

#left               "=="
                    "!="
                    "<="
                    ">="
                    '>'
                    '<'
                    ;

#left               '+'
                    '-'
                    ;

#left               '*'
                    '/'
                    ;

//Regular expressions
@string             '"' !'"'* '"'
                    ;

@identifier         'A-Za-z_' 'A-Za-z0-9_'*
                    ;

//Lexemes
#lexeme             real
                    ;

real               ->       real_integer '.' real_fraction
                    |       real_integer '.'?
                    |       '.' real_fraction
                    ;

real_integer        ->      real_integer '0-9'
                    |       '0-9'
                    ;

real_fraction       ->      real_fraction '0-9'
                    |       '0-9'
                    ;

//Whitespace grammar construct
#whitespaces        whitespace
                    ;

whitespace          ->      ' \r\n\t'+
                    |       "//" !'\n'* '\n'
                    ;

//Goal symbol
program$            ->      statement*
                    ;

statement           ->      "if" '(' expression ')' statement
                    |       "if" '(' expression ')' statement
                                "else" statement
                    |       "while" '(' expression ')' statement
                    |       '{' statement* '}'
                    |       expression ';'
                    |       ';'
                    ;

expression          ->      variable "=" expression
                    |       expression "==" expression
                    |       expression "!=" expression
                    |       expression '>' expression
                    |       expression '<' expression
                    |       expression "<=" expression
                    |       expression ">=" expression
                    |       expression '+' expression
                    |       expression '-' expression
                    |       expression '*' expression
                    |       expression '/' expression
                    |       '-' expression              #precedence '*'
                    |       '(' expression ')'
                    |       real
                    |       @string
                    |       variable
                    |       function '(' parameter_list? ')'
                    ;

parameter_list      ->      parameter_list ',' expression
                    |       expression
                    ;

variable
function            ->      @identifier
                    ;
```

2.6.2.1. Terminals based on regular expressions

The first, new feature used here is the definition block

```
//Regular expressions
@string                 '"' !'"'* '"'
                        ;

@identifier             'A-Za-z_' 'A-Za-z0-9_'*
                        ;
```

These two blocks introduce two terminal symbols, `@string` and `@identifier`. These terminal symbols are made of regular expressions. Regular expressions are some kind of formal language to describe a way in which characters or character-classes, words and sentences may appear in the input to match the corresponding terminal symbol and give this string sequence a semantic meaning. All terminal symbols in UniCC, the already known character-classes and strings, are some kind of regular expression. These terminals can be defined where they are used, but regular expressions may be defined that complex and powerful, that they require their own definition block.

The first of the above regular expressions describe a terminal `@string`, that begins with a double quotation mark `'"'`, a sequence of zero or multiple characters except a quotation mark (`!'"'*`), and a closing double quotation mark `'"'`. The second regular expression describes a terminal `@identifier` which is made of at least one character that may be in the upper- and lower-case letters or underscore (`'A-Za-z_`), and zero or multiple characters that are made up of upper-/lower-case letters, digits and underscores. This regular expression pattern matches for strings like `a`, `Hello`, `_test` or even `i22`, but not for `22` on its own, which should be recognized as an integer number by the grammar. More about the use of regular expressions and their making up is covered in the UniCC reference guide.

2.6.2.2. Overriding production precedence

The second new feature that is used here but has not covered yet is the production

```
'-' expression              #precedence '*'
```

This special `#precedence` directive sets a higher precedence level to the production to overwrite its default precedence that is associated with the subtraction-operator (`-`) to the precedence of the multiplication-operator `*`. This configuration is done to allow for an unary minus. Expressions like "-2*4" would be parsed as *-(2*4)* if this configuration is not done - a wrong result, because subtraction is below multiplication in the operator precedence order. Setting the production precedence to the level of the multiplication-operator (or higher), it is parsed the correct way as *(-2)*4*.

2.6.2.3. Multiple left-hand sides

The last new feature used in this grammar is the nonterminal definition

```
variable
function                ->      @identifier
                        ;
```

This definition associates one production with two nonterminals, **variable** and **function**. This construct allows to determine the final type of the nonterminal at runtime, by using UniCC's feature of semantic nonterminal determination. This feature allows to first check within the semantic code if an identifier is the name of a function or the name of a variable, and then return the appropriate nonterminal symbol. This feature could be seen as a dynamic change of the way how the grammar is recognized at runtime.

2.6.2.4. The "dangling else" problem

When compiling this grammar, we will be concerned by this shift-reduce conflict warning:

```
unicc: warning: state 81: Shift-reduce conflict on lookahead: "else"
    (22) statement -> if' (' expression )' statement .      { &eof "if" "while" '{' ';' '-' '(' '.' @st
    (23) statement -> if' (' expression )' statement .else' statement
```

This problem is called "dangling else", and occurs in many LALR-/LR-based grammars. The nature of this problem can be obtained from the following pseudo-code examples.

```
if( expression )
        statement;
else
        statement;
```

The above syntax is clear and not ambiguous. But what happens when we want to submit

```
if( expression )
        if( expression )
                statement;
else
        statement;
```

Now, the **else** is visually part of the first **if**, but the parser would take it as the **else** part of the second **if**. Both ways could be valid. UniCC notices this situation, that the grammar is ambigous, so the above warning is printed. This problem is also found in grammars for wide-spread languages like C or Java. The solution is to put the second **if** into a new statement block, and everything is clear. The warning can be ignored in this case.

```
if( expression )
{
        if( expression )
                statement;
}
else
        statement;
```

Except of this warning, the draft version of the XPL parser already runs very well in the UniCC Standard C Parser Template's test mode for some simple expressions. Only function calls don't work, because the semantically determined nonterminal selection is not implemented yet. The parser now always detects a variable and throws a parse error when the opening bracket for the function call are stated, and a variable followed by an opening bracket is not valid. This problem will be solved after adding some semantic determination code soon.

2.6.3. The virtual machine

The topic of intermediate code generation performed by a compiler to build a binary program or directly run it is that huge and complex, that it can't be covered in this manual or even in huger compiler textbooks. You only have to know, that every compiler creates some kind of intermediate representation of its input. In most cases some kind of a more or less abstract tree structure (abstract syntax tree) representing semantics and nesting, but there are also compilers that compile into an intermediate language that can be used for further optimization or platform-dependent code generation. Both methods are used in many of today's existing compiler implementations.

An abstract syntax tree (AST) is the representation of a parse tree where atomic syntactical elements are hidden, and only the semantic information (e.g. a literal from the code, a variable address or a node defining a conditional statement) and its nested structure is clearly defined. Such an AST-representation could already be used for a direct interpretation of the input. In C, for example, such an abstract syntax tree could be expressed as a huge union data structure, providing several structures for every language element. There could be one structure describing a value, one structure describing an assignment, one structure describing an IF-construct, and so on. The result is a tree structure that is made-up of many of such nodes, links and leafs that can be easily interpreted by a recursively executed virtual machine. Although this method is very simple to implement, it still requires a lot of coding efforts, and saving this tree to a file is not possible without the use of some nested structuring. The benefit of this method is, that no backpatching is required, because the structure is a logical tree.

The generation of intermediate code could also be done from an abstract syntax tree, but it can also directly be done out of the parser (in simple languages), where the parser serves as the immediate representation of the abstract syntax tree during input recognition. Intermediate code, in turn, can be established on various paradigms. 3-address-code is very useful to generate optimized code for register-machines. 1-address-code can be used for stack-based machines, like the Java Virtual Machine.

This is also the approach we want to implement in XPL. We define a virtual execution machine for a stack-based virtual machine, interpreting 1-address-code. This simple virtual machine remains entirely independent from the overlying language (here XPL). It can also be seen as a target-platform for any other kind of language, e.g. a simple BASIC-dialect.

2.6.3.1. xpl_value - a dynamic value object

The stack-elements of the simple virtual machine are described in a structure called `xpl_value`, defined in `xpl.h`.

```
/* Virtual machine values */
typedef enum
{
    XPL_NULLVAL,                            /* Nothing/Undefined */
    XPL_INTEGERVAL,                         /* Integer type */
    XPL_FLOATVAL,                           /* Float type */
    XPL_STRINGVAL                           /* String type */
} xpl_datatype;

typedef struct
{
    xpl_datatype    type;                   /* Value type */

    union
    {
        int         i;
        float       f;
        char*       s;
    } value;                                /* Value storage union */

    char*           strval;                 /* Temporary string value
                                                representation pointer */
} xpl_value;
```

To work with these value objects, a functions library is implemented in `xpl.value.c`.

```
#include "xpl.h"

/* Value Objects */

xpl_value* xpl_value_create( void )
{
    return (xpl_value*)xpl_malloc( (char*)NULL, sizeof( xpl_value ) );
}

xpl_value* xpl_value_create_integer( int i )
{
    xpl_value*  val;

    val = xpl_value_create();
    xpl_value_set_integer( val, i );

    return val;
}

xpl_value* xpl_value_create_float( float f )
{
    xpl_value*  val;

    val = xpl_value_create();
    xpl_value_set_float( val, f );

    return val;
}

xpl_value* xpl_value_create_string( char* s, short duplicate )
{
    xpl_value*  val;

    val = xpl_value_create();
    xpl_value_set_string( val, duplicate ? xpl_strdup( s ) : s );

    return val;
}

void xpl_value_free( xpl_value* val )
{
    if( !val )
        return;

    xpl_value_reset( val );
    free( val );
}

void xpl_value_reset( xpl_value* val )
{
    if( val->type == XPL_STRINGVAL && val->value.s )
```

```
        free( val->value.s );

    val->strval = xpl_free( val->strval );

    memset( val, 0, sizeof( xpl_value ) );
    val->type = XPL_NULLVAL;
}

xpl_value* xpl_value_dup( xpl_value* val )
{
    xpl_value*  dup;

    dup = xpl_value_create();

    if( !val )
        return dup;

    memcpy( dup, val, sizeof( xpl_value ) );

    dup->strval = (char*)NULL;

    if( dup->type == XPL_STRINGVAL )
        dup->value.s = xpl_strdup( dup->value.s );

    return dup;
}

void xpl_value_set_integer( xpl_value* val, int i )
{
    xpl_value_reset( val );
    val->type = XPL_INTEGERVAL;
    val->value.i = i;
}

void xpl_value_set_float( xpl_value* val, float f )
{
    xpl_value_reset( val );
    val->type = XPL_FLOATVAL;
    val->value.f = f;
}

void xpl_value_set_string( xpl_value* val, char* s )
{
    xpl_value_reset( val );
    val->type = XPL_STRINGVAL;
    val->value.s = s;
}

int xpl_value_get_integer( xpl_value* val )
{
    switch( val->type )
    {
        case XPL_INTEGERVAL:
            return val->value.i;
        case XPL_FLOATVAL:
            return (int)val->value.f;
        case XPL_STRINGVAL:
            return atoi( val->value.s );

        default:
            break;
    }
    return 0;
}

float xpl_value_get_float( xpl_value* val )
{
    switch( val->type )
    {
        case XPL_INTEGERVAL:
            return (float)val->value.i;
        case XPL_FLOATVAL:
            return val->value.f;
        case XPL_STRINGVAL:
            return (float)atof( val->value.s );

        default:
            break;
    }

    return 0.0;
}

char* xpl_value_get_string( xpl_value* val )
{
    char    buf     [ 128 + 1 ];
    char*   p;

    val->strval = xpl_free( val->strval );

    switch( val->type )
    {
        case XPL_INTEGERVAL:
            sprintf( buf, "%d", val->value.i );
            val->strval = xpl_strdup( buf );
            return val->strval;
        case XPL_FLOATVAL:
            sprintf( buf, "%f", val->value.f );

            /* Remove trailing zeros to make values look nicer */
            for( p = buf + strlen( buf ) - 1; p > buf; p-- )
            {
                if( *p == '.' )
                {
                    *p = '\0';
                    break;
                }
                else if( *p != '0' )
                    break;

                *p = '\0';
            }

            val->strval = xpl_strdup( buf );
            return val->strval;
        case XPL_STRINGVAL:
            return val->value.s;

        default:
            break;
    }

    return "";
```

```
}
```

2.6.3.2. Defining a code set

The internal assembly language for our virtual machine consists of an opcode, defining the operation to be performed, and an integer parameter that is only used by some operations as index reference. The following table describes the opcodes, their meaning and the use of the parameter.

Opcode	Operation	Parameter
NOP	Does nothing.	-
CAL	Call a built-in function. Expects the number of arguments as first integer item on the stack, then the arguments. Arguments will be dropped off the stack after the function was called, and the function's return value is pushed.	Function-Index
LIT	Load a literal value. Literal values are hold as `xpl_value`-objects within a `xpl_program` structure. For the literal load, the literal value is duplicated and pushed onto the stack.	Literal-Index
LOD	Load a variable content. Variables are hold in a `xpl_runtime` structure during runtime. For the variable load, the variable's `xpl_value`-object is duplicated and pushed onto the stack.	Variable-Index
STO	Stores the value on top of the stack into a variable. The variable's old value will be deleted.	Variable-Index
DUP	Duplicates current top of stack value and pushes it.	-
DRP	Drop current top of stack value and clear its memory.	-
JMP	Change instruction pointer, jump to address.	Code-Address
JPC	Conditionally change instruction pointer if value of top of stack is false. Jump to address if false. The top if stack item will be dropped.	Code-Address
EQU	Check for equality of next two stack operands, pushes integer 1 = equal, 0 = not-equal	-
NEQ	Check for non-equality of next two stack operands, pushes integer 1 = non-equal, 0 = equal	-
LOT	Take two operands off the stack, check if left operand is lower than right operand, pushes integer 1 = lower, 0 = greater or equal	-
LEQ	Take two operands off the stack, check if left operand is lower or equal than right operand, pushes integer 1 = lower or equal, 0 = greater	-
GRT	Take two operands off the stack, check if left operand is greater than right operand, pushes integer 1 = greater, 0 = lower or equal	-
GEQ	Take two operands off the stack, check if left operand is grater or equal than right operand, pushes integer 1 = greater or equal, 0 = lower	-
ADD	Take two operands off the stack, perform addition and push result onto the stack. In case of string-operands, perform string-conversion of both operands, concatenate and push the concatenated string onto the stack.	-
SUB	Take two operands off the stack, perform subtraction and push the result onto the stack. Strings will be converted to integer.	-
MUL	Take two operands off the stack, perform multiplication and push the result onto the stack. Strings will be converted to integer.	-
DIV	Take two operands off the stack, perform division and push the result onto the stack. Strings will be converted to integer.	-

Table 1: The virtual machine opcodes.

Defining this in C as enum `xpl_op` and structure `xpl_cmd` results in the following code.

```
/* Virtual machine opcodes */
typedef enum
{
    XPL_NOP,                                /* No operation */

    XPL_CAL,                                /* Call function */
    XPL_LIT,                                /* Load literal value */
    XPL_LOD,                                /* Load value from variable */
    XPL_STO,                                /* Store value into variable */
    XPL_DUP,                                /* Duplicate stack item */
    XPL_DRP,                                /* Drop stack item */
    XPL_JMP,                                /* Jump to address */
    XPL_JPC,                                /* Jump to address if false */

    XPL_EQU,                                /* Check for equal */
    XPL_NEQ,                                /* Check for not-equal */
    XPL_LOT,                                /* Check for lower-than */
    XPL_LEQ,                                /* Check for lower-equal */
    XPL_GRT,                                /* Check for greater-than */
    XPL_GEQ,                                /* Check for greater-equal */

    XPL_ADD,                                /* Add or join two values */
    XPL_SUB,                                /* Subtract two values */
    XPL_MUL,                                /* Multiply two values */
    XPL_DIV                                 /* Divide two values */
} xpl_op;

/* Command description */
typedef struct
{
    xpl_op      op;
    int         param;
} xpl_cmd;
```

2.6.3.3. xpl_program - representation of a compiled program

To provide a data representation of a compiled program, XPL requires a structure holding a symbol table for variables, the used literals and the program code. All information in this structure should remain untouched at program runtime, and is only build-up by the compiler.

```
/* Program structure */
typedef struct
{
    xpl_cmd*    program;                    /* Virtual machine program */
    int         program_cnt;                /* Numer of elements in program */

    xpl_value** literals;                   /* Array of literal objects */
    int         literals_cnt;               /* Number of elements in literals */

    char**      variables;                  /* The variable symbol table. The
                                                index of the variable name
                                                represents the address in the
                                                variables-member of the
                                                runtime data structure.
                                            */
    int         variables_cnt;              /* Number of elements
                                                in variables */

} xpl_program;
```

To build-up the program structure, we also require some modification functions in `xpl.program.c` and a function `xpl_emit()` to emit the virtual machine code.

```
#include "xpl.h"

/* Program structure handling */
int xpl_get_variable( xpl_program* prog, char* name )
{
    int     i;

    /* A function name with the same identifier may not exist! */
    if( xpl_get_function( name ) > -1 )
        return -1;

    /* Try to find variable index */
    for( i = 0; i < prog->variables_cnt; i++ )
        if( strcmp( prog->variables[ i ], name ) == 0 )
            return i;

    /* Else, eventually allocate memory for new variables */
    if( ( i % XPL_MALLOCSTEP ) == 0 )
    {
        prog->variables = (char**)xpl_malloc(
                            (char*)prog->variables, ( i + XPL_MALLOCSTEP )
                                                * sizeof( char** ) );
    }

    /* Introduce new variable */
    prog->variables[ prog->variables_cnt ] = xpl_strdup( name );

    return prog->variables_cnt++;;
}

int xpl_get_literal( xpl_program* prog, xpl_value* val )
{
    /* Else, eventually allocate memory for new variables */
    if( ( prog->literals_cnt % XPL_MALLOCSTEP ) == 0 )
    {
        prog->literals = (xpl_value**)xpl_malloc(
                                (char*)prog->literals,
                                    ( prog->literals_cnt + XPL_MALLOCSTEP )
                                        * sizeof( xpl_value* ) );
    }

    prog->literals[ prog->literals_cnt ] = val;
    return prog->literals_cnt++;
}

int xpl_emit( xpl_program* prog, xpl_op op, int param )
{
    if( ( prog->program_cnt % XPL_MALLOCSTEP ) == 0 )
    {
        prog->program = (xpl_cmd*)xpl_malloc(
                            (char*)prog->program,
                                ( prog->program_cnt + XPL_MALLOCSTEP )
                                    * sizeof( xpl_cmd ) );
    }

    prog->program[ prog->program_cnt ].op = op;
    prog->program[ prog->program_cnt ].param = param;
    return prog->program_cnt++;
}
```

```
void xpl_reset( xpl_program* prog )
{
    int     i;

    /* Variables */
    for( i = 0; i < prog->variables_cnt; i++ )
        xpl_free( prog->variables[ i ] );

    xpl_free( (char*)prog->variables );

    /* Literals */
    for( i = 0; i < prog->literals_cnt; i++ )
        xpl_value_free( prog->literals[ i ] );

    xpl_free( (char*)prog->literals );

    /* Program */
    xpl_free( (char*)prog->program );

    memset( prog, 0, sizeof( xpl_program ) );
}
```

And for debug purposes, we want to get a visible representation of the compiled program. This can be done with a program dump function that is implemented in `xpl.debug.c`.

```
#include "xpl.h"

static char opcodes[][3+1] =
{
    "NOP", "CAL", "LIT", "LOD", "STO", "DUP", "DRP", "JMP", "JPC",
    "EQU", "NEQ", "LOT", "LEQ", "GRT", "GEQ",
    "ADD", "SUB", "MUL", "DIV"
};

extern xpl_fn   xpl_buildin_functions[];

void xpl_dump( xpl_program* prog, xpl_runtime* rt )
{
    int         i;
    xpl_cmd*    cmd;

    for( i = 0; i < prog->program_cnt; i++ )
    {
        cmd = &( prog->program[i] );

        fprintf( stderr, "%s%03d: %s ",
                    ( rt && rt->ip == cmd ) ? ">" : " ",
                        i, opcodes[ cmd->op ] );

        if( cmd->op == XPL_JMP || cmd->op == XPL_JPC )
            fprintf( stderr, "%03d", cmd->param );
        else if( cmd->op == XPL_LIT )
            fprintf( stderr, "%d (%s%s%s)", cmd->param,
                        prog->literals[ cmd->param ]->type == XPL_STRINGVAL ?
                            "\"" : "",
                        xpl_value_get_string( prog->literals[ cmd->param ] ),
                        prog->literals[ cmd->param ]->type == XPL_STRINGVAL ?
                            "\"" : "" );
        else if( cmd->op == XPL_LOD || cmd->op == XPL_STO )
            fprintf( stderr, "%d => %s", cmd->param,
                        prog->variables[ cmd->param ] );
        else if( cmd->op == XPL_CAL )
            fprintf( stderr, "%d => %s()", cmd->param,
                        xpl_buildin_functions[ cmd->param ].name );

        fprintf( stderr, "\n" );
    }
}
```

2.6.3.4. xpl_fn: Integrating some useful functions

The build-in functions in XPL are also described in one structure, `xpl_fn`. This structure holds the name of the function as it can be called from an XPL program, the minimum and maximum count of parameters, and finally a C-function callback-pointer, requiring for functions that follow the prototype declaration

```
xpl_value* function( int argc, xpl_value** argv );
```

All build-in functions, some service functions and an array holding the functions as a symbol table is stated in `xpl.functions.c`.

```
#include "xpl.h"

xpl_fn  xpl_buildin_functions[] =
{
    {   "exit",     -1,     1,      XPL_exit    },
    {   "print",    1,      -1,     XPL_print   },
    {   "prompt",   -1,     1,      XPL_prompt  },
    {   "integer",  1,      1,      XPL_integer },
    {   "float",    1,      1,      XPL_float   },
    {   "string",   1,      1,      XPL_string  },
};

int xpl_get_function( char* name )
{
    int     i;

    /* Try to find function */
    for( i = 0; i < sizeof( xpl_buildin_functions ) / sizeof( xpl_fn ); i++ )
        if( strcmp( xpl_buildin_functions[ i ].name, name ) == 0 )
            return i;

    return -1;
}

int xpl_check_function_parameters( int function, int parameter_count, int line )
{
    if( xpl_buildin_functions[ function ].min > -1 )
    {
        if( parameter_count < xpl_buildin_functions[ function ].min )
        {
            fprintf( stderr,
                "line %d: Too less parameters in call to %s(), %d parameters "
                "required at minimum",
                    line, xpl_buildin_functions[ function ].name,
                        xpl_buildin_functions[ function ].min );
            return 1;
        }
    }
    else if( xpl_buildin_functions[ function ].max > -1 )
    {
        if( parameter_count > xpl_buildin_functions[ function ].max )
        {
            fprintf( stderr,
                "line %d: Too many parameters in call to %s(), %d parameters "
                "allowed at maximum",
                    line, xpl_buildin_functions[ function ].name,
                        xpl_buildin_functions[ function ].max );
            return 1;
        }
    }

    return 0;
}

/* Build-in functions follow */

xpl_value* XPL_print( int argc, xpl_value** argv )
{
    int     i;
    for( i = 0; i < argc; i++ )
        printf( "%s\n", xpl_value_get_string( argv[ i ] ) );

    return (xpl_value*)NULL;
}

xpl_value* XPL_prompt( int argc, xpl_value** argv )
{
    char    buf [ 256 + 1 ];

    if( argc > 0 )
        printf( "%s: ", xpl_value_get_string( argv[ 0 ] ) );

    if( fgets( buf, sizeof( buf ), stdin ) )
    {
        buf[ strlen( buf ) - 1 ] = '\0';
        return xpl_value_create_string( buf, 1 );
    }

    return xpl_value_create_string( "", 1 );
}

xpl_value* XPL_exit( int argc, xpl_value** argv )
{
    int     rc      = 0;

    if( argc > 0 )
        rc = xpl_value_get_integer( argv[ 0 ] );

    exit( rc );
    return (xpl_value*)NULL;
}

xpl_value* XPL_integer( int argc, xpl_value** argv )
{
    return xpl_value_create_integer( xpl_value_get_integer( *argv ) );
}

xpl_value* XPL_float( int argc, xpl_value** argv )
{
    return xpl_value_create_float( xpl_value_get_float( *argv ) );
}
```

```
xpl_value* XPL_string( int argc, xpl_value** argv )
{
    return xpl_value_create_string( xpl_value_get_string( *argv ), 1 );
}
```

2.6.3.5. xpl_runtime - the runtime data structure

This implementation of an XPL compiler and interpreter distinguishes between static data describing the compiled XPL program, and dynamic data that is only required at program runtime. The compiler creates and modifies the `xpl_program`-object, the interpreter only reads the `xpl_program`-object and works on a `xpl_runtime`-object containing the variable data and stack.

```
/* Runtime structure */
typedef struct
{
    xpl_value** variables;                  /* Array of objects representing
                                                the variable values. The
                                                number of objects is in
                                                the corresponding xpl_program
                                                structure in the member
                                                variables_cnt
                                            */

    xpl_value** stack;                      /* Array representing the value
                                                stack */
    int         stack_cnt;                  /* Defines the top-of-stack, and
                                                the numbers of elements
                                                resisting.
                                            */

    xpl_cmd*    ip;                         /* Instruction Pointer to the
                                                currently executed code
                                                address.
                                            */
} xpl_runtime;
```

2.6.3.6. Implementing the virtual machine

Finally, the virtual machine is implemented into `xpl.run.c` as function `xpl_run()`.

```
#include "xpl.h"

extern xpl_fn   xpl_buildin_functions[];

static int xpl_push( xpl_runtime* rt, xpl_value* val )
{
    if( ( rt->stack_cnt % XPL_MALLOCSTEP ) == 0 )
    {
        rt->stack = (xpl_value**)xpl_malloc(
                            (char*)rt->stack,
                                ( rt->stack_cnt + XPL_MALLOCSTEP )
                                    * sizeof( xpl_value ) );
    }

    rt->stack[ rt->stack_cnt ] = val;
    return rt->stack_cnt++;
}

static xpl_value* xpl_pop( xpl_runtime* rt )
```

```
{
    if( !rt->stack_cnt )
        return (xpl_value*)NULL;

    return rt->stack[ --rt->stack_cnt ];
}

static void xpl_stack( xpl_runtime* rt )
{
    int     i;

    for( i = 0; i < rt->stack_cnt; i++ )
        fprintf( stderr, "% 3d: %s\n", i,
                    xpl_value_get_string( rt->stack[ i ] ) );

    if( !i )
        fprintf( stderr, "--- Stack is empty ---\n" );
}

void xpl_run( xpl_program* prog )
{
    xpl_runtime rt;
    xpl_value*  val;
    int         i;

    /* Initialize runtime */
    memset( &rt, 0, sizeof( xpl_runtime ) );
    rt.variables = (xpl_value**)xpl_malloc( (char*)NULL,
                        prog->variables_cnt * sizeof( xpl_value ) );

    rt.ip = prog->program;

    /* Program execution loop */
    while( rt.ip < prog->program + prog->program_cnt )
    {
        /*
        fprintf( stderr, "IP: %p\n", rt.ip );
        xpl_dump( prog, &rt );
        xpl_stack( &rt );
        */
        switch( rt.ip->op )
        {
            case XPL_NOP:
                /* No nothing */
                break;

            case XPL_CAL:
                /* Calling build-in functions */
                {
                    int     param_cnt;

                    /* Last stack item contains the number of parameters */
                    val = xpl_pop( &rt );
                    param_cnt = xpl_value_get_integer( val );
                    xpl_value_free( val );

                    /* Call the function */
                    val = (*(xpl_buildin_functions[ rt.ip->param ].fn))
                            ( param_cnt, rt.stack + rt.stack_cnt - param_cnt );

                    /* If no value is returned, create a default value */
                    if( !val )
                        val = xpl_value_create_integer( 0 );

                    /* Discard the parameters from stack */
                    while( param_cnt > 0 )
                    {
                        xpl_value_free( xpl_pop( &rt ) );
                        param_cnt--;
                    }

                    /* Push the return value */
                    xpl_push( &rt, val );
                }
                break;

            case XPL_LIT:
                /* Load literal and push duplicate */
                xpl_push( &rt, xpl_value_dup(
                    prog->literals[ rt.ip->param ] ) );
                break;

            case XPL_LOD:
                /* Load value from variable and push duplicate */
                xpl_push( &rt, xpl_value_dup(
                    rt.variables[ rt.ip->param ] ) );
                break;

            case XPL_STO:
                /* Store value to variable */
                if( rt.variables[ rt.ip->param ] )
                    xpl_value_free( rt.variables[ rt.ip->param ] );

                rt.variables[ rt.ip->param ] = xpl_pop( &rt );
                break;

            case XPL_DUP:
                /* Duplicate stack item */
                val = xpl_pop( &rt );
                xpl_push( &rt, val );
                xpl_push( &rt, xpl_value_dup( val ) );
                break;

            case XPL_DRP:
                /* Drop stack item */
                xpl_value_free( xpl_pop( &rt ) );
                break;

            case XPL_JMP:
                /* Jump to address */
                rt.ip = prog->program + rt.ip->param;
                continue;

            case XPL_JPC:
                /* Jump to address only if stacked value is nonzero */
                if( !xpl_value_get_integer( ( val = xpl_pop( &rt ) ) ) )
                {
                    xpl_value_free( val );
                    rt.ip = prog->program + rt.ip->param;
                    continue;
                }

                xpl_value_free( val );
                break;
```

```
default:
    {
        xpl_datatype    prefer;
        xpl_value*      op  [ 2 ];

        /* Pop operands off the stack */
        op[1] = xpl_pop( &rt );
        op[0] = xpl_pop( &rt );

        /*
         * Get best matching type for operation from both operands
         */
        if( op[0]->type == XPL_STRINGVAL ||
                op[1]->type == XPL_STRINGVAL )
            prefer = XPL_STRINGVAL;
        else if( op[0]->type == XPL_FLOATVAL ||
                    op[1]->type == XPL_FLOATVAL )
            prefer = XPL_FLOATVAL;
        else
            prefer = XPL_INTEGERVAL;

        switch( rt.ip->op )
        {
            case XPL_ADD:
                /* Addition, or with Strings, concatenation */
                if( prefer == XPL_STRINGVAL )
                {
                    char*   str;

                    str = xpl_malloc( (char*)NULL,
                        ( strlen( xpl_value_get_string( op[0] ) )
                        + strlen( xpl_value_get_string( op[1] ) )
                        + 1 ) * sizeof( char ) );


                    sprintf( str, "%s%s",
                        xpl_value_get_string( op[0] ),
                        xpl_value_get_string( op[1] ) );

                    val = xpl_value_create_string( str, 0 );
                }
                else if( prefer == XPL_FLOATVAL )
                {
                    val = xpl_value_create_float(
                            xpl_value_get_float( op[0] ) +
                                xpl_value_get_float( op[1] ) );
                }
                else
                {
                    val = xpl_value_create_integer(
                            xpl_value_get_integer( op[0] ) +
                                xpl_value_get_integer( op[1] ) );
                }
                break;

            case XPL_SUB:
                /* Substraction */
                if( prefer == XPL_FLOATVAL )
                {
                    val = xpl_value_create_float(
                            xpl_value_get_float( op[0] ) -
                                xpl_value_get_float( op[1] ) );
                }
                else
                {
                    val = xpl_value_create_integer(
                            xpl_value_get_integer( op[0] ) -
                                xpl_value_get_integer( op[1] ) );
                }
                break;

            case XPL_MUL:
                /* Multiplication */
                if( prefer == XPL_FLOATVAL )
                {
                    val = xpl_value_create_float(
                            xpl_value_get_float( op[0] ) *
                                xpl_value_get_float( op[1] ) );
                }
                else
                {
                    val = xpl_value_create_integer(
                            xpl_value_get_integer( op[0] ) *
                                xpl_value_get_integer( op[1] ) );
                }
                break;

            case XPL_DIV:
                /* Division */
                if( prefer == XPL_FLOATVAL )
                {
                    val = xpl_value_create_float(
                            xpl_value_get_float( op[0] ) /
                                xpl_value_get_float( op[1] ) );
                }
                else
                {
                    val = xpl_value_create_integer(
                            xpl_value_get_integer( op[0] ) /
                                xpl_value_get_integer( op[1] ) );
                }
                break;

            default:
                {
                    float   res;

                    /*
                     * Compare by subtracting the left operand
                     * from the right operand, or with the string
                     * comparison function strcmp, resulting in:
                     *
                     * res == 0                 equal
                     * res != 0                 not equal
                     * res < 0                  lower than
                     * res <= 0                 lower-equal
                     * res > 0                  greater-than
                     * res >=0                  greater-equal
                     */
                    if( prefer == XPL_STRINGVAL )
                    {
                        res = (float)strcmp(
                                xpl_value_get_string( op[0] ),
                                        xpl_value_get_string( op[1] ) );
                                }
                                else if( prefer == XPL_FLOATVAL )
                                {
                                    res = xpl_value_get_float( op[0] )
                                            - xpl_value_get_float( op[1] );
                                }
                                else
                                {
                                    res = (float)xpl_value_get_integer( op[0] )
                                            - xpl_value_get_integer( op[1] );
                                }

                                /* Switch comparison */
                                switch( rt.ip->op )
                                {
                                    case XPL_EQU:
                                        val = xpl_value_create_integer(
                                                !res ? 1 : 0 );
                                        break;
                                    case XPL_NEQ:
                                        val = xpl_value_create_integer(
                                                res ? 1 : 0 );
                                        break;
                                    case XPL_LOT:
                                        val = xpl_value_create_integer(
                                                res < 0 ? 1 : 0 );
                                        break;
                                    case XPL_LEQ:
                                        val = xpl_value_create_integer(
                                                res <= 0 ? 1 : 0 );
                                        break;
                                    case XPL_GRT:
                                        val = xpl_value_create_integer(
                                                res > 0 ? 1 : 0 );
                                        break;
                                    case XPL_GEQ:
                                        val = xpl_value_create_integer(
                                                res >= 0 ? 1 : 0 );
                                        break;
                                }
                            }
                        }

                        /* Free the operands */
                        xpl_value_free( op[0] );
                        xpl_value_free( op[1] );

                        /* Push the operation or comparison result */
                        xpl_push( &rt, val );
                }
                break;
        }

        /* Increment instruction pointer */
        rt.ip++;
    }

    /*
     * Clear stack
     * (if code was clearly generated, this would not be required)
     */
    for( i = 0; i < rt.stack_cnt; i++ )
        xpl_value_free( rt.stack[ i ] );

    xpl_free( (char*)rt.stack );

    /* Clear variables */
    for( i = 0; i < prog->variables_cnt; i++ )
        xpl_value_free( rt.variables[ i ] );

    xpl_free( (char*)rt.variables );
}
```

2.6.4. Implementing the compiler

Now both modules, the XPL runtime environment and the parser need to mate. The parser must be extended to work as a compiler, the code must be executed when the program successfully has been compiled. To turn the parser into a compiler, semantic actions need to be inserted to appropriate positions to generate the machine code for our virtual machine along the parse process now.

The easiest part of this implementation is the nonterminal **expression**. The productions of this nonterminal only require some code generation to push values onto the stack and perform operations and comparisons. The function call, which is part of nonterminal **expression**, first pushes and evaluates all parameter values onto the stack, then pushes the number of parameters to be taken by the CAL operation to find out how many parameters shall be used for the function call, and to clear them later on. Semantic checking if the function call provides too less or too many parameters is done here by the function `xpl_check_function_parameters()` already.

The more difficult part is the nonterminal **statement**. This nonterminal requires the backpatching of addresses and code generation within its productions. The best example for this is the statement "while".

The XPL-compiler should compile a while-statement

```
while( i > 0 ) i = i - 1;
```

into a virtual machine code like this

```
 000: LOD 0 => i
 001: LIT 0 (0)
 002: GRT
 003: JPC 011
 004: LOD 0 => i
 005: LIT 1 (1)
 006: SUB
 007: DUP
 008: STO 0 => i
 009: DRP
 010: JMP 000
 011: NOP
```

The first operations from address 000 to address 002 are emitted from the comparison. The content of variable `i` and the literal `0` are pushed and compared for greater-than. The result of the comparison will then be pushed onto the stack (0 for false, 1 for true). The JPC instruction at address 003 now compares against this value on top of the stack, and if it is false, jumps to address 011. If the comparison is true, the machine will continue to run at address 004. The opcodes from address 004 to address 009 are the result of the decrementation of variable `i`. At address 010, a JMP operation is performed to begin at address 000 again, compare the content of variable `i` greater-than `0`. This is the way how an iteration is performed within a 1-address-code machine language.

Our compiler can only emit code sequentially, and looking at the production for statement "while",

```
statement -> "while" '(' expression ')' statement
```

the code for the JPC-call must be inserted already after the expression is parsed. Here, the technique of backpatching is applied in combination with another feature of UniCC, the anonymous nonterminals. Anonymous nonterminals provide a way to insert embedded grammar structures right in place into an existing production, including semantic actions. We are able to insert anonymous nonterminals to perform semantic code generation within an unfinished production.

Exemplary this is done with the following syntax

```
statement -> "while" '(' expression ')'
                  ( [* generate jpc code *] ) statement
                       [* backpatch the jpc-call *] ;
```

which is the same as we add a normal nonterminal with an empty right-hand-side and use it only once.

```
ANONYMOUS -> [* generate jpc code *] ;

statement -> "while" '(' expression ')'
                  ANONYMOUS statement
                       [* backpatch the jpc-call *] ;
```

Anonymous nonterminals make it possible to define the entire nonterminal (with all its productions) on the right-hand side of another nonterminal, avoiding the creation of a special extra nonterminal to fulfill one special task, like it is the case here.

Our anonymous nonterminal right after the closing bracket of the header of the "while"-statement generates a JPC operation with an empty address, and returns its own operation address as semantic value. Then the following nonterminal **statement** is parsed, and finally the production of the statement "while" will be reduced.

In the reduction code, we then know the final address where the JPC-operation should jump to, so we now can backpatch its parameter with the correct address. To finally generate the JMP-call at the bottom of the loop right behind the last statement code was executed, we also have to remember the last program memory address before the code for statement "while" was generated. Another anonymous nonterminal helps out, and we get the following exemplary version of the backpatching construct.

```
statement -> "while" ( [* stack up current program address *] ) '(' expression ')'
                 ( [* generate jpc code *] ) statement
                      [* generate jmp to stacked begin address,
                         backpatch the jpc-call *] ;
```

The implemented version of this theory is this one, and may look a little bit confusing when used the first time.

```
statement   ->  "while"

                (
                    [*
                        @@ = pcb->prog->program_cnt;
                    *]
                ):begin

                    '(' expression ')'

                (
                    [*
                        @@ = xpl_emit( pcb->prog, XPL_JPC, 0 );
                    *]
                ):jpc
```

```
                    statement

                [*
                    @@ = xpl_emit( pcb->prog, XPL_JMP, @begin );
                    pcb->prog->program[ @jpc ].param =
                        pcb->prog->program_cnt;
                *]
            ;
```

But anonymous nonterminals also have their disadvantages. They represent individual symbols that are only used in one position, but can be made up of a fully-fledged nonterminal definition with multiple productions and a return value. This is also the reason why the "if...else" statement is implemented a different way in the final XPL implementation. Trying to get the drafted version with two productions for "if" with "else" and "if" without "else" driven this way will fail, because the insertion of anonymous nonterminals into the existing productions removes the ambiguity of the grammar, and causes the wrong production to be reduced. This problem could also be resolved with the assistance of anonymous nonterminals, by simply adding an optional "else" part to the "if" statement, yielding in one production for the entire "if" and "if...else" construct.

Please note, that this change has its origin in the way how we want to produce our code only. In other compiler implementations this could be solved much better and differently than here, so the drafted grammar could also be used without anonymous nonterminals.

The final version of our grammar turning it into an XPL compiler is this one.

```
#!language              "C";

//Meta information
#parser                 "XPL";
#description            "eXample Programming Language";
#copyright              "In the public domain, 2011";
#version                "0.2";
#prefix                 "xpl";

#default action         [* @@ = @1; *];
#default epsilon action [* @@ = 0; *];


//Precedence and associativity
#left               "=";

#left               "=="
                    "!="
                    "<="
                    ">="
                    '>'
                    '<'
                    ;

#left               '+'
                    '-'
                    ;

#left               '*'
                    '/'
                    ;

//Regular expressions
@string<char*>      '"' !'"'* '"'               [* @@ = @>; *]
```

```
                    ;

@identifier<char*>  'A-Za-z_' 'A-Za-z0-9_'*     [* @@ = @>; *]
                    ;


//Lexemes
#lexeme             real
                    ;

real<float>         ->       real_integer '.' real_fraction

                            [*  @@ = @real_integer + @real_fraction; *]

                    |       real_integer '.'?

                            [*  @@ = @real_integer; *]

                    |       '.' real_fraction

                            [*  @@ = @real_fraction; *]
                    ;

real_integer<float> ->      real_integer '0-9':dig

                            [* @@ = 10 * @real_integer + @dig - '0'; *]

                    |       '0-9':dig

                            [*  @@ = @dig - '0'; *]
                    ;

real_fraction<float>->      real_fraction '0-9':dig

                            [* @@ = ( @real_fraction - '0' + @dig ) / 10.0; *]

                    |       '0-9':dig

                            [*  @@ = ( @dig - '0' ) / 10.0; *]
                    ;

//Whitespace grammar construct
#whitespaces        whitespace
                    ;

whitespace          ->      ' \r\n\t'+
                    |       "//" !'\n'* '\n'
                    ;

//Goal symbol
program$            ->      statement*
                    ;

statement           ->      "if" '(' expression ')'

                            (
                                [*
                                    @@ = xpl_emit( pcb->prog, XPL_JPC, 0 );
                                *]
                            ):jpc

                                statement
```

```
        (
            "else"

            (
                [*
                    @@ = pcb->prog->program_cnt;
                    xpl_emit( pcb->prog, XPL_JMP, 0 );
                *]
            ):jmp

            statement

            [* @@ = @jmp; *]

        |

            [* @@ = -1; *]
        ):jmp

        [*
            if( @jmp >= 0 )
            {
                pcb->prog->program[ @jmp ].param =
                    pcb->prog->program_cnt;
                pcb->prog->program[ @jpc ].param =
                    @jmp + 1;
            }
            else
                pcb->prog->program[ @jpc ].param =
                    pcb->prog->program_cnt;
        *]

|       "while"

        (
            [*
                @@ = pcb->prog->program_cnt;
            *]
        ):begin

            '(' expression ')'

        (
            [*
                @@ = xpl_emit( pcb->prog, XPL_JPC, 0 );
            *]
        ):jpc

            statement

        [*
            @@ = xpl_emit( pcb->prog, XPL_JMP, @begin );
            pcb->prog->program[ @jpc ].param =
                pcb->prog->program_cnt;
        *]

|       '{' statement* '}'

|       expression ';'
        [*
            xpl_emit( pcb->prog, XPL_DRP, 0 );
```

```
                                *]

                    |           ';'
                    ;

expression          ->          variable "=" expression
                                [*
                                    xpl_emit( pcb->prog, XPL_DUP, 0 );
                                    xpl_emit( pcb->prog, XPL_STO, @variable );
                                *]

                    |           expression "==" expression
                                [*
                                    xpl_emit( pcb->prog, XPL_EQU, 0 );
                                *]

                    |           expression "!=" expression

                                [*
                                    xpl_emit( pcb->prog, XPL_NEQ, 0 );
                                *]

                    |           expression '<' expression

                                [*
                                    xpl_emit( pcb->prog, XPL_LOT, 0 );
                                *]

                    |           expression '>' expression

                                [*
                                    xpl_emit( pcb->prog, XPL_GRT, 0 );
                                *]

                    |           expression "<=" expression

                                [*
                                    xpl_emit( pcb->prog, XPL_LEQ, 0 );
                                *]

                    |           expression ">=" expression

                                [*
                                    xpl_emit( pcb->prog, XPL_GEQ, 0 );
                                *]

                    |           expression '+' expression

                                [*
                                    xpl_emit( pcb->prog, XPL_ADD, 0 );
                                *]

                    |           expression '-' expression

                                [*
                                    xpl_emit( pcb->prog, XPL_SUB, 0 );
                                *]

                    |           expression '*' expression

                                [*
                                    xpl_emit( pcb->prog, XPL_MUL, 0 );
```

```
                *]

        |       expression '/' expression

                [*
                    xpl_emit( pcb->prog, XPL_DIV, 0 );
                *]

        |       '-' expression

                [*
                    xpl_emit( pcb->prog, XPL_LIT,
                        xpl_get_literal( pcb->prog,
                            xpl_value_create_integer( -1 ) ) );
                    xpl_emit( pcb->prog, XPL_MUL, 0 );
                *]

                #precedence '*'

        |       '(' expression ')'

        |       real

                [*
                    xpl_emit( pcb->prog, XPL_LIT,
                        xpl_get_literal( pcb->prog,
                            xpl_value_create_float( @real ) ) );
                *]

        |       @string

                [*
                    /*
                        Remove the quotation
                        marks from the string
                    */
                    @string[ strlen( @string ) - 1 ] = 0;

                    /*
                        Generate code
                    */
                    xpl_emit( pcb->prog, XPL_LIT,
                        xpl_get_literal( pcb->prog,
                            xpl_value_create_string(
                                @string + 1, 1 ) ) );
                *]

        |       variable

                [*
                    xpl_emit( pcb->prog, XPL_LOD,
                        @variable );
                *]

        |       function '(' parameter_list? ')'

                [*
                    /*
                        Semantic checks if the function
                        parameters are in a valid count.
                    */
                    if( xpl_check_function_parameters(
                                        @function, @parameter_list,
                                            pcb->line ) )
                                    pcb->error_count++;

                                /* We first push the number of parameters */
                                xpl_emit( pcb->prog, XPL_LIT,
                                    xpl_get_literal( pcb->prog,
                                        xpl_value_create_integer(
                                            @parameter_list ) ) );

                                /* Then we call up the function */
                                xpl_emit( pcb->prog, XPL_CAL, @function );
                            *]
                    ;

parameter_list<int> ->      parameter_list ',' expression
                            [* @@ = @parameter_list + @expression; *]

                    |       expression

                            [* @@ = 1; *]
                    ;

variable
function <int>    ->      @identifier

                            [*
                                if( ( @@ = xpl_get_function( @identifier ) )
                                        >= 0 )
                                    @!symbol:function;
                                else
                                    @@ = xpl_get_variable(
                                            pcb->prog, @identifier );
                            *]
                    ;

/* Parser Control Block */

#pcb
[*
xpl_program*    prog;
FILE*           input;
*];

/* Prologue & Epilogue */

#prologue
[*
#include "xpl.h"

extern xpl_fn   xpl_buildin_functions[];

#define UNICC_GETINPUT  fgetc( pcb->input )
*];

#epilogue
[*


int xpl_compile( xpl_program* prog, FILE* input )
{
    @@prefix_pcb    pcb;
    memset( &pcb, 0, sizeof( @@prefix_pcb ) );

    pcb.prog = prog;
    pcb.input = input;
    pcb.eof = EOF;

    @@prefix_parse( &pcb );

    return pcb.error_count;
}

*];
```

The full, compileable and executable sourcecode of XPL can be found in appendix 1 of this manual or on the web.

3. Using UniCC

3.1. Overview

The UniCC parser generator is a software that compiles augmented parser definitions into program-modules of a higher programming language or optionally into a structured, not further targetted parser description file.

A parser definition analyzed by UniCC is an UTF-8-ASCII formatted textfile that generally contains definitions for *terminal symbols*, *nonterminal symbols* and *productions* to describe a context-free grammar. These definitions are expressed in a Backus-Naur-Form-styled notation, but with many powerful extensions and the possibility of integrated semantic code segments.

In most cases, a parser definition also contains several configuration directives, or simply called *directives*. They are used for the configuration of symbol-, task-, generation- and augmentation-related features. There are also a few directives that must be defined before any directive or grammar construction is done, because they influence general settings that cannot be changed later on. These directives are called *top-level directives*.

Additionally, a parser definition file can contain operational programming code that is revised by UniCC and inserted into appropriate positions within the resulting program module. These code fragments have the purpose to fit a particular need within the parsing process. Code blocks can be specified to various parser directives, to productions and some special terminal definitions.

Due the target-language independency of the UniCC parser generator and its parser definition language, a parser definition file can also contain additional information called *tags*. These tags can be defined globally or associated with various grammatical objects. Use of this feature is in the interest of subsequent, from UniCC detached tasks which perform operations on the output of UniCC and the use this additional information for various purposes or results.

3.2. Building from source

UniCC was entirely established and developed on top of the Phorward Toolkit.

The Phorward Toolkit and its library *libphorward* provide many useful functions for general purpose and extended software-development tasks, including standard data structures, a system-independent interface, extending data types and regular expression management functions, required by UniCC to construct the lexical analyzers.

The Phorward Toolkit is released under the BSD License. More information can be obtained from the official product website at https://phorward.info, or on GitHub (https://github.com/phorward/phorward).

Before UniCC can be built, ensure that the Phorward Toolkit is installed in its latest version.

Getting the latest version is simple. Using git, type

```
git clone https://github.com/phorward/phorward.git
```

then, change into the cloned directory and run

```
./configure
make
make install
```

After that, clone the following repositories. They provide the UniCC Parser Generator and XPL, the demonstration of a C-style toy programming language done with UniCC.

```
git clone https://github.com/phorward/unicc.git
git clone https://github.com/phorward/xpl.git
```

Change into the directory `unicc` and, again run

```
./configure
make
make install
```

After UniCC was successfully build and installed, `xpl` can be compiled out of the box without any configuration script.

If the UniCC bootstrapping toolchain is wanted, configure UniCC with

```
./configure --with-bootstrap
```

this will bootstrap the UniCC grammar parser with multiple generation states. Please do also check the README.md files of both libphorward and UniCC to get more information and build and setup.

3.3. Command-line options

UniCC primarily provides a command-line interface to invoke the parser generation process. The general calling convention of the UniCC parser generator is

```
unicc OPTIONS... filename.par
```

This command-line interface supports various, combinable options to invoke, modify and specialize the parser generation process, or to trigger further tasks.

Option	**Long option name**	**Description**
-a	--all-warnings	Runs UniCC to print all warnings that come up with the grammar. UniCC normally suppresses some warning messages that raise up during the parse table constructions according to their importance.
-b *name*	--basename *name*	Defines the specified basename *name* to be used for the output file(s) instead of the one derived by the `#prefix` directive or by the name of the input filename. This basename is used for all output files if the provided parser template causes the construction of multiple files.
-G	--grammar	Dumps an overview of the finally constructed grammar to stderr, right before the parse-tables are generated.
-h	--help	Prints a short overview about the command-line options and exits.
-l *target*	--language *target*	Sets the target language via command-line. A "#!language" directive in the grammar definition will override this value.
-V	--version	Prints copyright and version information and exists.
-n	--no-opt	Disables state optimization; By default, the resulting LALR(1) parse states are optimized during table construction by introducing a special SHIFT_REDUCE action which combines a shift and reduction, which is possible when the last symbol of a production is shifted. Standard LALR(1) parsers only support SHIFT or REDUCE, not both operations at the same time. When this option is used, UniCC produces about 20-30% more LALR(1) states.
-P	--production	Dumps an overview about the finally produced productions and their semantic actions.
-s	--stats	Prints a statistics message to stderr when parser generation has entirely been finished.
-S	--states	Dumps the generated LALR(1) states that had been generated during the parse table generation process.
-t	--stdout	Prints all generated output to stdout instead of files.
-T	--symbols	Dumps an overview of all used symbols.
-v	--verbose	Prints process messages about the specific tasks during parser generation process. Also turns on `-s`.
-w	--warnings	Print relevant warnings.
-x	--xml	Triggers UniCC to run the parser description file generator additionally to the program module generator. The parser description file generator outputs an XML-based parser representation of the generated parse tables, which can be used by third-party code generators or grammar analysis and debugging tools.
-X	--XML	Triggers UniCC to only run the parser description file generator without running the program-module generator.

Table 2: The UniCC command-line interface options.

Errors and warnings are printed to STDERR, any other kind of output to STDOUT.

3.4. Code generators

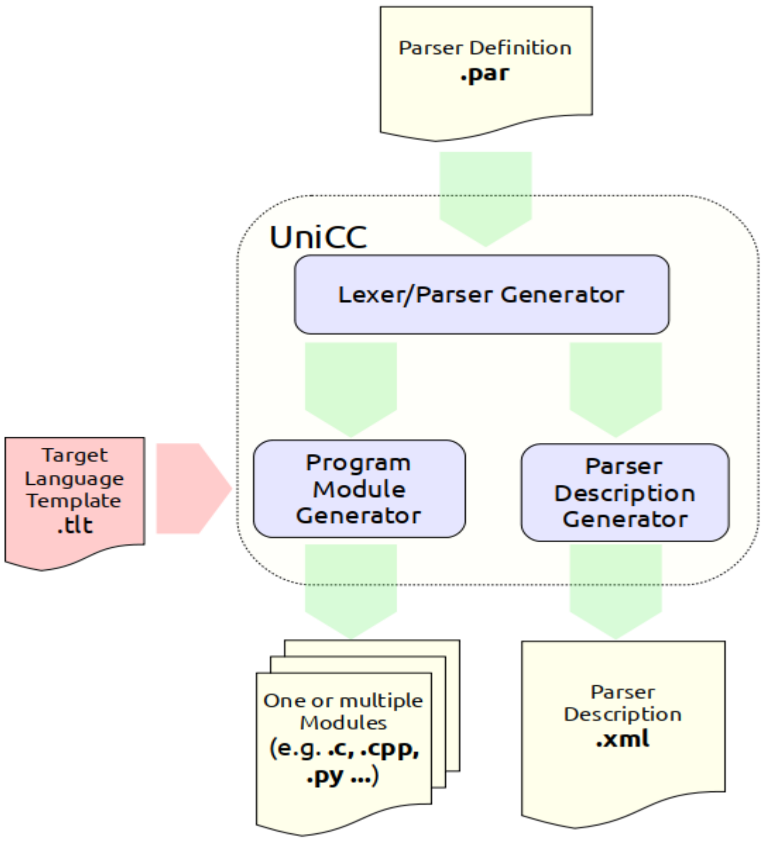


Fig. 7: An overview about the UniCC code-generators.

UniCC can not only be seen as a parser generator to compile a parser definition into a piece of program code to implement this parser. It is also a parser generator that can be used as the base for different ("any") kinds of parser analyzation, code generation and optimization issues. This is the reason why UniCC comes with two integrated code-generators: One code generator that builds program-modules expressed in a particular programming language, and one code generator to build an independent parser description file that describes the compiled form of the grammar and a transparent representation of the output parser.

The output of the first code-generator can directly be fed into a compiler or interpreter, whereas the output of latter code-generator can be analyzed by any type of other program with a specialized purpose.

3.4.1. The program-module generator

The *program-module generator* is the default code generator that is used by UniCC if nothing else is explicitly specified. It is used to build a parser-module in a specific high-level programming language. Thanks to its template-based approach, this code-generator is not targetted to one specific programming language. All target-language-related code is read from tags defined in a parser template file, which must follow a static structure that is pretended by the program-module generator in order to construct the output code.

The standard C parser template provided with UniCC is a parser template of such kind. It gives UniCC the ability to build parsers written in the C programming language which can be compiled after generation without any further modification. If more parser templates will exist somewhere in the future, UniCC will also be capable to generate program-modules for parsers written in other programming languages, like C++, C#, Java, Pascal, Fortran or anything else.

Given the very simple grammar

```
#!mode insensitive;
#parser "Simple";

start$ -> "Hello" "World"+ ;
```

UniCC constructs a C program that consists of more than 1000 lines of code using the program-module generator in combination with the standard C parser template. This output source can directly be passed to a standard C/C++ compiler like *gcc* without any further modification.

3.4.2. The XML-based parser description generator

The *parser description generator* outputs an XML-based representation of the generated parser. This parser description file contains the LALR(1) parse tables, tables for the lexical analyzer, conditioned semantic code, a structured listing of all symbols and productions with all of its tagged information, the original parser definition source and any warning or error messages produced by UniCC during the parser construction process.

Third-party programs can work on this information to generate individual parser code or code-parts, directly interpret, analyze, modify, represent or rewrite the compiled parser for any desired purpose.

To trigger the parser description generator, UniCC must be run with the *-x* or *-X* command-line option. *-x* runs both the program-module generator and the parser description generator, *-X* will only run the parser description generator.

With the very simple grammar from above,

```
#!mode insensitive;
#parser "Simple";

start$ -> "Hello" "World"+ ;
```

UniCC causes to build a grammar definition file like this when using the parser description file generator:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE parser SYSTEM "/usr/share/unicc/templates/unicc.dtd">
<parser unicc-version="1.2.0" mode="insensitive" name="Simple"
```

```
source="simple.par" basename="simple" target-language="C" char-min="0"
char-max="65534">
    <symbols>
        <symbol type="terminal" id="0" name="Hello" terminal-type="string"
defined-at="4" />
        <symbol type="terminal" id="1" name="World" terminal-type="string"
defined-at="4" />
        <symbol type="terminal" id="2" name="&amp;eof" terminal-type="system" />
        <symbol type="non-terminal" id="3" name="start" defined-at="4" />
        <symbol type="non-terminal" id="4" name="World+" derived-from="1"
defined-at="4" />
        <symbol type="non-terminal" id="5" name="start'" />
    </symbols>
    <productions>
        <production id="0" length="2" defined-at="4">
            <left-hand-side symbol-id="3" offset="0" />
            <right-hand-side symbol-id="0" offset="0" named="Hello" />
            <right-hand-side symbol-id="4" offset="1" named="World" />
        </production>
        <production id="1" length="2">
            <left-hand-side symbol-id="4" offset="0" />
            <right-hand-side symbol-id="4" offset="0" named="World" />
            <right-hand-side symbol-id="1" offset="1" named="World" />
        </production>
        <production id="2" length="1">
            <left-hand-side symbol-id="4" offset="0" />
            <right-hand-side symbol-id="1" offset="0" named="World" />
        </production>
        <production id="3" length="2">
            <left-hand-side symbol-id="5" offset="0" />
            <right-hand-side symbol-id="3" offset="0" named="start" />
            <right-hand-side symbol-id="2" offset="1" />
        </production>
    </productions>
    <states>
        <state id="0">
            <shift symbol-id="0" to-state="2" />
            <goto symbol-id="3" to-state="1" />
        </state>
        <state id="1" derived-from-state="0">
            <shift-reduce symbol-id="2" by-production="3" />
        </state>
        <state id="2" derived-from-state="0">
            <shift-reduce symbol-id="1" by-production="2" />
            <goto symbol-id="4" to-state="3" />
        </state>
        <state id="3" default-production="0" derived-from-state="2">
            <shift-reduce symbol-id="1" by-production="1" />
        </state>
    </states>
    <lexers>
        <lexer>
            <state id="0">
                <transition goto="3">
                    <character-class count="1">
                        <range from="72" to="72" />
                    </character-class>
                </transition>
                <transition goto="4">
                    <character-class count="1">
                        <range from="87" to="87" />
                    </character-class>
                </transition>
            </state>
            <state id="1" accept="1" />
            <state id="2" accept="2" />
            <state id="3">
                <transition goto="5">
                    <character-class count="1">
                        <range from="101" to="101" />
                    </character-class>
                </transition>
            </state>
            <state id="4">
                <transition goto="6">
                    <character-class count="1">
                        <range from="111" to="111" />
                    </character-class>
                </transition>
            </state>
            <state id="5">
                <transition goto="7">
                    <character-class count="1">
                        <range from="108" to="108" />
                    </character-class>
                </transition>
            </state>
            <state id="6">
                <transition goto="10">
                    <character-class count="1">
                        <range from="114" to="114" />
                    </character-class>
                </transition>
            </state>
            <state id="7">
                <transition goto="9">
                    <character-class count="1">
                        <range from="108" to="108" />
                    </character-class>
                </transition>
            </state>
            <state id="8">
                <transition goto="2">
                    <character-class count="1">
                        <range from="100" to="100" />
                    </character-class>
                </transition>
            </state>
            <state id="9">
                <transition goto="1">
                    <character-class count="1">
                        <range from="111" to="111" />
                    </character-class>
                </transition>
            </state>
            <state id="10">
                <transition goto="8">
                    <character-class count="1">
                        <range from="108" to="108" />
                    </character-class>
                </transition>
            </state>
        </lexer>
    </lexers>
```

```
    <value-types />
    <prologue />
    <epilogue />
    <pcb />
    <source>#!mode insensitive;
#parser "Simple";

start$ -&gt; "Hello" "World"+ ;
</source>
</parser>
```

All semantic code parts and their macros and variables are split into several tags which can be easily adapted, modified or enhanced. For example, the following production definition

```
variable<int*>  -> 'a-z':name   [* @@ = &variables[ @name - 'a' ]; *]
```

is compiled into the following XML structure

```
<production id="20" length="1" defined-at="55">
        <left-hand-side symbol-id="21" offset="0" />
        <right-hand-side symbol-id="37" offset="0" named="name" />
        <code defined-at="55">
                <raw> </raw>
                <variable target="left-hand-side" value-type="int*" value-type-id="0" />
                <raw> = &amp;variables[ </raw>
                <variable target="right-hand-side" offset="0" />
                <raw> - 'a' ]; </raw>
        </code>
</production>
```

which can be easily converted into another representation.

The entire document type definition (DTD) of the UniCC Parser Description Files is printed in the appendix 2 of this manual.

3.5. The parser construction modes

UniCC is a flexible parser generator that can handle two different methods to construct its parsers and their lexical analyzators.

The first and defaultly used method is called the *sensitive* parser construction mode. This construction mode is a speciality of UniCC, and gives a maximum of flexibility to implement parsers for nearly any type of context-free language. UniCC analyzes and rewrites the grammar according to several rules influencing whitespace and lexeme detection and separation. The lexical analysis, including the handling of whitespace, is broken down to single input characters enabling full context-free grammars on lexem level. A lexical analysis is still done silently in the background, but with the option that there is no direct cut between lexer and parser required.

The second method, called *insensitive* parser construction mode, always uses one single lexical analyzer that identifies terminal symbols. The difference to the sensitive mode is, that lesser states are produced, because the grammar is not rewritten, and whitespace is directly absorbed within the stage of lexical analysis. Overlapping character-classes can not be used in this mode. This construction mode can be compared to most other parser generators like the one used by the combination of *lex* and *yacc*.

It depends on the requirements of the grammar which construction mode should be used. The specialities on the two construction methods are described below. The construction mode can be changed with the `#!mode` top-level directive.

3.5.1. Sensitive mode

The whitespace sensitive parser construction mode gives a maximum of flexibility on whitespace and lexeme construction and their behavior, and is the default if no other construction mode is wanted. It is a UniCC-speciality that was never provided by any other parser generator before in this way.

The most common characteristic of this construction mode is, that UniCC entirely rewrites the grammar according to whitespace and lexeme definitions, to make whitespace only valid in selected situations. The definition of whitespace is not limited to one terminal symbol anymore in this mode. The symbol defining whitespace can be a nonterminal described as part of the context-free grammar itself without any limitations. An optional call to this nonterminal is added behind every terminal symbol and in front of the goal symbol during the grammar revision process. To disallow whitespace in particular constructs, some nonterminals can be defined as lexemes, and will be covered like terminal symbols during grammar revision in order to the whitespace-matching, but are also made up as part of the context-free grammar.

The advantage of this construction mode is, that whitespace and lexemes can be expressed in a much more powerful context-free grammar rather than as regular patterns matched by the lexical analyzer. The lexical analysis apparently becomes part of the parser with all its possibilities, but the grammar is expressed as whitespace would be handled apart from it. Additionally, the true lexical analyzer can be optionally used to parse the atomic terminals, as part of lexemes or on its own.

To make this parsing approach possible, overlapping character terminals are made unique and split up into nonterminals. A lexical analyzer is constructed individually for every LALR state, to only match symbols that are valid in the sensitive context of the given state. Some grammars that use this mode may cause a high number of states and many different lexical analyzers. But it enables the ultimative maximum of flexibility ever provided by any LALR(1) parser.

As an example, then following grammar is given.

```
#!mode      sensitive;

#left       '+' '-';
#left       '*' '/';

#whitespaces whitespace;

whitespace  -> ' \t\n'
            ;

#lexeme     integer;
integer     ->  '0-9'+
            ;

start$      -> expr*
            ;

expr        ->  expr '+' expr
            |  expr '-' expr
            |  expr '*' expr
            |  expr '/' expr
            |  '(' expr ')'
            |  integer
            ;
```

whitespace is configured as whitespace nonterminal here, *integer* is a lexem that is triggered as a coherent lexical unit, where not whitespace is allowed within. Using this grammar to parse the input string *18 * 2*, the following syntax tree will be constructed. The nonterminals *&whitespace**, *&whitespace+* and *&whitespace* are automatically inserted by UniCC. Also the symbol *\t-\n #*, likewise a generated nonternimal symbol, has been inserted by UniCC during character-class separation. Grammars modified by UniCC are heavier to read, but they gain this unique features in flexibility.

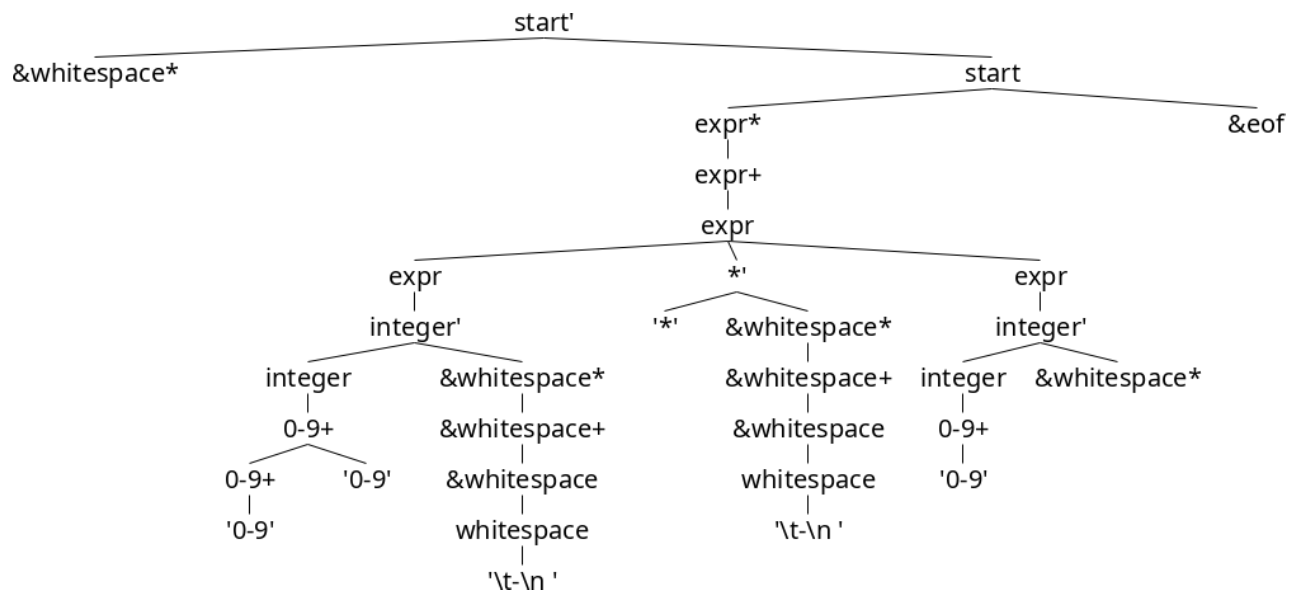


Fig. 8: Syntax tree of the string "18 * 2" using a sensitively constructed parser.

The underlying, rewritten grammar that is generated by UniCC can be dumped by using the *-G* command-line option.

3.5.2. Insensitive mode

The whitespace insensitive parser construction mode can be compared to most existing parser generators. It strictly separates the part of the lexical analysis from the parser. Whitespace is only handled and consumed by the lexical analyzer. Every terminal symbol is concerned as a lexical unit standing on its own. The grammar is not rewritten nor modified by UniCC, and results in a faster parser with lesser states.

The disadvantage of this parser construction mode is, that the whitespace-sensitive aspect, enabling the full control of any whitespace situation and speciality gets lost. Anyway, parser constructed in insensitive mode can be used for most parsing issues.

Trying to compile above grammar using `#!mode insensitive;` will throw some errors because this grammar uses features that can only be handled in insensitive mode.

```
#!mode      insensitive;

#left       '+' '-';
#left       '*' '/';

#whitespaces ' \t\n';

@integer    '0-9'+
            ;

start$      -> expr*
            ;

expr        ->  expr '+' expr
            |  expr '-' expr
            |  expr '*' expr
            |  expr '/' expr
            |  '(' expr ')'
            |  @integer
            ;
```

In insensitive mode, it is only allowed to use one terminal symbol as whitespace. This terminal symbol can be a character-class, a string or a regular expression definition. Nonterminal symbols can't be configured to be used as whitespace. The use of the lexeme-directive becomes effectless, so the former lexem nonterminal */integer* must be rewritten to a regular-expression terminal *@integer* here.

Parsing the same expression *18 * 2* yields in the following, much smaller syntax tree. The whitespace handling is not covered by the parser anymore, and is run "silently" in the background within the lexical analyzer.

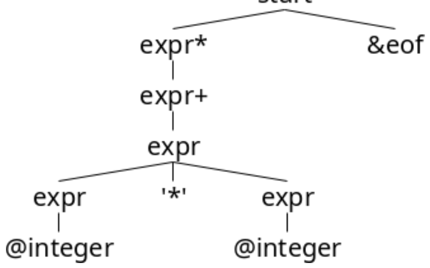


Fig. 9: Syntax tree of the string "18 * 2" using an insensitively constructed parser.

4. UniCC Grammar Definitions

4.1. Comments

Commenting grammars is possible everywhere in the grammar code. UniCC supports the C++-styled standard comment forms, which are `/*...*/` for block comments and `//` for single line comments, to ignore all the rest of a line.

```
//Define goal symbol
example$ -> "PRINT" expr
         |  "IF" expr "THEN" example ';' //Now we can do IFs! :)
         // | "START" expr
         |  "READ" var
         /*
            We want to implement this later:
         |  "GOTO" line
         |  "GOSUB" line
         */
         ;
```

It must be annotated, that this commenting style is only possible within the grammar code itself. Comments in semantic code blocks must be expressed in the commenting style of the respective target programming language that is used. These comments will also be copied into the resulting parser program-module.

4.2. Escape sequences

UniCC supports ANSI-C-styled escape-sequences within any string or value that is used in the grammar or any directive. The following table provides a listing of all available escape sequences.

Escape sequence	Description
`\a`	Bell (alert)
`\b`	Backspace
`\f`	Formfeed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\'`	Single quotation mark
`\"`	Double quotation mark
`\\`	Backslash
`\OOO`	ASCII character in octal notation, (O = octal digit)
`\xHH`	ASCII character in hexadecimal notation (H = hexadecimal digit)
`\uHHHH`	32-Bit Unicode character in hexadecimal notation (H = hexadecimal digit)
`\UHHHHHHHH`	64-Bit Unicode character in hexadecimal notation (H = hexadecimal digit)

Table 3: Global escape sequences to be used in UniCC.

4.3. Definition blocks

The UniCC grammar definition language provides blocks of several definitions. Each block has an introductional symbol, e.g. an identifier for a nonterminal definition, or a particular parser configuration directive, and ends with a semicolon (`;`).

```
//Confguration directive block
#lexeme int ;

//Nonterminal definition block
int -> '0-9'+ ;

//Terminal definition block (for terminals based on a regular expression)
@id name 'A-Za-z_'+ ;
```

Any other type of definition is invalid, and will result in a parse error.

4.4. Grammars

Grammars are expressed using rules that describe in which way symbols may appear. The symbols within a grammar define grammatical units in form of *terminal symbols* (terminals) which immediately are expected in the input, and *nonterminal symbols* (nonterminals). Nonterminal symbols union one or more grammatical rules, so called *productions* into one "fictive" symbol to be inserted into other productions, or to form the goal symbol. Every symbol (both terminal and nonterminal) in UniCC is always declared by its first use. Its particular description or definition - if required - can be made elsewhere later in the grammar. Symbols without a definition will be reported by UniCC, and maybe stop the further compilation process of the grammar, with a request for correction.

4.4.1. Terminal symbols

Terminal symbols (terminals) define a piece of raw input data that is read from the input stream. They can be defined and specified in various ways. A single character or a set of valid characters can form a terminal, but also a static character sequence or character sequences that matches to a regular (type-0) language, defined via regular expressions.

All terminal symbols are identified by an lexical analyzer. Every terminal symbol yields in a leaf within the resulting parse tree, it has no children.

4.4.1.1. Characters

Character terminals, also called *character-classes*, are the simplest definition type of a terminal symbol. This kind of terminal symbol defines a character or a set of valid input characters.

A character-class is specified by strings enclosed in single quotation marks `'...'`, and is defined by its use. Once a character-class terminal is used somewhere in the grammar or any directive, it is defined and becomes known to UniCC.

To match only one individual character, for example an **a**, then `'a'` is the correct definition for such a terminal. To define all characters from `a` to `f` as valid input, `'abcdef'` and `'a-f'` as range definition is possible. To define a terminal matching the characters from `a` to `z`, `0` to `9` and the `=` equal sign, `'a-z0-9='` can be specified. If the range is negative, for example `'z-a`, its still interpreted as `'a-z'`. To define the dash **-** itself as part of the character class, it the best to express it as first character of the character-class string, or in a place where no dash is interpreted as range definition, for example `'-+*/'` or `'a-z-+'`.

UniCC accepts escape sequences to define special characters and unprintable characters. UTF-8 characters are directly accepted. For example `'\n-â ¬'` is a valid definition with an escape-sequence for newline and the Euro currency sign.

Character terminals can be negated using an preceding exclamation-mark. For example `!'a-z'` accepts all characters except the range from `a` to `z` (this will include all characters in the UniCC character universe, from 0x0 to 0xFFFF, except above range).

In the UniCC Standard C Parser Template, a character-terminal is always associated with the data-type `int` containing the character-code of the matched character. This value can be used in semantic actions.

```
digit<int> -> '0-9':dig      [* @@ = @dig - '0'; *]
```

It depends on the used implementation language and parser template which data-type is used for terminals expressed by character-classes.

4.4.1.2. String sequences

String terminals define a sequence of characters, which must exactly match to the string specified in the current input.

They are defined similarly to character-terminals, but are enclosed by double quotation marks `"..."`. The same escape sequences as in character-terminals can be used. `"while"` is an example for a string definition matching a keyword. `"#special operation"` defines a string with a blank; If this blank is not given in the input, the string sequence is not matched. If two blanks are given, the string sequence is even not matched. Such a string definition overrides any whitespace definition, the blank is mandatory in this situation, and handled as part of the lexical analysis, rather than the grammar.

Internally, string terminals are handled like terminals based on regular expressions, and have the strongest level of specialization within the lexical analysis. This means, that a string-terminal that exactly fits to a current input string is matched before any regular expression terminal matching the same string does.

String terminals don't have a semantic data-type association. They stand on their own and represent a static lexical unit, in most cases some kind of keyword.

With the use of the `#case-insensitive strings` directive, string terminals can be configured to ignore upper-/lower case order, if case-conversion is possible with the characters they are made up from.

4.4.1.3. Regular expressions

Terminal symbols based on regular expressions are the most flexible style of expressing a terminal symbol. They can be made-up of a fully-fledged type-0 regular language. Regular expressions must explicitly be defined in their own definition block within the grammar, using the following syntax.

```
@identifier regular-expression-term [*semantic actions for the lexical analysis*];
```

The `@` character introduces the regular expression, both in its definition and wherever the symbol is used. The regular-expression itself is defined by sequences of the following, already known syntactical elements.

Construct	Usage
`'...'` or `!'...'`	Specifies a character, character-class or negated character-class.
`"..."`	Specifies a static string.
`.`	Specifies a character-class standing for "any character". Using this construct causes the terminal to be configured as "non-greedy".
`(` and `)`	Parentheses to build sub-expressions, equal to embedded productions.
`|`	The alternative operator to define multiple expressions at one expression level.
`*`	Kleene closure (none or several of previous expression) modifier.
`+`	Positive closure (one or several of previous expression) modifier.
`?`	Optional closure (none or one of previous expression) modifier.

Table 4: Lexical symbols for regular expression construction

To get more familiar with this syntax, a few examples follow.

```
//A simple identifier!
@identifier 'A-Za-z_'+ ;

//Extended identifier...
@extidentifier 'A-Za-z_' 'A-Za-z0-9_'* ;

//A fictive example with sub-expressions would be
@hello "Hello" ' \t'+ ( "World" | !'A-Za-z' )? ;

//A string value
@string '"' .* '"' ;
```

The order of the definition of terminals based on regular expression also indicates their level of specialization, so specialized regular expressions shall be defined first in the grammar. It is strongly recommended to describe all regular expressions before the first nonterminal definition is done, to avoid unexpected behaviors, which are having they origin in wrong definition orders in most cases.

Note, that terminals based on regular expression can always replace a character terminal or string terminal. Regular expression terminals are sorted behind string terminals, but before character terminals in their specialization order.

```
@abc      'abc'
          ;

example$ -> @abc
          | 'abc' //this production will never be matched!
          ;
```

It is obvious, that terminals which are defined via regular expressions are a very powerful tool. But they are also often the source for unexpected parser behaviors, that let some grammar developers become desperate in many situations. Especially in the sensitive parser construction mode, regular expressions should really be handled with care.

4.4.1.3.1. Semantic actions

Terminals defined via regular expressions can also be equipped with individual semantic code and a data-type for semantic augmentation, to return an individual semantic value to the reduction actions used in the grammar. This semantic code can be used for different purposes, also simultaneously: To extract semantic values, to modify the input, or to perform semantic symbol selections.

Same as within semantic reduction codes of productions, some macros within this semantic code block for terminals can be used to access the return value and the matched input string. Their meaning is not the same in all target languages, it depends on the target language template.

Macro	Usage
`@@`	Defines the return value that is associated with the terminal. This variable is of the same type that is specified for the symbol or the default type.
`@>`	Defines the start of the string. In the C standard template parser, this is a char-pointer to the first byte of the matched string.
`@<`	Defines the end of the string. In the C standard template parser, this is a char-pointer to the last byte of the matched string.
`@!symbol:name`	Sets the returned symbol to *name*, for semantic symbol selections.

Table 5: Semantic macros in regular expressions.

Some examples in C:

```
//Match an integer
@integer<int>    '0-9'+

    [* @@ = atoi( @> ); *]
    ;

//Match a string
@string<char*>   '"' !'"'* '"'

    [*
       @@ = @>;
    *]
    ;
```

In various target languages, the semantic action code of regular expression terminals may be the source for memory leaks if they are used faulty. Depending on the used parser target language template and the parser mode, the semantic part of a regular expression terminal can be executed multiple times, to allow for semantic symbol selections, but also to extract semantic values from the input, as shown above. For such cases, every parser template sets some variables or pre-processor directives to turn-off areas within the semantic code which may cause memory problems. The documentation of the particular UniCC parser target language template should handle this topic.

4.4.1.3.2. Defining the greediness (#greedy, #non-greedy)

Terminal symbols based on regular expressions can be defined to be in a *greedy* or *non-greedy* configuration, relating to their behavior during the process of the lexical analysis.

UniCC automatically switches a regular expression terminal that uses the dot-wildcard (`.`, for matching any character) as non-greedy. The greediness can also be explicitly set or overridden using the terminal symbol configuration directives `#greedy` and `#non-greedy`.

To describe the problem, the following grammar is used.

```
#!mode insensitive;
#!language "C";

#whitespaces @WHITESPACE;

@WHITESPACE      ' \t\r\n'+
                 | "/*" .* "*/"
                 ;

@int             '0-9'+
                 [* printf( "int %s\n", @> ); *]
                 ;

start$           -> @int ;
```

There is a whitespace definition for comments reading `"/*" .* "*/"`. This definition will result in a regular expression terminal matching a comment that begins with `/*` and ends with `*/`. The `.*` defines "none or multiple characters of any kind". In this special case, UniCC switches the nonterminal `@WHITESPACE` to be non-greedy, due to the use of the dot-placeholder (`.`).

Now, using the input

```
3 /* Hello */ 4 /* World */ 5
```

one would expect, that `3`, `4` and `5` are returned by the above grammar, and the comments are ignored. A non-greedy configuration of `@WHITESPACE`, as it is automatically set by UniCC in this example would correctly do this. Nevertheless, a "greedy" configuration of `@WHITESPACE` only return `3` and `5`, because a comment that is `/* Hello */ 4 /* World */` is identified. This first `*/` is also recognized in greedy-mode terminals, but the lexical analyzer will scan further until it (maybe) gets a second `*/`, which is the case here. Sometimes, this behavior is wanted, sometimes it is not.

Finally it depends on the grammar writer how it should be handled. To enable full control about if such regular expression-based terminals are configured to be greedy or non-greedy, UniCC offers the directives `#greedy` and `#nongreedy`, that had been mentioned above.

Changing above definition of `@WHITESPACE` to be read as

```
@WHITESPACE      ' \t\r\n'+
                 | "/*" .* "*/"
                 #greedy
                 ;
```

Would scan up the way in the second, greedy case. `#greedy` and `#non-greedy` are attached right before the definition's end marker (`;`) but behind possible semantic code. Switching terminals to greedy or non-greedy may result in entirely different parse trees and results, so that the use of this tool should only be done carefully.

4.4.1.3.3. Semantic terminal determination

UniCC supports a possibility to perform semantic terminal determination in lexical semantic actions. This possibility is well implemented and tested, but has some caveats to know about when used in the sensitive parser construction mode in combination with programming languages that don't have an automatic memory management (like C). It is more secure to use the method of semantic nonterminal determination as described below, which has no disadvantages according to the lexical way. So this topic is only shortly discussed here, and it is advised to use the nonterminal way of semantic terminal determination.

In some cases, there are terminal symbols that are build-up from the same input sequence, but require some more semantic checks to definitely decide which symbol matches. There are also cases, where the lexical value of a token has a special meaning to the grammar, e.g. a function name referenced in a compiler's symbol table. For such cases, UniCC features semantic symbol determinations within the lexical analyzer. All symbols that match to a given regular expression are unioned with one expression. Then, the semantic action code can perform checking tasks and set the matched symbol to one of the associated symbols. The association is done with the macro `@!symbol:<name>`, where name defines the symbol to be returned.

Given the example, that an empty string and a nonempty string shall be matched as single symbols, this can be handled with a semantic symbol selection.

```
//Match a string or empty string
@string empty_string <char*>    '"' !'"'* '"'

    [*
        /* Select symbol! */
        if( strlen( @> ) > 2 )
            @!symbol:string;
        else
            @!symbol:empty_string;
/*
    Run below code only when shifting, and not only at token detection.
    (this is one of the caveats in the standard C parser template when using
    semantic terminal determination with dynamic memory allocation)
*/
#if UNICC_ON_SHIFT
        @@ = strdup( @> );
#endif
    *]
    ;
```

4.4.1.4. Terminal anomalies

When using the whitespace sensitive parser construction mode, terminals could be broken down to grammatical constructs parsing lexemes. This sometimes raises a problem that is known as a terminal anomaly. This terminal anomaly can occur between regular expression-based terminals (these are terminal symbols, that are based on strings or regular expressions) and grammar constructs build on character-classes that may consume the same input sequence like the regular expression-based terminal does. In such a situation, the parser may run into an ambiguity conflict that matches both the content of a lexeme that is build of the same characters as the valid regular expression-based terminal does.

The following example defines a grammar that generates this kind of conflict.

```
#lexeme             ident;
#lexeme separation on;
#whitespaces        ' ';

program$        -> empty_or_stmt* '\0'
                ;

empty_or_stmt   -> '\n'
                |  stmt
                ;

stmt            -> "IF" empty_or_stmt*
                        "ELSE" empty_or_stmt*
                            "ENDIF"
                |  ident
                ;

ident           -> 'A-Za-z_' 'A-Za-z0-9_'*
                                ;
```

If this is compiled with `unicc -w`, the following warnings are raising up.

```
unicc: warning: state 5: Terminal anomaly at shift on 'A-Z_a-z' and reduce on 'ELSE'
    (0) empty_or_stmt+ -> empty_or_stmt+ .empty_or_stmt
    (2) empty_or_stmt* -> empty_or_stmt+ .        { '\x0' "ELSE" "ENDIF" }
unicc: warning: state 5: Terminal anomaly at shift on 'A-Z_a-z' and reduce on 'ENDIF'
    (0) empty_or_stmt+ -> empty_or_stmt+ .empty_or_stmt
    (2) empty_or_stmt* -> empty_or_stmt+ .        { '\x0' "ELSE" "ENDIF" }
```

UniCC detects these conflicts by testing suspicious nonterminals against all regular expression-based terminals. These messages declare, that the input for *ELSE* and *ENDIF* is both recognized by the nonterminal `ident` and their relating string terminals, defining them as keywords. Due to UniCCs build-in terminal precedence leveling, the string terminals will always take a higher precedence than the character-class terminals, but maybe the grammar developer gets an unwanted parse result. This warning helps to detect and fix this issue.

To avoid this warning, the `#reserve terminals` directive shall be used. It disables terminal anomaly detection and always assumes that regular expression terminals take precedence over any lexemes.

Terminal anomalies do not raise up in insensitive mode parsers.

4.4.2. Nonterminal symbols

Nonterminal symbols (nonterminals) can be seen as "grammar variables". They always cause a branch to a subsequent grammatical structure within the parse tree, and represent nodes to other nonterminals, which are nodes as usual, or terminals, which are leafs of the tree.

Nonterminals contain one or more *productions*. Productions form sequences of terminal and nonterminal symbols, describing the syntax to which the defined nonterminal can be expanded to. Everytime a nonterminal appears within a production, all its specific rules are valid and possible at the particular situation in the sequence.

Both the nonterminal symbol and its productions are described as a grammar definition block, and follow a variant of the Backus-Naur-Form meta language to describe context-free languages.

```
nonterminal -> production1 | production2 | productionN... ;
```

The nonterminal's name that should be defined appears on the "left-hand side". Left-hand side means "left from the arrow sign `->`". A grammar definition block unions the definition of nonterminals, their productions and the symbol sequences within the productions, which in turn can be definitions of terminal symbols. Everything which is production related is made on the "right-hand side", so right of the arrow sign `->`.

Here are some examples to follow.

```
//We want to use one regex-terminal
@name 'A-Za-z'+ ;

//Defining a goal symbol with one production
example$ -> hello empty world ;

//'hello', 'empty' and 'world' are nonterminals declared above.

//Now their definition follows:
hello -> "Hello" ;
world -> "World" | @name ; //Two productions.
empty -> ; //A nonterminal with an empty production.
```

A nonterminal definition can also be split into several grammar definition blocks; everytime the nonterminal appears on the left-hand side, all the defined productions are associated to it.

4.4.2.1. The goal-symbol

A special case is the *goal-symbol*. This nonterminal symbol defines the root of the parse tree, and must exist in every grammar. When the goal-symbol is finally reduced, the parse tree is completed and input was successfully tested for correctness. The goal-symbol is defined by appending a dollar-sign `$` to the nonterminal to be the goal-symbol on the left-hand side, like below with

```
example$ -> hello empty world ;
```

Only one goal symbol is allowed per grammar. Multiple goal symbol definitions throw an error, and no parser will be constructed.

4.4.2.2. Semantic nonterminal determination

There may be cases, where the distinct determination of a nonterminal requires more semantic checks to define a symbol's correct meaning. This can be the case if a grammar allows for function and variable names that are made up of character sequences. There may be a construct within the grammar to recognize the identifier, e.g. a regular expression terminal `@ident`. But there is no possibility on the grammar definition level to clearly define whether a parsed identifier refers to a function name or to a variable name. For this case, UniCC supports semantic nonterminal determinations. The semantic code block triggers some kind of selection task, for example a symbol table call, to find out if the addressed identifier is the name of a variable or a function. Then, the semantic code selects the kind of nonterminal to be used in the parser. The feature of semantic nonterminal determination allows to perform a dynamic manipulation of the parser within semantic actions.

This mechanism is done by declaring a production with multiple left-hand sides. The first specified left-hand side will always be used as default. The semantic selection is specified in the reduction code using a special macro `@!symbol:<name>`, where `<name>` refers to the name of the left-hand side symbol to be set.

For the problem described above, the following grammar would be the adequate solution.

```
@ident<char*>    'A-Za-z_'+   [* @@ = @>; *]
                 ;

//Two productions are used here
value$          -> variable | function
                            ;

//Multiple nonterminal definition with semantic symbol determination
variable
function            -> @ident   [*
                                /*
                                    Return symbol 'function' if this is the
                                    name of a function in the symbol table...
                                */
                                if( is_function( @ident ) )
                                    @!symbol:function;
                            *]
                 ;
```

A similar mechanism can also be implemented within the use of lexical analyzer terminal determination, but is not advised. Refer the topic to get more information about related caveats.

4.4.3. Productions

A production defines a sequence of symbols that is valid in the context the production may appear in. Every production is associated with a nonterminal symbol, a nonterminal symbol in turn is made up of one or multiple productions (see above).

Productions can only be defined within a grammar definition block as part of a nonterminal definition, by using a Backus-Naur-Form-compliant syntax. All productions a nonterminal is constructed from are defined on the right of the arrow-sign `->`, which is the reason why a production is also called *the right-hand side* of a grammatical rule. In turn, the nonterminal it belongs to is called *the left-hand side*.

Multiple production definitions on the right-hand side are separated by pipes (`|`), or must be stated into a new grammar definition block.

```
nonterm -> hello | world ;
```

equals to

```
nonterm -> hello ;
nonterm -> world ;
```

When the parser of a defined grammar is executed, it reads terminal-symbols from the input according the sequence rules defined by the grammar's productions. Every symbol (both terminal and nonterminal symbol) is shifted, which means it is consumed and put on a stack when its matched in the input.

Then, if a sequence of symbols resisting on the stack exactly matches to one production, it will be reduced. In this case, the term *handle* is used, so there is a handle to a production on the stack. To reduce the production means, that the parser validated the input according to a productions rules as valid. It then replaces the productions sequence (the handle) by its left-hand side nonterminal, which is part of the next, subsequent production, or the goal-symbol, which defines the input to be valid and successfully terminates the parser. All this shifting and reducing is done on an internal stack within the parser, which is holding the current parser state. The parser is oriented on the parse-tables constructed by UniCC. They predict which terminal symbols are valid in the current context and which action must be performed next.

A production may also exist of no symbol sequence. These productions are called *empty productions*, or sometimes *epsilon productions*. Their appearance performs an immediate reduction of the nonterminal, if no other production matches. This is an example for a nonterminal with two productions. Latter one is an empty production.

```
nonterm -> hello | ;
```

4.4.3.1. Semantic actions

The reduction of a production means, that a new node in the parse-tree is constructed. The terminal symbols within the production's rule will become leafs, where the nonterminal symbols become nodes to subsequent, previously reduced rules. Every symbol within the parse tree can be augmented with user-defined data.

For example, a terminal symbol `@integer` may hold the integer value of the analyzed integer number. When a production is defined as `@integer '+' @integer`, and matches the current sequence handle, a reduction is caused. Right before this is done, there are three symbols on the stack: An integer number, the operator **+**, and another integer number. Because every symbol within the parse tree can be augmented with a value, the semantic value behind this sequence can be calculated right when the parse tree is constructed. This means: The programmer is able to put individual code to every production, which can pass values to the newly constructed node and to upperlying productions.

This reduction code is written as a code-block `[* ... *]` behind the sequence that defines the production. Within each code-block, UniCC provides a set of macros to access the left-hand side (the "return value" of the rule) and the right-hand side items, respective every symbol of the sequence on the reduced right-hand side.

Macro	Usage
`@@`	Defines the semantic value to be associated with the left-hand side. It can be seen as the "return value" of the production.
`@<offset>`	Access the semantic values of right-hand side symbols via they offset.
`@<name>`, `@"<name>"`, `@'<name>'`	Access right-hand side via their speaking alias names, which can be an identifier or a string with blanks and other, special characters, when put in quotation marks.
`@!symbol:<name>`	Specifiy semantic-value dependent nonterminal within a production's reduction code.

Table 6: Semantic macros to be used within reduction actions.

The reduction value (also referred as the *left-hand side value*) can be accessed with the macro `@@`.

```
boolean$ -> "true"  [* @@ = 1; *]
         |  "false" [* @@ = 0; *]
         ;
```

There are several ways to access symbols on the right-hand side. The simplest is to access them by their offset of appearance. This is done with the variables `@1`, `@2`, `@3` etc. UniCC validates all used semantic macros within reduction code blocks, so if there is an access to offset 3 in a production that has only two symbols, it will drop an error.

A simple, augmented grammar:

```
//Get integer from input
@integer    '0-9'+                [* @@ = atoi( @> ); *];

//Parse tiny expressions
example$ -> expr                  [* printf( "= %d\n", @1 ); *]
         ;

expr     -> @integer '+' @integer  [* @@ = @1 + @3; *]
         |      @integer '-' @integer  [* @@ = @1 - @4; *] //<- Error!
         ;
```

UniCC does also support the feature of providing individual reference identifiers for every symbol of the right-hand side, using the syntax `symbol:identifier`. For example `'a-z':char` or `@name:ident` would be adequate right-hand side identifiers. Its also possible to provide long-strings, for example `'0-9'+:"A special number"`.

In case a nonterminal or regular-expression-based terminal appears, an identifying name for it is automatically associated with the same name as the symbol, so no identifier must be provided manually. Note, that this automatism fits only to the first occurrence of the symbol on the particular right-hand side. If the same symbol appears multiple times on the same right-hand side, its first occurence can be accessed with this default identifier only.

Identified symbols can then by accessed by `@identifier`, `@"Identifier String"` or `@'Identifier String'` within the reduction code. The offset-access is always possible and can be mixed, as below.

```
example$ -> expr                          [* printf( "= %d\n", @expr ); *]
         | @integer:"Hello Folks!"        [* printf( "int: %d\n",
                                                    @"Hello Folks!" ); *]
         ;

expr     -> @integer:i1 '+' @integer:i2  [* @@ = @i1 + @i3; *]
         |      @integer '-' @integer:i2     [* @@ = @integer - @3; *]
         |      '-' @integer                 [* @@ = -@1; *]
         ;
```

Some target programming languages, like C, are strongly typed. If there's no special data type given, the default type specified by the UniCC standard C parser template is `int`. UniCC provides a way to assign data-types to nonterminals, and this breaks down to production level and the reduction code. If we assume a nonterminal `ident` is made up of several characters, its return type should be a `char*` that points to a constructed string buffer. Nonterminal `ident` can them simply be declared to hold data-type `char*`.

```
string<char*>    -> 'A-Za-z_'            [* @@ = string_create();
                                            @@ = string_add_char( @@, @1 );
                                         *]
                 |  string 'A-Za-z_'     [* @@ = string_add_char( @@, @1 ); *]
                 ;
```

`string` is always of type `char*` when its used, so constructions like

```
statement -> "print" string         [* printf( "%s\n", @string ); *];
```

are possible. In other target language templates, this must not be a problem; it strongly relies on the target language template.

In some cases, a default action to set nonterminal values by default is wanted, if no action is assigned to the particular production. By using the `#default action` and `#default epsilon action` directives, such a default code block can be specified for both production with a sequence and for epsilon productions.

```
#default action         [* @@ = @1; *];
#default epsilon action [* @@ = ' '; *];

example$ -> abc             [* printf( ">%c<\n", @1 ); *];
abc      -> 'a' | 'b' | 'c' | ;
```

4.4.3.2. Virtual productions

UniCC provides a virtual production feature, which is very useful to prototype a grammar quickly. Virtual productions are created when the modifiers `*`, `+` and `?` are used. These modifiers can be assigned to any symbol, which is virtually turned into a nonterminal with some productions then.

Modifier	Meaning
`*`	Kleene closure (none or several of previous expression) modifier
`+`	Positive closure (one or several of previous expression) modifier
`?`	Optional closure (none or one of previous expression) modifier

Table 7: Virtual production modifiers.

Each of these modifiers expand into one or in case of the Kleene-closure two automatically derived nonterminal symbols that provide the desired language construct when UniCC compiles the grammar.

A simple example is to parse integer symbols, like

```
integer -> '0-9'+ ;
```

To allow for a nonterminal `statement` in several times, one could write

```
statements_or_null -> statements | ;
statements -> statement statements | statement ;
```

or one could say

```
statements_or_null -> statement* ;
```

Virtual productions have one disadvantage: Their expanding content can't be augmented with semantic code. This is because their most common use is in language prototyping, or in places where no special semantic operations on the input are required. All default semantic action blocks for productions and empty productions are automatically assigned and executed.

4.4.3.3. Anonymous nonterminals

There are several situations where the grammar developer requires to create a new nonterminal that is only used once to build-up a desired grammatical construct. Other situations require the creation of a nonterminal with only one empty production just to perform semantic tasks within another production before this production is reduced. For this special kind of situation, UniCC provides the feature of *anonymous nonterminals*, which can be defined right in place where they are required. An anonymous nonterminal is stated on a right-hand side by surrounding its productions with brackets `(` and `)`. The nonterminal has no naming or alias, an automatically generated name will be assigned by UniCC. Within the brackets, there is the ordinary way of defining productions, separated by the already-known `|`-separator (pipe). A data-type preceding the brackets allows to define a data-type for the anonymous nonterminal.

```
var_type      -> <BOOLEAN>(
                     "default"
                          [* @@ = TRUE; *]
                     | //This is empty!
                          [* @@ = FALSE; *] ):default

                @ident ';'

                [*
                       printf( "var: %s %s\n",
                              @default ? "default" : "", @ident );
                *]
                ;
```

Within the semantic action code of productions defined within anonymous nonterminals, access to the semantic values of all the right-hand side items in front of the appearance of the anonymous nonterminal is possible. The values are intermixed with the semantic values defined within the anonymous production.

A real-life example is this one, to stack-up a value temporarily within a right-hand side. It defines an anonymous nonterminal with only one empty production and a reduction code.

```
procedure_def   -> "def" @ident:funcname

                    //Need to stack the current_depth value!
                    <int>(

                        /*
                            This is an anonymous nonterminal with an empty
                            production, to be used as "embedded semantic code".
                        */
                        [*  @@ = pcb->current_depth++;
                            fprintf( pcb->debugfile, "New procedure >%s< %d\n",
                                      @funcname, pcb->current_depth ); *]

                         ):depth

                    '{' statements* '}'

                    [*
                        /*
                            This is the usual production's semantic block!
                            Do some code generation here, then replace the
                            stacked value!
                        */
                        pcb->current_depth = @depth;
                    *]
                    ;
```

In some parser generators, this feature is called "embedded actions", but in UniCC, the generic term "anonymous nonterminals" is used for both embedded actions and embedded productions.

4.5. Directives

Directives are used to configure and influence various behaviors and settings of UniCC related to the parsing, revision or construction of the grammar and its parse tables. Directives do always begin with a `#`-symbol (hash), similar to C-preprocessor directives. A special kind of directive are the "top-level directives", which begin with the sequence `#!`.

Directives may appear anywhere in the grammar definition, except top-level directives, because they modify options and parameters of the fundamental behavior of UniCC or the grammar, which affect any subsequential directive or definition. They must be specified on top of the grammar, but their use is not required (so the default settings for these top-level directives take place).

Directives follow nearly the same syntactic rules than any other definition in UniCC. The synopsis is

```
#directive-name parameter1 parameter2 parameterN ;
```

Because the directive names are build-in and constant, some of them even contain blanks. The syntax of the parameter-part depends on the directive itself. The various directives are explained in detail in the following sections, besides their behavior in the different parsing modes.

4.5.1. #!mode

The `#!mode` top-level directive defines the parser construction mode UniCC should follow. Only the values `sensitive` and `insensitive` are allowed as parameters. This mode selection influences the general way how UniCC constructs or rewrites (if necessary!) the grammar, and must be specified before any other directive or grammar construct.

```
!#mode sensitive;
```

More about the parser construction modes UniCC provides is explained in a separate chapter.

4.5.2. #!language

The `#!language` top-level directive specifies the implementation language to which the parser should be compiled in which language its semantics are written in. It may only be defined once on top of a grammar before any other directive (except other top-level directives) or language construct follows.

```
#!language "C" ;
```

The value specified here defines the parser template that is selected by the UniCC program module generator. UniCC itself does not really care about its value or meaning, due it has no language-specific decisions build-in.

The selected parser template describes various parts of the parser broken down to single parse tables cells that are used to construct parsers in nearly any possible, problem-oriented programming language.

If UniCC is forced to create a parser description file, the `#language`-directive will only be attached as attribute.

Please read more about the UniCC code generation possibilities in the section about the UniCC code generators.

4.5.3. #case insensitive strings

The `#case insensitive strings` directive switches string-based terminals to be case-insensitive or not. The directive can be called multiple times. Each time it is called and switched, subsequent string-based terminal symbol definitions are threatened according to the current case-order configuration state.

The `#case insensitive strings` directive allows for one parameter, which is of type boolean and accepts `on` and `off`.

```
//All subsequent strings are case-insensitive!
#case insensitive strings on;

example$ -> "hello" | test; //This string can be written in any case order!

#case insensitive strings off;

test -> "world"; //This one must be in lower case order!
```

Default value is `off`.

4.5.4. #default action, #default epsilon action, #default value type

The `#default action` and `#default epsilon action` directives define a default semantic action code that is used for productions with no attached semantic block, including generated and virtual productions. They expect a code block as parameter.

```
#default action         [* @@ = @1; *] ;
#default epsilon action [* @@ = 0; *] ;

example$    -> @int ; //here, the default action will be used!
            | @float  [* printf( "%f\n" ); *] //Not here!
            | //here, default epsilon action is inserted!
            ;
```

`#default action` and `#default epsilon action` can only be defined once, subsequent calls will produce a warning and are ignored.

The `#default value type` directives allows to define a default value type that is used for every symbol if no individual value type is specified. The parser templates for the standard UniCC code generator, and maybe subsequent code generators working on the XML output of UniCC may provide their own default value types. This directive allows to override this default by explicitly specifying a value type triggered as default.

```
#default value type     <float> ;
```

4.5.5. #copyright, #description, #parser, #prefix, #version (all deprecated)

These are the simplest parser directives, which only have the purpose to store an informal string value.

`#parser`, `#description`, `#copyright` and `#version` should be used to name and describe the parser and its version implemented by the grammar. `#prefix` defines a prefix value that can be inserted in function- or variable-identifiers within the generated parser to allow for various parsers in one input source file, or to meet specific symbol naming conventions in generated parser modules. When one of these directives is specified multiple times, their values will be glued together to one huge string.

```
#parser       "myBASIC";
#version      "0.43c";
#description  "A simple BASIC-compiler";
#copyright    "(C) 2011 by BasicMan";

#prefix       "mybasic";
```

All these values are only hold by UniCC and can be inserted into the parser-template or will be written to the XML output file - depending on the desired output code generator.

In the program-code generator, their values will expand with the template variables @@copyright, @@description, @@parser, @@prefix and @@version.

All these directives are flagged as deprecated, because they don't have any real benefit to the user and are relict of older times.

4.5.6. #prologue, #epilogue

The `#prologue` and `#epilogue` directives are used to define any program code that is inserted before and behind the parser implementation in the yielding parser module.

```
//Some includes and variables into the prologue
#prologue
[*
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


static int variables[ 1000 ];
static int next_var = 0;

*]
;

//Defining the parser call in the epilogue
#epilogue
[*
int main()
{
        @@prefix_pcb    pcb;
        memset( &pcb, 0, sizeof( pcb ) );

        return @@prefix_parse( &pcb );
}
*]
;
```

In many UniCC parsers that had been extended to real compilers, these two directives take much (nearly the most!) content of the parser definition files, because they contain the surrounding code that is required to run the parser in its environment it was implemented for. It strongly depends on the implementation language and parser template how these directives are handled exactly. Multiple calls of `#prologue` or `#epilogue` are possible and glue all specified code blocks together.

Please read more annotations about the UniCC standard parser template for the C programming language and its features and interfaces, which are mostly switched and modified by C preprocessor directives that are put into `#prologue` and `#epilogue` directives.

4.5.7. #left, #right, #nonassoc, #precedence

The `#left`, `#right` and `#nonassoc` directives set associativity and precedence weightings to terminal symbols, and are used to resolve conflicts in ambiguous grammars. They influence the behavior of UniCC when constructing the parse-tables, whether to shift (right-associativity) or to reduce (left-associativity) on shift-reduce conflicts. `#nonassoc` defines terminal symbols not to be associative in any way - if this is tried at parser's runtime, it will throw a syntax error.

Depending on the left- or right-associativity configuration, the syntax tree grows left- or right-leaning. This can be visualized with two simple grammars and their related syntax trees on the input expression *1+2+3+4;*.

```
#!mode      insensitive;

#left       '+' ;

@integer    '0-9'+
            ;

start$      -> expr* ';'
            ;

expr        ->  expr '+' expr
            |  @integer
            ;
```

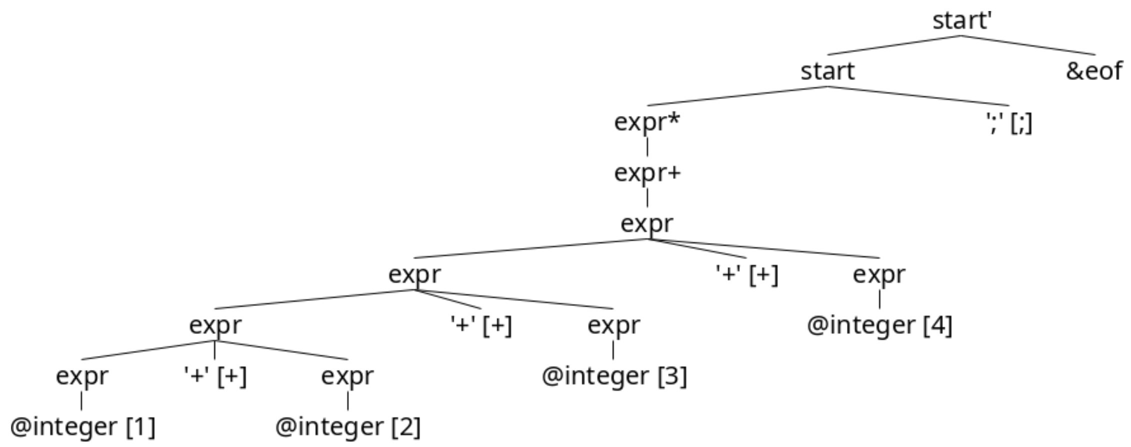


Fig. 10: Syntax tree of input "1+2+3+4;" in a left-leaning configuration.

Now the opposite using the `#right` directive.

```
#!mode      insensitive;

#right       '+' ;

@integer    '0-9'+
            ;
```

```
start$      -> expr* ';'
            ;

expr        ->  expr '+' expr
            |  @integer
            ;
```

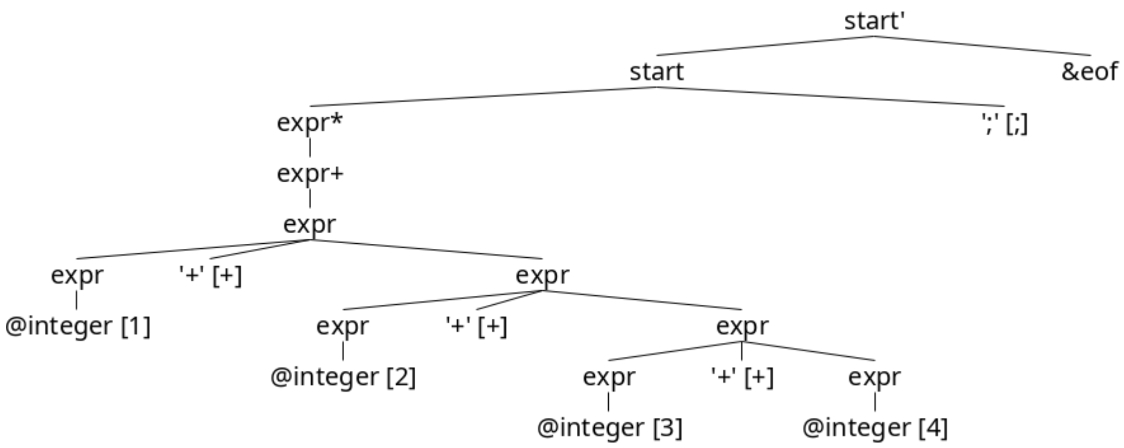


Fig. 11: Syntax tree of input "1+2+3+4;" in a right-leaning configuration.

Multiple calls of these directives cause higher precedence leveling, so it is necessary to perform multiple calls of `#left` to build-up a precedence/associativity matrix for, e.g. expressions. All symbols passed to the first `#left` directive call get a precedence level of 1, all symbols passed to the second `#left` call get a precedence level of 2, and so on.

```
#!mode      insensitive;

#nonassoc   '=';

#left       '+' '-';
#left       '*' '/';

#right      '!';
#right      '^';

@integer    '0-9'+
            ;

start$      -> expr* ';'
            ;

expr        -> expr '=' expr
            |  expr '+' expr
            |  expr '-' expr
            |  expr '*' expr
            |  expr '/' expr
            |  expr '^' expr
            |  '!' expr
            |  '(' expr ')'
            |  @integer
            ;
```

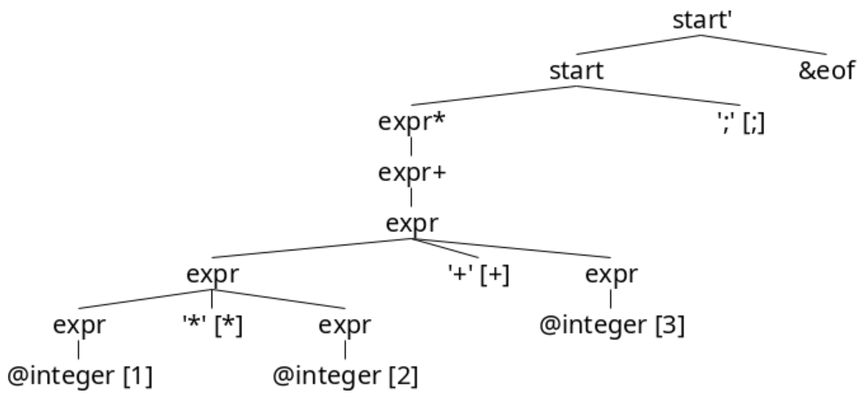


Fig. 12: Syntax tree with association precedences.

As a comparison to the above, correctly parsed syntax tree, this is the version without any associativity and precedence weightings.

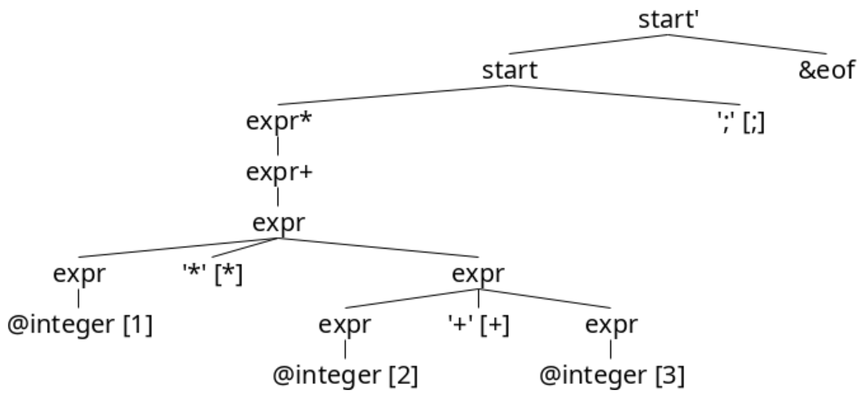


Fig. 13: Syntax tree without association precedences. UniCC compile-time warnings had been ignored.

The `#precedence`-directive is a special-case directive. It is the only kind of directive that can be used on production level. `#precedence` is used to assign a terminal's precedence to a production, to let the production take a higher level of precedence in some conflicting situations. This feature was adopted from well-known parser generators like GNU bison.

Given a grammar, that allows for both an ary and unary minus. Correct precedence associaton has already been done for `+`, `-`, `*` and `/`.

```
#left                   '+' '-' ;
#left                   '*' '/' ;

@integer        '0-9'
                ;

expression$     ->      expression '+' expression
                |       expression '-' expression
                |       expression '*' expression
                |       expression '/' expression
                |       '-' expression
                |       @integer
                ;
```

Let this parser run with the expression *-2*3*, it will produce this incorrect parse tree.

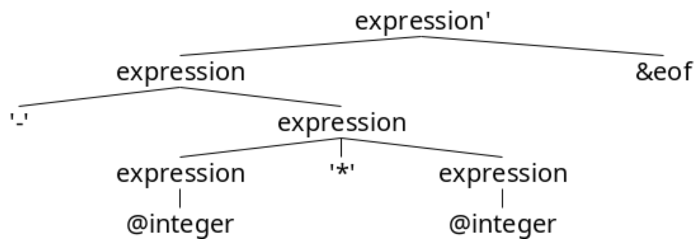


Fig. 14: The incorrect syntax tree of "-2*3".

By using the `#precedence`-directive, the production defining the unary minus can be configured to take a higher precedence than the ary minus normally takes. Defining it

```
'-' expression            #precedence '*'
```

as production definition, the correct syntax tree will be constructed for *-2*3*.

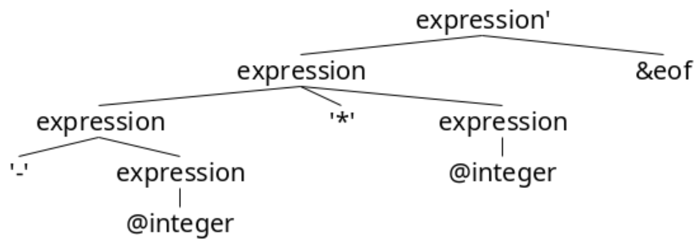


Fig. 15: Correct syntax tree of "-2*3", using a production precedence declaration.

The `#precedence`-directive has to be put right behind the production and its reduction code, so

```
'-' expression         [* @@ = -@2; *]    #precedence '*'
```

could be a valid reduction code submit.

4.5.8. #whitespaces

The `#whitespaces` directives defines symbols to be handled as whitespace, which means they are allowed everywhere between terminals and simply ignored. In programming languages, comments, blanks, tabs and in some languages even line breaks are handled as whitespace.

`#whitespaces` is one of the most influencing UniCC directives, and works different in both of the two parser construction modes.

4.5.8.1. #whitespaces in sensitive mode

If `#whitespaces` is used in sensitive mode (default), it allows for symbols of any kind, including nonterminal symbols. This means, that whitespace can be constructed from sub-grammars which are made-up of entire grammatical rules and their subsequent rules. Using this parser construction mode, UniCC rewrites the entire grammar to make whitespace become valid in situations after a regular terminal is shifted, or at the beginning of the grammar. This grammar revisions comes visible when calling UniCC with the `-G` option, to print out the revised grammar that is used to finally construct the parse tables.

For example, the simple grammar

```
#!mode sensitive;
```

```
#whitespaces whitespace ;

@COMMENT '/' !'/'+ '/' ;

start$ -> "Hello" "World";
whitespace -> ' ' | '\t' | @COMMENT;
```

yields in a revised grammar UniCC outputs as

```
$ unicc -G w_sensitive.par

GRAMMAR

    whitespace [ ' ' '\t' @COMMENT ] lexem:1 prec:0 assoc:N v:(null)
      (1) -> ' '
      (2) -> '\t'
      (3) -> @COMMENT

    start [ "Hello" ] lexem:0 prec:0 assoc:N v:(null)
      (0) -> Hello' World'

    start' [ "Hello" ] lexem:0 prec:0 assoc:N v:(null)
      (4) -> start &eof

    &whitespace [ ' ' '\t' @COMMENT ] lexem:1 prec:0 assoc:N v:(null)
      (5) -> whitespace

    &whitespace+ [ ' ' '\t' @COMMENT ] lexem:1 prec:0 assoc:N v:(null)
      (6) -> &whitespace+ &whitespace
      (7) -> &whitespace

    &whitespace* [ ' ' '\t' @COMMENT ] lexem:1 prec:0 assoc:N v:(null)
      (8) -> &whitespace+
      (9) ->

    Hello' [ "Hello" ] lexem:0 prec:0 assoc:N v:(null)
      (10) -> "Hello" &whitespace*

    World' [ "World" ] lexem:0 prec:0 assoc:N v:(null)
      (11) -> "World" &whitespace*

    start'' [ ' ' '\t' @COMMENT "Hello" ] lexem:0 prec:0 assoc:N v:(null)
      (12) -> &whitespace* start'

$
```

which introduces new virtual nonterminals *&whitespace*, *&whitespace+*, *&whitespace**, *Hello'* and *start'*.

But it uses a fully-fledged nonterminal *whitespace* which could possibly exist of very flexible, grammatical constructions. The behavior of terminal symbols within the revision can also be influenced by the directives `#lexeme`, `#fixate` and `#lexeme separation`.

Whitespace becomes part of the grammar, but its existence is hidden from the semantic actions of the grammar developer. There is no obvious indicator if whitespace handling has been done, this can only be indicated on demand.

4.5.8.2. #whitespaces in insensitive mode

In the insensitive parser construction mode, the `#whitespaces` directive only accepts terminal symbols. Trying to specify a nonterminal at `#whitespaces` results in an error, because whitespace consumed by the parser can't be handled here. The grammar is not rewritten as in sensitive mode.

All specified terminal symbols are flagged as whitespace, and are simply ignored when read by the resulting lexical analyzer. The grammar is not rewritten, but the grammar developer is limited to only use character-classes, strings and regular expressions as whitespaces symbols, without a flexible grammar beyond.

Resulting grammars result in much lesser states and will be parsed faster.

4.5.9. #lexeme

The `#lexeme` directive can only be used within the sensitive parser construction mode. It configures nonterminal symbols including all their subsequent productions and terminal-/nonterminal calls to be handled as coherent lexical units, the so called *lexeme*. `#lexeme` directly influences the grammar revision process. Each as `#lexeme` configured symbol is handled like a terminal-symbol within the whitespace grammar revision, so whitespace is allowed behind a lexem, but not within.

Given is this example grammar:

```
#whitespaces ' ' ;

start$ -> "print" value ;

value -> ident | integer ;

ident -> 'A-Za-z_' ident | 'A-Za-z_' ;

integer -> '0-9' integer | '0-9' ;
```

When this grammar is passed to UniCC, it will produce a parser that allows to parse statements like

```
print hello
print 4711
print h e ll     o  wor ld
print 1  675467     9  123
```

The two last lines are internally parsed as "helloworld" and "16754679123", but this is not the desired syntax for our language. Adding a lexem configuration

```
#lexeme ident integer ;
```

to this grammar yields in a different, correct parser, which only accepts the lexeme we want to allow for.

The difference that is done during the automatic revision of the grammar caused by the `#whitespaces`-directive in combination with the sensitive mode can be made visible using the `-G` command-line flag to let UniCC print out the final grammar. The initial version of the revised grammar is this one, if no lexeme configuration is done.

```
GRAMMAR
```

```
    start [ "print" ] lexem:0 prec:0 assoc:N v:(null)
      (0) -> print' value

    value [ 'A-Z_a-z' '0-9' ] lexem:0 prec:0 assoc:N v:(null)
      (1) -> ident
      (2) -> integer

    ident [ 'A-Z_a-z' ] lexem:0 prec:0 assoc:N v:(null)
      (3) -> A-Z_a-z' ident
      (4) -> A-Z_a-z'

    integer [ '0-9' ] lexem:0 prec:0 assoc:N v:(null)
      (5) -> 0-9' integer
      (6) -> 0-9'

    start' [ "print" ] lexem:0 prec:0 assoc:N v:(null)
      (7) -> start &eof

    &whitespace [ ' ' ] lexem:1 prec:0 assoc:N v:(null)
      (8) -> ' '

    &whitespace+ [ ' ' ] lexem:1 prec:0 assoc:N v:(null)
      (9) -> &whitespace+ &whitespace
      (10) -> &whitespace

    &whitespace* [ ' ' ] lexem:1 prec:0 assoc:N v:(null)
      (11) -> &whitespace+
      (12) ->

    print' [ "print" ] lexem:0 prec:0 assoc:N v:(null)
      (13) -> "print" &whitespace*

    0-9' [ '0-9' ] lexem:0 prec:0 assoc:N v:(null)
      (14) -> '0-9' &whitespace*

    A-Z_a-z' [ 'A-Z_a-z' ] lexem:0 prec:0 assoc:N v:(null)
      (15) -> 'A-Z_a-z' &whitespace*

    start'' [ ' ' "print" ] lexem:0 prec:0 assoc:N v:(null)
      (16) -> &whitespace* start'
```

The rewritten nonterminals derived from the terminal symbols are extended by their origin symbol name with an appended quote ('). In this case, the character-class terminal `'A-Za-z_'` is rewritten to a new nonterminal with the name *A-Za-z_'*, which allows for whitespace behind every character that matches the character-class.

The version with lexemes configured will be a little shorter, because UniCC revises not all terminals of the grammar to allow for whitespace.

```
GRAMMAR

    ident [ 'A-Z_a-z' ] lexem:1 prec:0 assoc:N v:(null)
      (3) -> 'A-Z_a-z' ident
      (4) -> 'A-Z_a-z'

    integer [ '0-9' ] lexem:1 prec:0 assoc:N v:(null)
      (5) -> '0-9' integer
      (6) -> '0-9'
    start [ "print" ] lexem:0 prec:0 assoc:N v:(null)
      (0) -> print' value

    value [ 'A-Z_a-z' '0-9' ] lexem:0 prec:0 assoc:N v:(null)
      (1) -> ident'
      (2) -> integer'

    start' [ "print" ] lexem:0 prec:0 assoc:N v:(null)
      (7) -> start &eof

    &whitespace [ ' ' ] lexem:1 prec:0 assoc:N v:(null)
      (8) -> ' '

    &whitespace+ [ ' ' ] lexem:1 prec:0 assoc:N v:(null)
      (9) -> &whitespace+ &whitespace
      (10) -> &whitespace

    &whitespace* [ ' ' ] lexem:1 prec:0 assoc:N v:(null)
      (11) -> &whitespace+
      (12) ->

    print' [ "print" ] lexem:0 prec:0 assoc:N v:(null)
      (13) -> "print" &whitespace*

    ident' [ 'A-Z_a-z' ] lexem:0 prec:0 assoc:N v:(null)
      (14) -> ident &whitespace*

    integer' [ '0-9' ] lexem:0 prec:0 assoc:N v:(null)
      (15) -> integer &whitespace*

    start'' [ ' ' "print" ] lexem:0 prec:0 assoc:N v:(null)
      (16) -> &whitespace* start'
```

Now, nonterminals *integer* and *ident* stay on their own, they are not revised. Instead, there is now a nonterminal *integer'* and *ident'* that allows for whitespace behind the lexeme. With a sharp look to the nonterminals generated to parse the whitespace, it gets more clear that whitespace-related nonterminals are themselves some kind of lexeme.

It is obvious, that the `#lexeme` directive is one of UniCC's most powerful features in the sensitive parser construction mode, because it enables for richer grammatical constructions than any regular language allows for, with the difference that this constructs are handled as they would lexeme taken from the lexical analyzer.

If a grammar constructs with lexemes produce shift-reduce warnings, `#lexeme` can be used in combination with the `#lexeme separation` directive.

4.5.10. #lexeme separation

The `#lexeme separation` directive can only be used within the sensitive parser construction mode and in combination with the `#lexeme` directive. It is closely related to the `#fixate` directive, and configures all symbols configured as lexemes to be handled like fixate. It is a boolean directive, and can be switched `on` or `off`. Calling it without boolean parameter switches `#lexeme separation` as `on`.

Based on the sample grammar from above, it should be allowed for multiple values where only one value should be allowed yet. This could be done by introducing a new virtual production.

```
start$ -> "print" value+ ;
```

Feeding this to UniCC and with warning reporting switched on, this causes a shift-reduce conflict in two states, because UniCC detects an ambiguous grammar in both lexeme.

```
unicc: warning: state 9: Shift-reduce conflict on lookahead: 'A-Z_a-z'
    (5) ident -> 'A-Z_a-z' .ident
    (6) ident -> 'A-Z_a-z' .      { ' ' '\n' 'A-Z_a-z' '0-9' }

unicc: warning: state 10: Shift-reduce conflict on lookahead: '0-9'
    (7) integer -> '0-9' .integer
    (8) integer -> '0-9' .      { ' ' '\n' 'A-Z_a-z' '0-9' }
```

By using `#lexeme separation on ;`, the warnings will be suppressed, and UniCC always selects the way that input for an lexeme is consumed as much as possible.

4.5.11. #fixate

The `#fixate` directive can only be used within the sensitive parser construction mode. It suppresses shift-reduce warnings that may raise up for some nonterminals caused by the grammar construction, and always prefers to shift. `#fixate` expects a list of nonterminal symbols to be configured as fixed.

4.5.12. #reserve terminals

The `#reserve terminals` directive switches if terminal anomaly detection is performed by UniCC or not. If `#reserve terminals` is switched on, UniCC assumes that terminals based on regular expressions take higher precedence in all places, and does not perform terminal anomaly detection. It is a boolean directive, and can be switched `on` or `off`. Calling it without boolean parameter switches `#reserve terminals` as `on`.

More about terminal anomalies can be found in the section terminal anomalies. The directive can only be used in the sensitive parser construction mode.

4.6. Special symbols

UniCC grammars may contain some special terminal and nonterminal symbols under various circumstances. These special symbols begin with an ampersand `&`, and cannot be expressed in the grammar itself, except the symbol `&error`. Their meaning and usage is listed in the following table below.

Special symbol	Type	Usage
&embedded_<num>	nonterminal	Anonymous nonterminals get this name with a consecutive number.
&eof	terminal	Specifies the end-of-file symbol. This special symbol exists in every grammar, and will be automatically appended to the goal-symbol. The end-of-file symbol itself is determined by the parser, but is not bound to a special character sequence or value. In the UniCC Standard C Parser Template, the end-of-file symbol can be dynamically set to fit to the particular input.
&error	terminal	Specifies the error resynchronization symbol. This symbol is used to resynchronize the parser after getting scrambled by a parse error. The chapter error recovery covers the use and further meaning of the error resynchronization terminal.
&whitespace, &whitespace*, &whitespace+	nonterminal	This group of nonterminals are only inserted in sensitive mode grammars that make use of the `#whitespaces` directive. Calls to this symbol are added to appropriate positions relating the lexeme configuration of the grammar. More about this topic can be read in chapter about the `#whitespaces` directive.

Table 8: Special symbols in UniCC parsers.

4.7. Error recovery

Requirements to modern parsers and compilers expect, that a parse error on wrong input don't causes a final stop of the parser or compiler. As many errors as possible should be reported to the user before a correction of the input and a re-compilation process is invoked. UniCC parsers normally stop on invalid input, respective a parse error. To get a parse error resynchronized and continue parsing, the special symbol `&error` can be used within the grammar.

```
//Some grammar-related directives
#!language      "C";
#whitespaces    ' \t';
#default action [* @@ = @1; *];

#left           '+' '-';
#left           '*' '/';

//Defining the grammar
calculator$     -> expression                [* printf( "= %d\n",
                                                  @expression );
                                                printf( "(%d errors)\n",
                                                  pcb->error_count );
                                             *]
                ;

expression      -> expression '+' expression [* @@ = @1 + @3; *]
                | expression '-' expression  [* @@ = @1 - @3; *]
                | expression '*' expression  [* @@ = @1 * @3; *]
                | expression '/' expression  [* @@ = @1 / @3; *]
                | '(' expression ')'         [* @@ = @2; *]
                | integer
                | &error                     [* printf( "Error: "
                                                    "Expecting integer on "
                                                    "line %d, column %d\n",
                                                    pcb->line, pcb->column );
                                                pcb->error_count++;
                                                @@ = 0;
                                             *]
                ;

integer         -> '0-9'                     [* @@ = @1 - '0'; *]
                | integer '0-9'              [* @@ = @integer * 10 +
                                                     @2 - '0'; *]
                ;

//End of definition
```

Error resynchronization means that the error is handled by the grammar and its semantic actions rather than the parser. In the above example, the parser will, in case of a parse error, try to pop as many items off the parse stack until it finds the error resynchronization token, `&error`. This is shifted then, and the parser comes into a new, valid state that is covered by the grammar, so the production with the error state takes place. The parse error had been recovered, respective it is not a parse error anymore. By modifying the actions in the semantic action of a error state, error messages can still be forced and errors counted, as it is done in above example. But the parser continues, and will output correct values on wrong input, e.g. "2*5+x".

```
$ unicc -w error.par
$ cc -o error error.c
$ echo -n "2*5+x" | ./error -s
Error: Expecting integer on line 1, column 5
```

```
= 10
(1 errors)
$
```

`&error` can be used on several positions, so it is also possible to report errors with it or to silently fix them and let the parser continue. It must be mentioned, that the way how parse errors and the error synchronization token is handled relies on the used target language parser template, and is not a direct feature of the UniCC Parser Generator itself.

4.8. Abstract Syntax Tree Notations

As a new feature introduced with UniCC version 1.3, abstract syntax tree notations can be specified within a grammar. These notations instruct the resulting parser to automatically construct an abstract syntax tree (AST) during the full parse tree construction. The feature of constructing abstract syntax trees rather than performing semantics during the parse process is a new approach which will become more important in a next UniCC major release (see https://github.com/phorward/unicc/wiki/UniCC-v2).

For AST notations, the grammar definition language has been slightly modified to allow for some more distinctive syntax. The following example implements the well-known expression grammar without any semantics. The benefit is, that such kinds of grammars can be implemented in any UniCC target without any modification.

```
#whitespaces    ' \t';

@integer        '0-9'+                    = int;

#left           '+' '-';
#left           '*' '/';

//Defining the grammar
calculator$     : expression
                ;

expression      : expression '+' expression  = add
                | expression '-' expression  = sub
                | expression '*' expression  = mul
                | expression '/' expression  = div
                | '(' expression ')'
                | @integer
                ;
```

That's all. This example already uses the new-style syntax, where `->` is replaced by `:` and the new operators `:=` and `=` take place.

The equal-operator (=) can be used on productions and token definitions to define a generated AST node, while the defines-equal-operator (:=) is used to specify the defined nonterminal as name for the AST node. In case of multiple nonterminal definitions, the first will be used.

Compile and run with C:

```
$ unicc expr.ast.par
$ cc -o expr.ast expr.ast.c
$ ./expr.ast -l

ok
1+2*3+4
add
 add
  int (1)
  mul
   int (2)
   int (3)
 int (4)
```

Compile and run with Python:

```
$ unicc -l python expr.ast.par
$ python expr.ast.py
>1+2*3+4
add
 add
  int (1)
  mul
   int (2)
   int (3)
 int (4)
```

Please note, that this feature is still in beta, and will become useful in an upcoming UniCC version.

5. C parser target

This part of the manual relates to the shipped C parser target from version 1.3 and higher.

5.1. Overview

The C parser target shipped with UniCC enables target code generation support for the C programming language. The template also provides facilities for further grammar processing and integration of the generated parser modules with other C modules.

The C parser target is also used by UniCC itself for bootstrap, meaning that UniCC constructs its own parser out of itself.

5.2. Features

Below is the feature set the C parser target provides.

- Well tested, feature proven, used by UniCC's own grammar parser
- Platform and C-compiler independent, based on the C standard library only
- ANSI C89 and C99 compliant
- Thread-safe, parsers can recursively be called using an extendible parser control block (pcb)
- Wide-character and UTF-8 unicode input support
- Trace and stack trace facilities
- Build-in error recovery
- Automatic construction of an AST data structure when abstract syntax tree notation is provided
- Build-in syntax tree generator
- Symbol and production tables for debug and syntax tree construction
- Provides a default parser test environment if no semantic code is given
- Dynamic end-of-file behavior

5.3. Contributions

Any contributions to the parser template are welcome. Contributions can be sent to us, so we will integrate them into the distribution version of UniCC and the standard template, if they are necessary and useful for everyone.

5.4. Use and configuration

As the C standard parser template has been developed simultaneously with UniCC itself and results in UniCC's own grammar parser, it is the best integrated parser template so far.

To use it with a parser, the top-level directive

```
#!language "C" ;
```

can be set, and `c.tlt` must be in the path directed by `UNICC_TPLDIR`.

Alternatively, the `--language` or `-l` parameter can be set.

Without any configuration, the C standard template results in parsers that are perfectly useable for grammar testing, by reading character by character from stdin. Errors are printed to stderr. To integrate the parser into other execution environments, some more configuration is necessary. Configuration of the parser is done using C pre-processor directives only, which must be defined in the header-code to disable their default declarations.

All symbols of the parsers which are or could become visible from outside the parser module contain a prefix value, which is configured using the `#prefix` directive.

5.5. Value-types in semantic actions

The C parser target sets by default the data-type **int** for all symbols used in the grammar. A character-class terminal always returns the matched character as int-value.

```
test$ -> '0-9A-Za-z'       [* printf( "= %d %c\n", @1, (char)@1 ); *]
```

In semantic actions submitted to regular-expression terminals, the following table shows data-types and description.

Macro	C-Datatype	Usage
`@@`	*defined by the terminal symbol*, default is int	Defines the return value that is associated with the terminal. This variable is of the same type that is specified for the symbol or the default type.
`@>`	UNICC_SCHAR*	Calls the function *@@prefix_lexem()*, which returns the semantic content of the scanned lexem as UNICC_SCHAR-pointer. UNICC_SCHAR is of type **wchar_t** if UNICC_WCHAR is switched, else of type **char**. In UTF-8-mode, the memory behind this pointer is allocated, but memory allocation is done by the parser template automatically, and connected to the *lexem* member of the parser control block. The pointer is always a zero-terminated string.
`@<`	int	Defines the length of the matched string, in characters. This is in UTF-8 mode also the amount of characters, not the amount of bytes (!). To get the number of characters in UTF-8 mode, call `strlen( @> )`.

Table 9: Macros in lexer-related actions and their behavior in the C parser target.

The following example shows how to use the macros.

```
@ident<char*>  'A-Za-z'+    [*
                                /* This works in UTF-8 mode and
                                   without Unicode support */
                                printf( "ident: length %d >%s<\n", @<, @> );

                                /* This works in wide-character mode */
                                printf( "ident: length %d >%S<\n", @<, @> );

                                #if UNICC_ON_SHIFT
                                /* Copy current token */
                                @@ = strdup( @> );
                                #endif
                            *];
```

5.5.1. Semantic terminal selection

The feature of semantic terminal determination is turned off by default within the C parser target, because it may be the source for memory leaks in the sensitive parser mode. It can be switched on by setting the define `UNICC_SEMANTIC_TERM_SEL` to 1. It is strongly recommended to use the feature of semantically determined nonterminals instead.

If the feature of semantically determined terminal recognition is turned on, semantic actions may be run multiple times in some cases, to correctly identify the current input in combination with semantically determined terminal selection actions. For these special cases, code blocks that allocate memory for token return should be expressed in blocks of

```
#if UNICC_ON_SHIFT
.
.
.
#endif
```

conditioned directives, to avoid multiple memory allocations and lost of allocated pointer addresses. The allocation of memory in above example could be performed like this.

```
@ident name <char*> 'A-Za-z'+
                            [*
                                /* Semantic terminal determination */
                                if( strcmp( @>, "name" ) == 0 )
                                    @!symbol:name;

                                #if UNICC_ON_SHIFT
                                /* Copy current token */
                                @@ = strdup( @> );
                                #endif
                            *];
```

This causes the block within UNICC_ON_SHIFT only will be executed if the token had been clearly identified before and is stacked.

5.6. The parser control block (@@prefix_pcb)

The parser control block is the main data structure which is passed to all parser-related functions in order to provide any runtime information on the current parser state, input buffering and much more. It is always accessible within all configuration macros as well as semantic codes, and simply identified by a variable called `pcb`. The parser control block itself is a typedef that is referred as `@@prefix_pcb`. `@@prefix` is replaced by the value specified at the #prefix directive in UniCC, if not specified, it will be replaced by nothing (empty string), so `_pcb` will be the default type name.

Using the `#pcb` directive as described in the UniCC reference above, individual members of the parser control block can be defined, which avoids the problem of using global variables; Global variables must not be used in the standard C parser. This makes it entirely thread-safe, recursive calls of the parser are possible. If there is the need to pass on any values or pointers to subsequent parts of the parse-tree that hold global run-time information of the parser, they should be put into the parser control block using the `#pcb` directive.

Its also strongly recommended to put the input file or buffer pointer into the parser control block if the parser's input should be read from another source than stdin. If this is the case, make sure UNICC_GETINPUT is defined correctly.

Most of the members of the parser control block are used internally, but some of them may also be used (please as read-only!) by the compiler writer. Modification of these initial members during runtime is not recommended.

```
/* Parser Control Block */
typedef struct
{
    /* Stack */
    @@prefix_tok*       stack;
    @@prefix_tok*       tos;

    /* Stack size */
    unsigned int        stacksize;

    /* Values */
    @@prefix_vtype      ret;
    @@prefix_vtype      test;

    /* State */
    int                 act;
    int                 idx;
    int                 lhs;

    /* Lookahead */
    int                 sym;
    int                 old_sym;
    unsigned int        len;

    /* Input buffering */
    UNICC_SCHAR*        lexem;
    UNICC_CHAR*         buf;
    UNICC_CHAR*         bufend;
    UNICC_CHAR*         bufsize;

    /* Lexical analysis */
    UNICC_CHAR          next;
    UNICC_CHAR          eof;
    UNICC_BOOLEAN       is_eof;

    /* Error handling */
    int                 error_delay;
    int                 error_count;

    unsigned int        line;
    unsigned int        column;

#if UNICC_SYNTAXTREE
    /* Syntax tree */
    @@prefix_syntree*   syntax_tree;
#endif

    @@prefix_ast*       ast;

    /* User-defined components */
    @@pcb

} @@prefix_pcb;
```

Member	Type	Content
act	int	The current action to perform (shift, reduce, shift&reduce).
buf	UNICC_CHAR*	Holds the current input buffer. This input buffer holds all the characters necessary to identify the current input token.
bufend	UNICC_CHAR*	Pointer to the end of the input buffer, for faster data appending operations.
bufsize	UNICC_CHAR*	The pointer to the last character of the input buffer. This is used when input buffer reallocation is needed.
column	unsigned int	Contains the current column within the input, beginning from the current line (`line` member variable). First character is 1. Should be used for error reporting.
eof	UNICC_CHAR*	Defines the value of the end-of-file character. This is -1 by default (EOF), but can be explicitly set to any other value. This member can be changed, and defines the dynamic end-of-file handing.
error_count	int	Current number of errors.
error_delay	int	This is set during error recovery, to reduce inherited errors.
idx	int	Index of reduced production.
is_eof	UNICC_BOOLEAN	Defines if end-of-file character has already been read.
len	unsigned int	Holds the length of the matching string in the current input buffer.
lexem	UNICC_SCHAR*	Retrieves a semantic string pointer to the current input. This member is filled by the `@@prefix_lexem()` function and relates to the input type switching (UTF-8 or Unicode).
line	unsigned int	Contains the current line number within the input. It can be used for error reporting in combination with the `column` member variable. First line begins at 1.
lhs	int	Left-hand side index during reduction.
next	UNICC_CHAR*	Temporary character holding space.
old_sym	int	Holds the ID of the old lookahead symbol during error recovery. It will automatically be re-used and reset.
ret	@@prefix_vtype	Last return value of reduction action or symbol to be shifted.
stack	@@prefix_tok*	The parser state and value stack.

stacksize	unsigned int	Variable to determine the maximum stack size of *stack*.
sym	int	Holds the ID of the lookahead symbol which is the current token.
test	@@prefix_vtype	A zero test value.
tos	@@prefix_tok*	Top of stack pointer.
syntax_tree	@@prefix_syntree*	Receives the pointer of the root node of an automatically constructed syntax tree. This member only exists if UNICC_SYNTAXTREE is defined.
ast	@@prefix_ast*	Abstract syntax tree root node.

Table 10: Member variables of @@prefix_pcb.

To avoid compile errors, names differing from the above ones must be chosen to add members to the parser control block structure. UniCC itself does not parse or check this, because this is a template-related problem.

5.7. Additional data structures

There are also some additional data structures used in the C parser target. Knowledge on them is only required when interested in the syntax tree construction feature.

5.7.1. The value type structure (@@prefix_vtype)

`@@prefix_vtype` is the union that will hold a semantic value on the parse stack and within the syntax tree. If there is only one data-type used in the template, `@@prefix_vtype` is only an alias for this data-type, else it is an union containing a member called `value_<id>` for every value data type, where `<id>` is an index counted from 0 for every type. This union is automatically constructed by the code generator in UniCC.

5.7.2. The stack token description structure (@@prefix_tok)

The parse stack and the syntax tree generator make use of the `@@prefix_tok`-structure. This structure contains information about the symbol, its semantic value and informations about state and position in the input.

```
typedef struct
{
    @@prefix_vtype      value;

    @@prefix_syminfo*   symbol;
    int                 state;
    unsigned int        line;
    unsigned int        column;
} @@prefix_tok;
```

Member	Type	Content
column	unsigned int	The column of the symbol occurrence in the input.
id	int	The symbol id, which refers to the stack symbol. Every symbol, terminal or nonterminal, has its own id.
line	unsigned int	The line of the symbol occurrence in the input.
symbol	@@prefix_syminfo*	Pointer to the symbol entry in the parser symbol table. This symbol table contains the name and additional informations about the symbol.
state	int	The state that was current when the symbol was pushed.
value	@@prefix_vtype	The semantic value that is associated with the symbol.

Table 11: Member variables of @@prefix_tok.

5.7.3. The symbol information table (@@prefix_syminfo)

Parsers emitted by the C parser target contain a symbol table holding information about the grammatical symbols used in the parser. This symbol table is used by the parser itself to get additional informations about symbols, e.g. its type or if its configured as whitespace. For debug purposes, this table also contains the original name of the symbol as defined in UniCC.

The grammar symbol table is a structure of type `@@prefix_syminfo`, and can be accessed via a static, global variable `@@prefix_symbols` in within the parser module.

```
/* Typedef for symbol information table */
typedef struct
{
    char*                   name;
    char*                   emit;
    int                     type;
    UNICC_BOOLEAN           lexem;
    UNICC_BOOLEAN           whitespace;
    UNICC_BOOLEAN           greedy;
} @@prefix_syminfo;
```

Member	Type	Content
greedy	int	Defines if the symbol (regular expression terminals only) should be scanned in greedy or non-greedy mode. 1 if true, 0 else.
lexem	int	Defines a nonterminal as a lexem. 1 if true, 0 else.
name	char*	The name of the symbol, as defined in the UniCC parser definition file.
emit	char*	String to be emitted by a particular symbol in abstract syntax tree construction.
type	int	Gets the symbol type. 0 for nonterminal, 1 for character-class terminal, 2 for regular-expression terminal (this includes strings!), 3 for special terminal (e.g. the error resync token or end-of-file symbol).
whitespace	int	Defines a symbol that is whitespace. 1 if true, 0 else.

Table 12: Member variables of @@prefix_syminfo.

5.7.4. The production information table (@@prefix_prodinfo)

All parsers constructred via the C parser target also contain a production information table holding information about the productions used in the parser. This production information table is used by the parser itself to get additional informations about its productions.

The production information table is a structure of type `@@prefix_prodinfo` and can be referred as a static, global variable called `@@prefix_productions`.

```
/* Typedef for production information table */
typedef struct
{
    char*                   definition;
    char*                   emit;
    int                     length;
    int                     lhs;
} @@prefix_prodinfo;
```

Member	Type	Content
definition	char*	A string representing the production's definition as expressed in the UniCC parser definition.
emit	char*	String to be emitted by a particular production reduction in abstract syntax tree construction.
length	int	The length of the production. This is the number of symbols on the right-hand side.
lhs	int	The id of the default left-hand side symbol (indexing an entry in the symbol information table).

Table 13: Member variables of @@prefix_prodinfo.

5.8. Unicode support

The C parser target supports Unicode in several ways. Unicode support means that it is capable to consume Unicode-characters as input.

In general, the template supports three types of input methods.

Type	Define switch	Description	UNICC_CHAR (internal)	UNICC_SCHAR (external)
ASCII	-	Unicode-less mode, works with symmetric 8-bit ASCII input only.	char	char
UTF-8 (default)	UNICC_UTF8	UTF-8 mode supporting 8-bit asymmetric input sequences.	wchar_t	char
Wide-character	UNICC_WCHAR	Wide-character 32-bit symmetric input mode.	wchar_t	wchar_t

Table 14: Input type modes of the C parser target.

To configure a parser template for a specific mode, the defines above (UNICC_UTF8, UNICC_WCHAR) must be set to 1. If both are set to 0, ASCII mode will be used. The data type of both UNICC_CHAR and UNICC_SCHAR depend on these switches. By default, UNICC_UTF8 is switched as 1, so that UTF-8 input processing mode will be entered.

5.9. Abstract syntax tree construction

With the availability of the new abstract syntax tree notation feature provided with UniCC version 1.3 and higher, the C parser target constructs a browsable abstract syntax tree with the following node structure:

```
/* Abstract Syntax Tree */
typedef struct @@prefix_AST @@prefix_ast;

struct @@prefix_AST
{
    char*           emit;
    UNICC_SCHAR*    token;

    @@prefix_ast*   parent;
    @@prefix_ast*   child;
    @@prefix_ast*   prev;
    @@prefix_ast*   next;
};
```

The simple print function to browse the AST is this:

```
void print_ast( @@prefix_ast* node )
{
    while( node )
    {
        if( node->token && strcmp( node->emit, node->token ) != 0 )
            printf( " (%s)", node->token );

        printf( "\n" );

        print_ast( stream, node->child );
        node = node->next;
    }
}
```

The target provides the following functions:

- `@@prefix_ast_print( FILE* stream, @@prefix_ast* node )` to print the AST or parts of it
- `@@prefix_ast_free( @@prefix_ast* node )` to recursively free any consumed memory.

5.10. The build-in syntax tree visualizer (deprecated)

The C parser target features a build-in syntax tree visualizing automatism. This automatism can be enabled by defining `UNICC_SYNTAXTREE` to 1.

The core aspect of this feature is to retrieve a visual tree representation of the parsed input, which can be used for different purposes. One example are the graphical syntax trees that are printed in this manual - most of them had been build automatically using this feature, in combination with some SVG rendering tools.

This feature is marked as deprecated. A similar feature is the new abstract syntax tree notation which is always available. It will replace this feature in future.

The syntax tree data structure is made up of elements of type `@@prefix_syntree`.

```
/* Parse tree node */
typedef struct @@prefix_SYNTREE @@prefix_syntree;

struct @@prefix_SYNTREE
{
    @@prefix_tok        symbol;
    UNICC_SCHAR*        token;

    @@prefix_syntree*   parent;
    @@prefix_syntree*   child;
    @@prefix_syntree*   prev;
    @@prefix_syntree*   next;
};
```

Member	Type	Content
child	@@prefix_syntree*	Pointer to the first child-node, if element is not a leaf.
parent	@@prefix_syntree*	Pointer to the parent node, if any. This value is NULL in the root node.
prev	@@prefix_syntree*	Pointer to previous node in the current level.
symbol	@@prefix_tok	A copy of the stack element representing the symbol. It contains all the information that can be found in the @@prefix_tok structure.
token	UNICC_SCHAR*	A copy of the string that represented the token in the input. This is only filled when the symbol is a terminal symbol and a leaf.
next	@@prefix_syntree*	Pointer to next node in the current level.

Table 15: Member variables of @@prefix_syntree.

An example for a rendering function working on this parse tree:

```
void print_syntax_tree( @@prefix_pcb* pcb, @@prefix_syntree* node )
{
    int         i;
    static int  rec;

    if( !node )
        return;

    while( node )
    {
        for( i = 0; i < rec; i++ )
            printf( "." );
        printf( "%s", node->symbol.symbol->name );

        if( node->token )
            printf( ": '%s'", node->token );

        printf( stream, "\n" );

        rec++;
        print_syntax_tree( pcb, node->child );
        rec--;

        node = node->next;
    }
}
```

results in an output like

```
calculator'
.&whitespace*
.calculator
..expression
...expression
....integer'
.....integer
......0-9: '1'
.....&whitespace*
...+'
....+: '+'
....&whitespace*
...expression
....expression
.....integer'
......integer
.......0-9: '2'
......&whitespace*
....*'
.....*: '*'
.....&whitespace*
....expression
.....integer'
......integer
.......0-9: '3'
......&whitespace*
..&eof: ''
```

5.11. The default main()

The C parser target provides a default main()-function.

This function will be made available if no semantic code is submitted to the `#epilogue`-directive, or if UNICC_MAIN is manually configured to 1.

The default main()-function provides a test mode to test the parser behavior. Input is read from stdin. When a parser is compiled with the build-in main()-function, it is enabled to use the following command-line parameters to change the parsers test-mode behavior.

Short option	Long option	Usage
-s	--silent	Run in silent mode, so no additional output is printed.
-e	--endless	Runs the parser in an endless loop. A detected end-of-file (EOF) stops the parser and starts a new parser.
-l	--line-mode	Runs the parser in line mode, where the newline-character is configured as end-of-file token.

Table 16: Command-line options of the build-in main()-function.

If UNICC_SYNTAXTREE is enabled, a syntax-tree will be printed to stderr after the input has been correctly parsed.

Given a grammar

```
//Some grammar-related directives
#!language      "C";
#whitespaces    ' \t';
#lexeme         integer;
#default action [* @@ = @1; *];

#left           '+' '-';
#left           '*' '/';

//Defining the grammar
calculator$     -> expression                [* printf( "= %d\n",
                                                 @expression ); *]
                ;

expression      -> expression '+' expression [* @@ = @1 + @3; *]
                | expression '-' expression  [* @@ = @1 - @3; *]
                | expression '*' expression  [* @@ = @1 * @3; *]
                | expression '/' expression  [* @@ = @1 / @3; *]
                | '(' expression ')'         [* @@ = @2; *]
                | integer
                ;

integer         -> '0-9'                     [* @@ = @1 - '0'; *]
                | integer '0-9'              [* @@ = @integer * 10 +
                                                     @2 - '0'; *]
                ;

//End of definition
```

and compiling this with

```
unicc -w expression.par
cc -o expression expression.c
```

the parser test mode can be invoked with

```
./expression -l
```

If `-l` is not submitted, a parse error occurs when input is read via stdin from keyboard, and the enter-key is pressed. Writing the expression to be parsed to a file will be consumed without `-l`.

```
echo -n "1+2*3" >test.expr
./expression <test.expr
```

or simply

```
echo -n "1+2*3" | ./expression
```

5.12. General parser invocation (@@prefix_parse())

The general parser invocation function of the C standard parser template is called `@@prefix_parse()`, where `@@prefix` will be replaced by the prefix specified at the #prefix-directive. If no prefix is given, the function is just `_parse()`. If the parser function is called from within a function defined in the parser definition file itself, it is still possible to write `@@prefix_parse()` also in the semantic code. The `@@prefix` variable will be replaced with its correct content by the code generator when the code is build.

The function's prototype is

```
UNICC_STATIC @@prefix_vtype @@prefix_parse( @@prefix_pcb* pcb );
```

The function only requests for one parameter, which is a structure of the parser control block. The structure must be initialized to zero, and user-defined values must be filled correctly before the parse-function is invoked. Else, the parser will come into unpredictable states and parse errors, or simply cores. It is also possible to pass `(@@prefix_pcb*)NULL` to the parser invocation function. In this case, the function will allocate a parser control block structure on its own, but this is only useful in validating parsers that do not perform much semantic actions working on pointers of the parser control block.

The parser-function returns a value of kind `@@prefix_vtype`, which is the return value of the goal symbol. To find out if errors occurred, the variable `error_count` from the parser control block should be checked.

5.13. UTF-8 character fetch function (@@prefix_utf8_getchar())

If the parser is compiled with UNICC_UTF8-enabled (default), the macro UNICC_GETINPUT directs by default to the function *@@prefix_utf8_getchar(getchar)*. The function reads characters from a character retrieval function (by default `getchar()`) until a valid UTF-8 sequence was matched. It can be used by individual tasks also.

The function's prototype is

```
UNICC_STATIC UNICC_CHAR @@prefix_utf8_getchar( int (*getfn)() );
```

The function expects a function pointer as parameter to a function returning one byte per character. `@@prefix_utf8_getchar()` then automatically calls `getfn` until the correct number of sequences is taken to match one valid UTF-8 character.

5.14. Take current lexical value (@@prefix_lexem())

The function `@@prefix_lexem()` has the purpose to return the current input token recognized from a terminal symbol as UNICC_SCHAR*-pointer that can be used in semantic actions.

The function's prototype is

```
UNICC_STATIC UNICC_SCHAR* @@prefix_lexem( @@prefix_pcb* pcb );
```

It should not be called in reduction actions, only on shift-operations, respective in terminal actions. The `@>` macro within semantic terminal actions calls `@@prefix_lexem()` internally. The function allocates memory only in UTF-8 mode, all other input modes force only a pointer mapping, so no memory allocation is done. Memory management will be taken automatically, by the parser, so it is not required to care about.

5.15. Configuration macro reference

This short reference should bring an overview of the provided macros that can be pre-defined in the C parser target, and their behavior. All macros within the UniCC C Standard Template can be overridden when defined in the `#prologue` semantic code area or at compiler invocation. If not submitted, the C target forces their definition to default values.

5.15.1. UNICC_CLEARIN

Clears buffered input of the current token. This macro requires the parser control block as its parameter, and is pre-defined as

```
#define UNICC_CLEARIN( pcb )   @@prefix_clear_input( pcb )
```

so it invokes the build-in function `_clear_input`. It can be re-defined by any desired task, but its behavior must complain with the one given by `_clear_input` and the lexer functions reading the input.

5.15.2. UNICC_DEBUG

Switches parser trace, which is written to stderr. The following table gives information about the trace levels to be set by defining UNICC_DEBUG to a desired debug level depth.

UNICC_DEBUG level depth	Description
0	No debug (default)
1	Enables parser and error recovery debugging
2	Additionally, enables debugging for the lexical analyzer
3	Additionally, enables debugging for the input buffering
4	Additionally, enables debugging for single character fetching (only in UTF-8 mode)

Table 17: C parser target debug levels.

Enable debug mode by setting

```
#define UNICC_DEBUG 1
```

To show stack contents, also switch on UNICC_STACKDEBUG. Here is an example for a debug mode level 1 session with stack debug enabled.

```
$ unicc -w debug.par
$ cc -o debug -DUNICC_DEBUG=1 -DUNICC_STACKDEBUG=1 debug.c
$ echo -n "1+2*3" | ./debug -s
debug: current token 7 (0-9)
debug: sym = 7 (0-9) [len = 1] tos->state = 0 act = reduce idx = 13
debug: Stack Dump: 0
debug: << reducing by production 13 (&whitespace* -> )
debug: after reduction, shifting nonterminal 14 (&whitespace*)
debug: current token 7 (0-9)
debug: sym = 7 (0-9) [len = 1] tos->state = 2 act = shift idx = 7
debug: Stack Dump: 0 2 (&whitespace*)
debug: >> shifting terminal 7 (0-9)
debug: << reducing by production 7 (integer -> '0-9')
debug: after reduction, shifting nonterminal 9 (integer)
```

```
debug: current token 1 (+)
debug: sym = 1 (+) [len = 1] tos->state = 3 act = reduce idx = 13
debug: Stack Dump: 0 2 (&whitespace*) 3 (integer)
debug: << reducing by production 13 (&whitespace* -> )
debug: after reduction, shifting nonterminal 14 (&whitespace*)
debug: << reducing by production 20 (integer' -> integer &whitespace*)
debug: after reduction, shifting nonterminal 21 (integer')
debug: << reducing by production 6 (expression -> integer')
debug: after reduction, shifting nonterminal 11 (expression)
debug: current token 1 (+)
debug: sym = 1 (+) [len = 1] tos->state = 4 act = shift/reduce idx = 7
debug: Stack Dump: 0 2 (&whitespace*) 4 (expression)
debug: >> shifting terminal 1 (+)
debug: current token 7 (0-9)
debug: sym = 7 (0-9) [len = 1] tos->state = 7 act = reduce idx = 13
debug: Stack Dump: 0 2 (&whitespace*) 4 (expression) 7 (+)
debug: << reducing by production 13 (&whitespace* -> )
debug: after reduction, shifting nonterminal 14 (&whitespace*)
debug: << reducing by production 14 (+' -> '+' &whitespace*)
debug: after reduction, shifting nonterminal 15 (+')
debug: current token 7 (0-9)
debug: sym = 7 (0-9) [len = 1] tos->state = 10 act = shift idx = 7
debug: Stack Dump: 0 2 (&whitespace*) 4 (expression) 10 (+')
debug: >> shifting terminal 7 (0-9)
debug: << reducing by production 7 (integer -> '0-9')
debug: after reduction, shifting nonterminal 9 (integer)
debug: current token 3 (*)
debug: sym = 3 (*) [len = 1] tos->state = 3 act = reduce idx = 13
debug: Stack Dump: 0 2 (&whitespace*) 4 (expression) 10 (+') 3 (integer)
debug: << reducing by production 13 (&whitespace* -> )
debug: after reduction, shifting nonterminal 14 (&whitespace*)
debug: << reducing by production 20 (integer' -> integer &whitespace*)
debug: after reduction, shifting nonterminal 21 (integer')
debug: << reducing by production 6 (expression -> integer')
debug: after reduction, shifting nonterminal 11 (expression)
debug: current token 3 (*)
debug: sym = 3 (*) [len = 1] tos->state = 16 act = shift/reduce idx = 9
debug: Stack Dump: 0 2 (&whitespace*) 4 (expression) 10 (+') 16 (expression)
debug: >> shifting terminal 3 (*)
debug: current token 7 (0-9)
debug: sym = 7 (0-9) [len = 1] tos->state = 9 act = reduce idx = 13
debug: Stack Dump: 0 2 (&whitespace*) 4 (expression) 10 (+') 16 (expression) 9 (*)
debug: << reducing by production 13 (&whitespace* -> )
debug: after reduction, shifting nonterminal 14 (&whitespace*)
debug: << reducing by production 16 (*' -> '*' &whitespace*)
debug: after reduction, shifting nonterminal 17 (*')
debug: current token 7 (0-9)
debug: sym = 7 (0-9) [len = 1] tos->state = 13 act = shift idx = 7
debug: Stack Dump: 0 2 (&whitespace*) 4 (expression) 10 (+') 16 (expression) 13 (*')
debug: >> shifting terminal 7 (0-9)
debug: << reducing by production 7 (integer -> '0-9')
debug: after reduction, shifting nonterminal 9 (integer)
debug: current token 8 (&eof)
debug: sym = 8 (&eof) [len = 0] tos->state = 3 act = reduce idx = 13
debug: Stack Dump: 0 2 (&whitespace*) 4 (expression) 10 (+') 16 (expression) 13 (*') 3 (integer)
debug: << reducing by production 13 (&whitespace* -> )
debug: after reduction, shifting nonterminal 14 (&whitespace*)
debug: << reducing by production 20 (integer' -> integer &whitespace*)
debug: after reduction, shifting nonterminal 21 (integer')
debug: << reducing by production 6 (expression -> integer')
debug: after reduction, shifting nonterminal 11 (expression)
debug: current token 8 (&eof)
```

```
debug: sym = 8 (&eof) [len = 0] tos->state = 18 act = reduce idx = 3
debug: Stack Dump: 0 2 (&whitespace*) 4 (expression) 10 (+') 16 (expression) 13 (*') 18 (expressi
debug: << reducing by production 3 (expression -> expression *' expression)
debug: after reduction, shifting nonterminal 11 (expression)
debug: current token 8 (&eof)
debug: sym = 8 (&eof) [len = 0] tos->state = 16 act = reduce idx = 1
debug: Stack Dump: 0 2 (&whitespace*) 4 (expression) 10 (+') 16 (expression)
debug: << reducing by production 1 (expression -> expression +' expression)
debug: after reduction, shifting nonterminal 11 (expression)
debug: current token 8 (&eof)
debug: sym = 8 (&eof) [len = 0] tos->state = 4 act = shift idx = 0
debug: Stack Dump: 0 2 (&whitespace*) 4 (expression)
debug: >> shifting terminal 8 (&eof)
debug: << reducing by production 0 (calculator -> expression ~&eof)
debug: after reduction, shifting nonterminal 10 (calculator)
debug: << reducing by production 21 (calculator' -> &whitespace* calculator)
debug: goal symbol reduced, exiting parser
debug: parse completed with 0 errors
= 7
$
```

5.15.3. UNICC_ERROR_DELAY

Defines the error delay that is used to mark a parse error as successfully recovered. If there raise parse errors within this number of shifts after an initial parse error, during error recovery, no following error will be reported. UNICC_ERROR_DELAY is set to 3 shifts by default.

5.15.4. UNICC_GETINPUT

Defines a macro that is called when a character is fetched for buffing in the current input. The C parser target buffers input according to the needs of the lexer. This buffering is done automatically, but it is required to provide a way how characters are read from the input. By default, `UNICC_GETINPUT` is defined as

```
#define UNICC_GETINPUT @@prefix_utf8_getchar( getchar )
```

in UTF-8 mode, and

```
#define UNICC_GETINPUT getchar()
```

in all other modes.

By re-defining it, a function must be given that returns the next character and moves the input pointer one character to the next one. It must be assured, that no more input is returned by `UNICC_GETINPUT` when the end-of-file marker has been reached.

One could define `UNICC_GETINPUT` as

```
#define UNICC_GETINPUT *pcb->inputstring++
```

to read from a buffered string, but this could read over the string's end, when `UNICC_GETINPUT` is called multiple times (which can be the case!). So a definition of

```
#define UNICC_GETINPUT *( *pcb->inputstring ? pcb->inputstring : pcb->inputstring++ )
```

would be more adequate and causes no unwanted effects.

Check out the function @@prefix_utf8_getchar() if UTF-8 input processing is wanted.

5.15.5. UNICC_MAIN

`UNICC_MAIN` controls if the parser features its automatic main function. This feature enables to rapidly prototype a grammar and immediately test its result. The main function calls the parser in an endless loop and allows.

By default, if no semantic code is specified by `#epilogue`, UNICC_MAIN will be configured to 1, so a main function is generated. If an epilogue is given, it is defined automatically to 0 when no previous definition is done. To specify an epilogue and switch the parser's build-in main function on

```
#define UNICC_MAIN 1
```

must be done previously.

5.15.6. UNICC_MALLOCSTEP

Defines the increment size of units allocated in one row in every allocation task. This is done to minimize the amount of malloc/realloc calls, so reallocation of memory is only done when the next step of this number is reached. This macro fits to the reallocation of the input-buffering and stacks. It is configured to 128 by default, so for example 128 bytes for input buffering or 128 elements of the value stack elements in an array are allocated. The next reallocation occurs when this limit of UNICC_MALLOCSTEP is reached again.

5.15.7. UNICC_OUTOFMEM

Is a macro that is called when a malloc/realloc call fails. It can be replaced with a product-related error function call or similar. Its default definition fprintf's a "Memory error" to stderr and exits the program with error code 1.

5.15.8. UNICC_PARSE_ERROR

This macros is invoked on a parse error. It requires the parser control block as its parameter, and is pre-defined with an error text written to stderr.

The UNICC_PARSE_ERROR-macro defaults to

```
#define UNICC_PARSE_ERROR( pcb ) \
    fprintf( stderr, "line %d: syntax error on token '%s'\n", \
    ( pcb )->line, \
        ( ( ( pcb )->sym >= 0 ) ? \
            @@prefix_symbols[ ( pcb )->sym ].name : @@prefix_lexem( pcb ) ) )
```

but can be replaced by any other function call or macro construct.

5.15.9. UNICC_REDUCE, UNICC_SHIFT

Internal use only, may not be changed or redefined.

5.15.10. UNICC_STACKDEBUG

If switched 1 in combination with UNICC_DEBUG, stack content will be printed additionally to stderr.

5.15.11. UNICC_STATIC

A define that holds the keyword `static` by default. It can be configured to be empty or anything else, just in case static data should be made accessible outside the parser module. The parser invocation function is always not declared to be UNICC_STATIC.

5.15.12. UNICC_UTF8

Switches UTF-8 mode on, if configured as 1 (default).

5.15.13. UNICC_WCHAR

Switches wide-character mode on, if configured as 1. If UNICC_WCHAR is set, UNICC_UTF8 will be automatically disabled.

6. C++ parser target

The C++ target is a fork of the C parser target, and allows to compile into C++ source files.

6.1. Overview

For now, input processing is done in the same way as in the C target, but this may be changed in the future. Because the C++ target is a fork of the C target, all macros and features from there should also work here. The major difference between the C and the C++ target is, that latter one emits classes.

Generated parsers in the C++ should behave like parsers for the C target. They use the same algorithms for input caching, so that characters are requested only once from the input source. Input source can be anything that emits characters.

6.2. Features

Parsers generated by the C++ target so far provide

- Established on C++ standard library
- Thread-safe, parsers can recursively be called
- Automatic construction of an AST data structure when abstract syntax tree notation is provided
- Provides a default parser test environment if no semantic code is given
- Dynamic end-of-file behavior

6.3. Contributions

Any contributions to the parser target are welcome. Contributions can be sent to us, so we will integrate them into the distribution version of UniCC and the standard template, if they are necessary and useful for everyone.

6.4. Example with semantic actions

The Python target is currently not covered in detail, so only a short example is presented. For now, it is important that the correct indention level is satisfied, which is a dependency for Python. The AST notations introduced with UniCC v1.3 and higher integrate perfectly with the Python target.

```
//Some grammar-related directives
#!language      "C++";
#whitespaces    ' \t';
#lexeme         integer;
#default action [* @@ = @1; *];

#left           '+' '-';
#left           '*' '/';

#prologue       [*
    #include <stdlib.h>
    #include <stdio.h>
    #include <string.h>
    #include <locale.h>
    #include <iostream>
    *];

//Defining the grammar
calculator$     -> expression                [* std::cout << "= " <<
                                                    @expression << std::endl *]
                ;

expression      -> expression '+' expression [* @@ = @1 + @3 *]
                | expression '-' expression  [* @@ = @1 - @3 *]
                | expression '*' expression  [* @@ = @1 * @3 *]
                | expression '/' expression  [* @@ = @1 / @3 *]
                | '(' expression ')'         [* @@ = @2 *]
                | integer
                ;

integer         -> '0-9'                     [* @@ = @1 - '0' *]
                | integer '0-9'              [* @@ = @integer * 10 + @2 - '0' *]
                ;
```

Compile and run it with

```
unicc expr.cpp.par
g++ -o expr expr.cpp.cpp
```

6.5. Classes

Right now, the C++ targets emits only the class **@@prefix_parser**, which behaves as the parser control block from the C target, but encapsulates all functions a private and public members. The object can be referenced in any semantic actions with the `this` keyword.

Any other data objects known from the C target are remaining as public structures without any object-oriented aspects.

7. Python parser target

This part of the manual relates to the shipped Python parser target, version 0.3, and higher.

7.1. Overview

The Python target enables UniCC target code generation support for the Python programming language (https://python.org).

The target has been implemented in 2017 when UniCC was internally revised. Since version 0.3 shipped with UniCC v1.3, it runs stable and already supports features for abstract syntax tree generation and traversal.

Generated parsers in the Python target mostly behave like parsers for the C target. They use the same algorithms for input caching, so that characters are requested only once from the input source. Input source can be anything that emits characters.

7.2. Features

Parsers generated by the Python target so far provide

- Support for Python 2 and 3
- No dependencies on any third-party library
- Thread-safe, parsers can recursively be called
- Automatic construction of an AST data structure when abstract syntax tree notation is provided
- Provides a default parser test environment if no semantic code is given
- Dynamic end-of-file behavior

Error recovery is currently not implemented.

7.3. Contributions

Any contributions to the parser target are welcome. Contributions can be sent to us, so we will integrate them into the distribution version of UniCC and the standard template, if they are necessary and useful for everyone.

7.4. Example with semantic actions

The Python target is currently not covered in detail, so only a short example is presented. For now, it is important that the correct indention level is satisfied, which is a dependency for Python. The AST notations introduced with UniCC v1.3 and higher integrate perfectly with the Python target.

```
#!language      "python";

#whitespaces    ' \t';
#lexeme         integer;
#default action [*@@ = @1*];

#left           '+' '-';
#left           '*' '/';
```

```
//Defining the grammar
calculator$     -> expression                [*print("= %d" % @expression)*]
                ;

expression      -> expression '+' expression [*@@ = @1 + @3*]
                | expression '-' expression  [*@@ = @1 - @3*]
                | expression '*' expression  [*@@ = @1 * @3*]
                | expression '/' expression  [*@@ = @1 / @3*]
                | '(' expression ')'         [*@@ = @2*]
                | integer
                ;

integer         -> '0-9'                     [*@@ = ord(@1) - ord('0')*]
                | integer '0-9'              [*@@ = @integer * 10 + ord(@2) - ord('0')*]
                ;

//End of definition
```

Compile and run it with

```
unicc expr.py.par
python expr.py.py
```

7.5. Classes

7.5.1. @@prefixNode

This class defines an AST node.

`emit` defines the emitted AST node name, `match` the string from the input. The member `children` is a list of child node objects.

7.5.2. @@prefixParserControlBlock

Defines the parser control block, which saves any current state information of the parser.

7.5.3. @@prefixParser

This is the main parser class.

Its only public function so far is `@@prefixParser.parse()`, which requires an optional parameter defining the input. This parameter can either be of type *str* or *unicode*, or a callable that emits characters. This function is only called once.

8. Appendix I: eXample Programming Language

This is the appendix that includes the full source code of the XPL programming language from the tutorial.

To obtain the latest copy of the source code presented here, clone the source repository with

```
git clone https://github.com/phorward/xpl.git
```

8.1. Makefile

```
XPL             =     xpl

XPL_PARFILE     = xpl.par
XPL_PARSER      = xpl.parser
XPL_PARSER_C    = $(XPL_PARSER).c
XPL_PARSER_H    = $(XPL_PARSER).h
XPL_PARSER_O    = $(XPL_PARSER).o

XPL_SOURCE      =     \
                    xpl.debug.c \
                    xpl.functions.c \
                    xpl.main.c \
                    xpl.program.c \
                    xpl.run.c \
                    xpl.util.c \
                    xpl.value.c \
                    $(XPL_PARSER_C)

XPL_OBJECTS      =     \
                    xpl.debug.o \
                    xpl.functions.o \
                    xpl.main.o \
                    xpl.program.o \
                    xpl.run.o \
                    xpl.util.o \
                    xpl.value.o \
                    $(XPL_PARSER_O)


all: $(XPL)

$(XPL): $(XPL_SOURCE) Makefile
    $(CC) -o $@ $(XPL_SOURCE)

$(XPL_PARSER_C): $(XPL_PARFILE)
    unicc -svwb $(XPL_PARSER) $(XPL_PARFILE)

clean:
    rm -f $(XPL_PARSER_C)
    rm -f $(XPL_PARSER_H)
    rm -f $(XPL_OBJECTS)
    rm -f $(XPL)
```

8.2. xpl.par

```
#!language              "C";

//Meta information
#parser                 "XPL";
#description            "eXample Programming Language";
#copyright              "In the public domain, 2011";
#version                "0.2";
#prefix                 "xpl";

#default action         [* @@ = @1; *];
#default epsilon action [* @@ = 0; *];


//Precedence and associativity
#left               "=";

#left               "=="
                    "!="
                    "<="
                    ">="
                    '>'
                    '<'
                    ;

#left               '+'
                    '-'
                    ;

#left               '*'
                    '/'
                    ;

//Regular expressions
@string<char*>      '"' !'"'* '"'                [* @@ = @>; *]
                    ;

@identifier<char*>  'A-Za-z_' 'A-Za-z0-9_'*     [* @@ = @>; *]
                    ;


//Lexemes
#lexeme             real
                    ;

real<float>         ->       real_integer '.' real_fraction

                            [*  @@ = @real_integer + @real_fraction; *]

                    |       real_integer '.'?

                            [*  @@ = @real_integer; *]

                    |       '.' real_fraction

                            [*  @@ = @real_fraction; *]
                    ;

real_integer<float> ->      real_integer '0-9':dig
```

```
                                [* @@ = 10 * @real_integer + @dig - '0'; *]

                    |           '0-9':dig

                                [*  @@ = @dig - '0'; *]
                    ;

real_fraction<float>->          real_fraction '0-9':dig

                                [* @@ = ( @real_fraction - '0' + @dig ) / 10.0; *]

                    |           '0-9':dig

                                [*  @@ = ( @dig - '0' ) / 10.0; *]
                    ;

//Whitespace grammar construct
#whitespaces        whitespace
                    ;

whitespace          ->          ' \r\n\t'+
                    |           "//" !'\n'* '\n'
                    ;

//Goal symbol
program$            ->          statement*
                    ;

statement           ->          "if" '(' expression ')'

                                (
                                    [*
                                        @@ = xpl_emit( pcb->prog, XPL_JPC, 0 );
                                    *]
                                ):jpc

                                    statement

                                (
                                    "else"

                                    (
                                        [*
                                            @@ = pcb->prog->program_cnt;
                                            xpl_emit( pcb->prog, XPL_JMP, 0 );
                                        *]
                                    ):jmp

                                    statement

                                    [* @@ = @jmp; *]

                                |

                                    [* @@ = -1; *]
                                ):jmp

                                [*
                                    if( @jmp >= 0 )
                                    {
                                        pcb->prog->program[ @jmp ].param =
                                            pcb->prog->program_cnt;
                                        pcb->prog->program[ @jpc ].param =
                                            @jmp + 1;
                                    }
                                    else
                                        pcb->prog->program[ @jpc ].param =
                                            pcb->prog->program_cnt;
                                *]

                        |       "while"

                                (
                                    [*
                                        @@ = pcb->prog->program_cnt;
                                    *]
                                ):begin

                                    '(' expression ')'

                                (
                                    [*
                                        @@ = xpl_emit( pcb->prog, XPL_JPC, 0 );
                                    *]
                                ):jpc

                                    statement

                                [*
                                    @@ = xpl_emit( pcb->prog, XPL_JMP, @begin );
                                    pcb->prog->program[ @jpc ].param =
                                        pcb->prog->program_cnt;
                                *]

                        |       '{' statement* '}'

                        |       expression ';'
                                [*
                                    xpl_emit( pcb->prog, XPL_DRP, 0 );
                                *]

                        |       ';'
                        ;

    expression          ->      variable "=" expression
                                [*
                                    xpl_emit( pcb->prog, XPL_DUP, 0 );
                                    xpl_emit( pcb->prog, XPL_STO, @variable );
                                *]

                        |       expression "==" expression
                                [*
                                    xpl_emit( pcb->prog, XPL_EQU, 0 );
                                *]

                        |       expression "!=" expression

                                [*
                                    xpl_emit( pcb->prog, XPL_NEQ, 0 );
                                *]

                        |       expression '<' expression

                                [*
                                    xpl_emit( pcb->prog, XPL_LOT, 0 );
                                *]

                        |       expression '>' expression

                                [*
                                    xpl_emit( pcb->prog, XPL_GRT, 0 );
                                *]

                        |       expression "<=" expression

                                [*
                                    xpl_emit( pcb->prog, XPL_LEQ, 0 );
                                *]

                        |       expression ">=" expression

                                [*
                                    xpl_emit( pcb->prog, XPL_GEQ, 0 );
                                *]

                        |       expression '+' expression

                                [*
                                    xpl_emit( pcb->prog, XPL_ADD, 0 );
                                *]

                        |       expression '-' expression

                                [*
                                    xpl_emit( pcb->prog, XPL_SUB, 0 );
                                *]

                        |       expression '*' expression

                                [*
                                    xpl_emit( pcb->prog, XPL_MUL, 0 );
                                *]

                        |       expression '/' expression

                                [*
                                    xpl_emit( pcb->prog, XPL_DIV, 0 );
                                *]

                        |       '-' expression

                                [*
                                    xpl_emit( pcb->prog, XPL_LIT,
                                        xpl_get_literal( pcb->prog,
                                            xpl_value_create_integer( -1 ) ) );
                                    xpl_emit( pcb->prog, XPL_MUL, 0 );
                                *]

                                #precedence '*'

                        |       '(' expression ')'

                        |       real

                                [*
                                    xpl_emit( pcb->prog, XPL_LIT,
                                          xpl_get_literal( pcb->prog,
                                              xpl_value_create_float( @real ) ) );
                                  *]

                        |         @string

                                  [*
                                      /*
                                          Remove the quotation
                                          marks from the string
                                      */
                                      @string[ strlen( @string ) - 1 ] = 0;

                                      /*
                                          Generate code
                                      */
                                      xpl_emit( pcb->prog, XPL_LIT,
                                          xpl_get_literal( pcb->prog,
                                              xpl_value_create_string(
                                                  @string + 1, 1 ) ) );
                                  *]

                        |         variable

                                  [*
                                      xpl_emit( pcb->prog, XPL_LOD,
                                          @variable );
                                  *]

                        |         function '(' parameter_list? ')'

                                  [*
                                      /*
                                          Semantic checks if the function
                                          parameters are in a valid count.
                                      */
                                      if( xpl_check_function_parameters(
                                              @function, @parameter_list,
                                                  pcb->line ) )
                                          pcb->error_count++;

                                      /* We first push the number of parameters */
                                      xpl_emit( pcb->prog, XPL_LIT,
                                          xpl_get_literal( pcb->prog,
                                              xpl_value_create_integer(
                                                  @parameter_list ) ) );

                                      /* Then we call up the function */
                                      xpl_emit( pcb->prog, XPL_CAL, @function );
                                  *]
                        ;

parameter_list<int> ->            parameter_list ',' expression
                                  [* @@ = @parameter_list + @expression; *]

                        |         expression

                                  [* @@ = 1; *]
                        ;

variable
function <int>    ->            @identifier
```

```
                            [*
                                if( ( @@ = xpl_get_function( @identifier ) )
                                        >= 0 )
                                    @!symbol:function;
                                else
                                    @@ = xpl_get_variable(
                                            pcb->prog, @identifier );
                            *]
                    ;

/* Parser Control Block */

#pcb
[*
xpl_program*    prog;
FILE*           input;
*];

/* Prologue & Epilogue */

#prologue
[*
#include "xpl.h"

extern xpl_fn   xpl_buildin_functions[];

#define UNICC_GETINPUT  fgetc( pcb->input )
*];

#epilogue
[*


int xpl_compile( xpl_program* prog, FILE* input )
{
    @@prefix_pcb    pcb;
    memset( &pcb, 0, sizeof( @@prefix_pcb ) );

    pcb.prog = prog;
    pcb.input = input;
    pcb.eof = EOF;

    @@prefix_parse( &pcb );

    return pcb.error_count;
}

*];
```

8.3. xpl.h

```
#ifndef XPL_H
#define XPL_H

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <locale.h>

/* Virtual machine values */
typedef enum
{
    XPL_NULLVAL,                     /* Nothing/Undefined */
    XPL_INTEGERVAL,                  /* Integer type */
    XPL_FLOATVAL,                    /* Float type */
    XPL_STRINGVAL                    /* String type */
} xpl_datatype;

typedef struct
{
    xpl_datatype    type;            /* Value type */

    union
    {
        int         i;
        float       f;
        char*       s;
    } value;                         /* Value storage union */

    char*           strval;          /* Temporary string value
                                         representation pointer */
} xpl_value;

/* Functions */
typedef struct
{
    char*           name;            /* Function name */
    int             min;             /* Minimal parameter count */
    int             max;             /* Maximal parameter count */
    xpl_value*      (*fn)( int, xpl_value** ); /* Function callback pointer */
} xpl_fn;

/* Virtual machine opcodes */
typedef enum
{
    XPL_NOP,                         /* No operation */

    XPL_CAL,                         /* Call function */
    XPL_LIT,                         /* Load literal value */
    XPL_LOD,                         /* Load value from variable */
    XPL_STO,                         /* Store value into variable */
    XPL_DUP,                         /* Duplicate stack item */
    XPL_DRP,                         /* Drop stack item */
    XPL_JMP,                         /* Jump to address */
    XPL_JPC,                         /* Jump to address if false */

    XPL_EQU,                         /* Check for equal */
    XPL_NEQ,                         /* Check for not-equal */
    XPL_LOT,                         /* Check for lower-than */
    XPL_LEQ,                         /* Check for lower-equal */
    XPL_GRT,                            /* Check for greater-than */
    XPL_GEQ,                            /* Check for greater-equal */

    XPL_ADD,                            /* Add or join two values */
    XPL_SUB,                            /* Substract two values */
    XPL_MUL,                            /* Multiply two values */
    XPL_DIV                             /* Divide two values */
} xpl_op;

/* Command description */
typedef struct
{
    xpl_op      op;                     /* Opcode */
    int         param;                  /* Parameter */
} xpl_cmd;

/* Program structure */
typedef struct
{
    xpl_cmd*    program;                /* Virtual machine program */
    int         program_cnt;            /* Number of elements in program */

    xpl_value** literals;               /* Array of literal objects */
    int         literals_cnt;           /* Number of elements in literals */

    char**      variables;              /* The variable symbol table. The
                                            index of the variable name
                                            represents the address in the
                                            variables-member of the
                                            runtime data structure.
                                        */
    int         variables_cnt;          /* Number of elements
                                            in variables */

} xpl_program;

/* Runtime structure */
typedef struct
{
    xpl_value** variables;              /* Array of objects representing
                                            the variable values. The
                                            number of objects is in
                                            the corresponding xpl_program
                                            structure in the member
                                            variables_cnt
                                        */

    xpl_value** stack;                  /* Array representing the value
                                            stack */
    int         stack_cnt;              /* Defines the top-of-stack, and
                                            the numbers of elements
                                            resisting.
                                        */

    xpl_cmd*    ip;                     /* Instruction Pointer to the
                                            currently executed code
                                            address.
                                        */
} xpl_runtime;

/* Some general defines */
#define XPL_MALLOCSTEP                  256
```

```
/* Import function prototypes */
#include "xpl.proto.h"

#endif
```

8.4. xpl.proto.h

```
#ifndef XPL_PROTO_H
#define XPL_PROTO_H

/* xpl.debug.c */
void xpl_dump( xpl_program* prog, xpl_runtime* rt );

/* xpl.functions.c */
int xpl_get_function( char* name );
xpl_value* XPL_print( int argc, xpl_value** argv );
xpl_value* XPL_prompt( int argc, xpl_value** argv );
xpl_value* XPL_exit( int argc, xpl_value** argv );
xpl_value* XPL_integer( int argc, xpl_value** argv );
xpl_value* XPL_float( int argc, xpl_value** argv );
xpl_value* XPL_string( int argc, xpl_value** argv );

/* xpl.parser.c */
int xpl_compile( xpl_program* prog, FILE* input );

/* xpl.program.c */
int xpl_get_variable( xpl_program* prog, char* name );
int xpl_get_literal( xpl_program* prog, xpl_value* val );
int xpl_emit( xpl_program* prog, xpl_op op, int param );
void xpl_reset( xpl_program* prog );

/* xpl.run.c */
void xpl_run( xpl_program* prog );

/* xpl.util.c */
char* xpl_malloc( char* oldptr, int size );
char* xpl_strdup( char* str );
char* xpl_free( char* ptr );

/* xpl.value.c */
xpl_value* xpl_value_create( void );
xpl_value* xpl_value_create_integer( int i );
xpl_value* xpl_value_create_float( float f );
xpl_value* xpl_value_create_string( char* s, short duplicate );
void xpl_value_free( xpl_value* val );
void xpl_value_reset( xpl_value* val );
xpl_value* xpl_value_dup( xpl_value* val );
void xpl_value_set_integer( xpl_value* val, int i );
void xpl_value_set_float( xpl_value* val, float f );
void xpl_value_set_string( xpl_value* val, char* s );
int xpl_value_get_integer( xpl_value* val );
float xpl_value_get_float( xpl_value* val );
char* xpl_value_get_string( xpl_value* val );

#endif
```

8.5. xpl.debug.c

```
#include "xpl.h"

static char opcodes[][3+1] =
{
    "NOP", "CAL", "LIT", "LOD", "STO", "DUP", "DRP", "JMP", "JPC",
    "EQU", "NEQ", "LOT", "LEQ", "GRT", "GEQ",
    "ADD", "SUB", "MUL", "DIV"
};

extern xpl_fn   xpl_buildin_functions[];

void xpl_dump( xpl_program* prog, xpl_runtime* rt )
{
    int         i;
    xpl_cmd*    cmd;

    for( i = 0; i < prog->program_cnt; i++ )
    {
        cmd = &( prog->program[i] );

        fprintf( stderr, "%s%03d: %s ",
                    ( rt && rt->ip == cmd ) ? ">" : " ",
                        i, opcodes[ cmd->op ] );

        if( cmd->op == XPL_JMP || cmd->op == XPL_JPC )
            fprintf( stderr, "%03d", cmd->param );
        else if( cmd->op == XPL_LIT )
            fprintf( stderr, "%d (%s%s%s)", cmd->param,
                        prog->literals[ cmd->param ]->type == XPL_STRINGVAL ?
                            "\"" : "",
                        xpl_value_get_string( prog->literals[ cmd->param ] ),
                        prog->literals[ cmd->param ]->type == XPL_STRINGVAL ?
                            "\"" : "" );
        else if( cmd->op == XPL_LOD || cmd->op == XPL_STO )
            fprintf( stderr, "%d => %s", cmd->param,
                        prog->variables[ cmd->param ] );
        else if( cmd->op == XPL_CAL )
            fprintf( stderr, "%d => %s()", cmd->param,
                        xpl_buildin_functions[ cmd->param ].name );

        fprintf( stderr, "\n" );
    }
}
```

8.6. xpl.functions.c

```
#include "xpl.h"

xpl_fn  xpl_buildin_functions[] =
{
    {   "exit",     -1,     1,      XPL_exit    },
    {   "print",    1,      -1,     XPL_print   },
    {   "prompt",   -1,     1,      XPL_prompt  },
    {   "integer",  1,      1,      XPL_integer },
    {   "float",    1,      1,      XPL_float   },
    {   "string",   1,      1,      XPL_string  },
};

int xpl_get_function( char* name )
{
    int     i;

    /* Try to find function */
    for( i = 0; i < sizeof( xpl_buildin_functions ) / sizeof( xpl_fn ); i++ )
        if( strcmp( xpl_buildin_functions[ i ].name, name ) == 0 )
            return i;

    return -1;
}

int xpl_check_function_parameters( int function, int parameter_count, int line )
{
    if( xpl_buildin_functions[ function ].min > -1 )
    {
        if( parameter_count < xpl_buildin_functions[ function ].min )
        {
            fprintf( stderr,
                "line %d: Too less parameters in call to %s(), %d parameters "
                "required at minimum",
                    line, xpl_buildin_functions[ function ].name,
                        xpl_buildin_functions[ function ].min );
            return 1;
        }
    }
    else if( xpl_buildin_functions[ function ].max > -1 )
    {
        if( parameter_count > xpl_buildin_functions[ function ].max )
        {
            fprintf( stderr,
                "line %d: Too many parameters in call to %s(), %d parameters "
                "allowed at maximum",
                    line, xpl_buildin_functions[ function ].name,
                        xpl_buildin_functions[ function ].max );
            return 1;
        }
    }

    return 0;
}

/* Build-in functions follow */

xpl_value* XPL_print( int argc, xpl_value** argv )
{
    int     i;
    for( i = 0; i < argc; i++ )
        printf( "%s\n", xpl_value_get_string( argv[ i ] ) );

    return (xpl_value*)NULL;
}

xpl_value* XPL_prompt( int argc, xpl_value** argv )
{
    char    buf [ 256 + 1 ];

    if( argc > 0 )
        printf( "%s: ", xpl_value_get_string( argv[ 0 ] ) );

    if( fgets( buf, sizeof( buf ), stdin ) )
    {
        buf[ strlen( buf ) - 1 ] = '\0';
        return xpl_value_create_string( buf, 1 );
    }

    return xpl_value_create_string( "", 1 );
}

xpl_value* XPL_exit( int argc, xpl_value** argv )
{
    int     rc      = 0;

    if( argc > 0 )
        rc = xpl_value_get_integer( argv[ 0 ] );

    exit( rc );
    return (xpl_value*)NULL;
}

xpl_value* XPL_integer( int argc, xpl_value** argv )
{
    return xpl_value_create_integer( xpl_value_get_integer( *argv ) );
}

xpl_value* XPL_float( int argc, xpl_value** argv )
{
    return xpl_value_create_float( xpl_value_get_float( *argv ) );
}

xpl_value* XPL_string( int argc, xpl_value** argv )
{
    return xpl_value_create_string( xpl_value_get_string( *argv ), 1 );
}
```

8.7. xpl.main.c

```
#include "xpl.h"

int main( int argc, char** argv )
{
    xpl_program     program;
    FILE*           src;

    /* Initialize program structure to all zero */
    memset( &program, 0, sizeof( xpl_program ) );

    /* Open input file, if provided. */
    if( argc > 1 )
    {
        if( !( src = fopen( argv[1], "rb" ) ) )
        {
            fprintf( stderr, "Can't open file '%s'\n", argv[1] );
            return 1;
        }
    }
    else
    {
        printf( "Usage: %s FILENAME\n", *argv );
        return 1;
    }

    /* Call the compiler */
    if( xpl_compile( &program, src ) > 0 )
        return 1;

    /* Dump program before execution */
    xpl_dump( &program, (xpl_runtime*)NULL );

    /* Run program */
    xpl_run( &program );

    /* Clean-up used memory */
    xpl_reset( &program );

    if( src != stdin )
        fclose( src );

    return 0;
}
```

8.8. xpl.program.c

```
#include "xpl.h"

/* Program structure handling */
int xpl_get_variable( xpl_program* prog, char* name )
{
    int     i;

    /* A function name with the same identifier may not exist! */
    if( xpl_get_function( name ) > -1 )
        return -1;

    /* Try to find variable index */
    for( i = 0; i < prog->variables_cnt; i++ )
        if( strcmp( prog->variables[ i ], name ) == 0 )
            return i;

    /* Else, eventually allocate memory for new variables */
    if( ( i % XPL_MALLOCSTEP ) == 0 )
    {
        prog->variables = (char**)xpl_malloc(
                            (char*)prog->variables, ( i + XPL_MALLOCSTEP )
                                                        * sizeof( char** ) );
    }

    /* Introduce new variable */
    prog->variables[ prog->variables_cnt ] = xpl_strdup( name );

    return prog->variables_cnt++;;
}

int xpl_get_literal( xpl_program* prog, xpl_value* val )
{
    /* Else, eventually allocate memory for new variables */
    if( ( prog->literals_cnt % XPL_MALLOCSTEP ) == 0 )
    {
        prog->literals = (xpl_value**)xpl_malloc(
                                (char*)prog->literals,
                                    ( prog->literals_cnt + XPL_MALLOCSTEP )
                                        * sizeof( xpl_value* ) );
    }

    prog->literals[ prog->literals_cnt ] = val;
    return prog->literals_cnt++;
}

int xpl_emit( xpl_program* prog, xpl_op op, int param )
{
    if( ( prog->program_cnt % XPL_MALLOCSTEP ) == 0 )
    {
        prog->program = (xpl_cmd*)xpl_malloc(
                            (char*)prog->program,
                                ( prog->program_cnt + XPL_MALLOCSTEP )
                                    * sizeof( xpl_cmd ) );
    }

    prog->program[ prog->program_cnt ].op = op;
    prog->program[ prog->program_cnt ].param = param;
    return prog->program_cnt++;
}
```

```
void xpl_reset( xpl_program* prog )
{
    int     i;

    /* Variables */
    for( i = 0; i < prog->variables_cnt; i++ )
        xpl_free( prog->variables[ i ] );

    xpl_free( (char*)prog->variables );

    /* Literals */
    for( i = 0; i < prog->literals_cnt; i++ )
        xpl_value_free( prog->literals[ i ] );

    xpl_free( (char*)prog->literals );

    /* Program */
    xpl_free( (char*)prog->program );

    memset( prog, 0, sizeof( xpl_program ) );
}
```

8.9. xpl.run.c

```
#include "xpl.h"

extern xpl_fn   xpl_buildin_functions[];

static int xpl_push( xpl_runtime* rt, xpl_value* val )
{
    if( ( rt->stack_cnt % XPL_MALLOCSTEP ) == 0 )
    {
        rt->stack = (xpl_value**)xpl_malloc(
                            (char*)rt->stack,
                                ( rt->stack_cnt + XPL_MALLOCSTEP )
                                    * sizeof( xpl_value ) );
    }

    rt->stack[ rt->stack_cnt ] = val;
    return rt->stack_cnt++;
}

static xpl_value* xpl_pop( xpl_runtime* rt )
{
    if( !rt->stack_cnt )
        return (xpl_value*)NULL;

    return rt->stack[ --rt->stack_cnt ];
}

static void xpl_stack( xpl_runtime* rt )
{
    int     i;

    for( i = 0; i < rt->stack_cnt; i++ )
        fprintf( stderr, "% 3d: %s\n", i,
                    xpl_value_get_string( rt->stack[ i ] ) );

    if( !i )
        fprintf( stderr, "--- Stack is empty ---\n" );
}

void xpl_run( xpl_program* prog )
{
    xpl_runtime rt;
    xpl_value*  val;
    int         i;

    /* Initialize runtime */
    memset( &rt, 0, sizeof( xpl_runtime ) );
    rt.variables = (xpl_value**)xpl_malloc( (char*)NULL,
                        prog->variables_cnt * sizeof( xpl_value ) );

    rt.ip = prog->program;

    /* Program execution loop */
    while( rt.ip < prog->program + prog->program_cnt )
    {
        /*
        fprintf( stderr, "IP: %p\n", rt.ip );
        xpl_dump( prog, &rt );
        xpl_stack( &rt );
        */
```

```
switch( rt.ip->op )
{
    case XPL_NOP:
        /* No nothing */
        break;

    case XPL_CAL:
        /* Calling build-in functions */
        {
            int     param_cnt;

            /* Last stack item contains the number of parameters */
            val = xpl_pop( &rt );
            param_cnt = xpl_value_get_integer( val );
            xpl_value_free( val );

            /* Call the function */
            val = (*(xpl_buildin_functions[ rt.ip->param ].fn))
                    ( param_cnt, rt.stack + rt.stack_cnt - param_cnt );

            /* If no value is returned, create a default value */
            if( !val )
                val = xpl_value_create_integer( 0 );

            /* Discard the parameters from stack */
            while( param_cnt > 0 )
            {
                xpl_value_free( xpl_pop( &rt ) );
                param_cnt--;
            }

            /* Push the return value */
            xpl_push( &rt, val );
        }
        break;

    case XPL_LIT:
        /* Load literal and push duplicate */
        xpl_push( &rt, xpl_value_dup(
            prog->literals[ rt.ip->param ] ) );
        break;

    case XPL_LOD:
        /* Load value from variable and push duplicate */
        xpl_push( &rt, xpl_value_dup(
            rt.variables[ rt.ip->param ] ) );
        break;

    case XPL_STO:
        /* Store value to variable */
        if( rt.variables[ rt.ip->param ] )
            xpl_value_free( rt.variables[ rt.ip->param ] );

        rt.variables[ rt.ip->param ] = xpl_pop( &rt );
        break;

    case XPL_DUP:
        /* Duplicate stack item */
        val = xpl_pop( &rt );
        xpl_push( &rt, val );
        xpl_push( &rt, xpl_value_dup( val ) );
        break;
```

```
case XPL_DRP:
    /* Drop stack item */
    xpl_value_free( xpl_pop( &rt ) );
    break;

case XPL_JMP:
    /* Jump to address */
    rt.ip = prog->program + rt.ip->param;
    continue;

case XPL_JPC:
    /* Jump to address only if stacked value is nonzero */
    if( !xpl_value_get_integer( ( val = xpl_pop( &rt ) ) ) )
    {
        xpl_value_free( val );
        rt.ip = prog->program + rt.ip->param;
        continue;
    }

    xpl_value_free( val );
    break;

default:
    {
        xpl_datatype    prefer;
        xpl_value*      op  [ 2 ];

        /* Pop operands off the stack */
        op[1] = xpl_pop( &rt );
        op[0] = xpl_pop( &rt );

        /*
         * Get best matching type for operation from both operands
         */
        if( op[0]->type == XPL_STRINGVAL ||
                op[1]->type == XPL_STRINGVAL )
            prefer = XPL_STRINGVAL;
        else if( op[0]->type == XPL_FLOATVAL ||
                    op[1]->type == XPL_FLOATVAL )
            prefer = XPL_FLOATVAL;
        else
            prefer = XPL_INTEGERVAL;

        switch( rt.ip->op )
        {
            case XPL_ADD:
                /* Addition, or with Strings, concatenation */
                if( prefer == XPL_STRINGVAL )
                {
                    char*   str;

                    str = xpl_malloc( (char*)NULL,
                        ( strlen( xpl_value_get_string( op[0] ) )
                        + strlen( xpl_value_get_string( op[1] ) )
                        + 1 ) * sizeof( char ) );


                    sprintf( str, "%s%s",
                        xpl_value_get_string( op[0] ),
                        xpl_value_get_string( op[1] ) );
                    val = xpl_value_create_string( str, 0 );
                }
                else if( prefer == XPL_FLOATVAL )
                {
                    val = xpl_value_create_float(
                            xpl_value_get_float( op[0] ) +
                                xpl_value_get_float( op[1] ) );
                }
                else
                {
                    val = xpl_value_create_integer(
                            xpl_value_get_integer( op[0] ) +
                                xpl_value_get_integer( op[1] ) );
                }
                break;

            case XPL_SUB:
                /* Substraction */
                if( prefer == XPL_FLOATVAL )
                {
                    val = xpl_value_create_float(
                            xpl_value_get_float( op[0] ) -
                                xpl_value_get_float( op[1] ) );
                }
                else
                {
                    val = xpl_value_create_integer(
                            xpl_value_get_integer( op[0] ) -
                                xpl_value_get_integer( op[1] ) );
                }
                break;

            case XPL_MUL:
                /* Multiplication */
                if( prefer == XPL_FLOATVAL )
                {
                    val = xpl_value_create_float(
                            xpl_value_get_float( op[0] ) *
                                xpl_value_get_float( op[1] ) );
                }
                else
                {
                    val = xpl_value_create_integer(
                            xpl_value_get_integer( op[0] ) *
                                xpl_value_get_integer( op[1] ) );
                }
                break;

            case XPL_DIV:
                /* Division */
                if( prefer == XPL_FLOATVAL )
                {
                    val = xpl_value_create_float(
                            xpl_value_get_float( op[0] ) /
                                xpl_value_get_float( op[1] ) );
                }
                else
                {
                    val = xpl_value_create_integer(
                            xpl_value_get_integer( op[0] ) /
                                xpl_value_get_integer( op[1] ) );
                }
                break;

            default:
                {
                    float   res;

                    /*
                     * Compare by subtracting the left operand
                     * from the right operand, or with the string
                     * comparison function strcmp, resulting in:
                     *
                     * res == 0                  equal
                     * res != 0                  not equal
                     * res < 0                   lower than
                     * res <= 0                  lower-equal
                     * res > 0                   greater-than
                     * res >=0                   greater-equal
                     */
                    if( prefer == XPL_STRINGVAL )
                    {
                        res = (float)strcmp(
                                xpl_value_get_string( op[0] ),
                                xpl_value_get_string( op[1] ) );
                    }
                    else if( prefer == XPL_FLOATVAL )
                    {
                        res = xpl_value_get_float( op[0] )
                                - xpl_value_get_float( op[1] );
                    }
                    else
                    {
                        res = (float)xpl_value_get_integer( op[0] )
                                - xpl_value_get_integer( op[1] );
                    }

                    /* Switch comparison */
                    switch( rt.ip->op )
                    {
                        case XPL_EQU:
                            val = xpl_value_create_integer(
                                    !res ? 1 : 0 );
                            break;
                        case XPL_NEQ:
                            val = xpl_value_create_integer(
                                    res ? 1 : 0 );
                            break;
                        case XPL_LOT:
                            val = xpl_value_create_integer(
                                    res < 0 ? 1 : 0 );
                            break;
                        case XPL_LEQ:
                            val = xpl_value_create_integer(
                                    res <= 0 ? 1 : 0 );
                            break;
                        case XPL_GRT:
                            val = xpl_value_create_integer(
                                    res > 0 ? 1 : 0 );
                            break;
                        case XPL_GEQ:
                            val = xpl_value_create_integer(
                                    res >= 0 ? 1 : 0 );
                            break;
```

```
                                }
                            }
                        }

                        /* Free the operands */
                        xpl_value_free( op[0] );
                        xpl_value_free( op[1] );

                        /* Push the operation or comparison result */
                        xpl_push( &rt, val );
                }
                break;
        }

        /* Increment instruction pointer */
        rt.ip++;
    }

    /*
     * Clear stack
     * (if code was clearly generated, this would not be required)
     */
    for( i = 0; i < rt.stack_cnt; i++ )
        xpl_value_free( rt.stack[ i ] );

    xpl_free( (char*)rt.stack );

    /* Clear variables */
    for( i = 0; i < prog->variables_cnt; i++ )
        xpl_value_free( rt.variables[ i ] );

    xpl_free( (char*)rt.variables );
}
```

8.10. xpl.util.c

```
#include "xpl.h"

/* Function for memory allocation */
char* xpl_malloc( char* oldptr, int size )
{
    char*   retptr;

    if( oldptr )
        retptr = (char*)realloc( oldptr, size );
    else
        retptr = (char*)malloc( size );

    if( !retptr )
    {
        fprintf( stderr, "%s, %d: Fatal error, XPL ran out of memory\n",
                    __FILE__, __LINE__ );
        exit( 1 );
    }

    if( !oldptr )
        memset( retptr, 0, size );

    return retptr;
}

char* xpl_strdup( char* str )
{
    char*   nstr;

    nstr = xpl_malloc( (char*)NULL, ( strlen( str ) + 1 ) * sizeof( char ) );
    strcpy( nstr, str );

    return nstr;
}

char* xpl_free( char* ptr )
{
    if( !ptr )
        return (char*)NULL;

    free( ptr );
    return (char*)NULL;
}
```

8.11. xpl.value.c

```
#include "xpl.h"

/* Value Objects */

xpl_value* xpl_value_create( void )
{
    return (xpl_value*)xpl_malloc( (char*)NULL, sizeof( xpl_value ) );
}

xpl_value* xpl_value_create_integer( int i )
{
    xpl_value*  val;

    val = xpl_value_create();
    xpl_value_set_integer( val, i );

    return val;
}

xpl_value* xpl_value_create_float( float f )
{
    xpl_value*  val;

    val = xpl_value_create();
    xpl_value_set_float( val, f );

    return val;
}

xpl_value* xpl_value_create_string( char* s, short duplicate )
{
    xpl_value*  val;

    val = xpl_value_create();
    xpl_value_set_string( val, duplicate ? xpl_strdup( s ) : s );

    return val;
}

void xpl_value_free( xpl_value* val )
{
    if( !val )
        return;

    xpl_value_reset( val );
    free( val );
}

void xpl_value_reset( xpl_value* val )
{
    if( val->type == XPL_STRINGVAL && val->value.s )
        free( val->value.s );

    val->strval = xpl_free( val->strval );

    memset( val, 0, sizeof( xpl_value ) );
    val->type = XPL_NULLVAL;
}
```

```
xpl_value* xpl_value_dup( xpl_value* val )
{
    xpl_value*  dup;

    dup = xpl_value_create();

    if( !val )
        return dup;

    memcpy( dup, val, sizeof( xpl_value ) );

    dup->strval = (char*)NULL;

    if( dup->type == XPL_STRINGVAL )
        dup->value.s = xpl_strdup( dup->value.s );

    return dup;
}

void xpl_value_set_integer( xpl_value* val, int i )
{
    xpl_value_reset( val );
    val->type = XPL_INTEGERVAL;
    val->value.i = i;
}

void xpl_value_set_float( xpl_value* val, float f )
{
    xpl_value_reset( val );
    val->type = XPL_FLOATVAL;
    val->value.f = f;
}

void xpl_value_set_string( xpl_value* val, char* s )
{
    xpl_value_reset( val );
    val->type = XPL_STRINGVAL;
    val->value.s = s;
}

int xpl_value_get_integer( xpl_value* val )
{
    switch( val->type )
    {
        case XPL_INTEGERVAL:
            return val->value.i;
        case XPL_FLOATVAL:
            return (int)val->value.f;
        case XPL_STRINGVAL:
            return atoi( val->value.s );

        default:
            break;
    }

    return 0;
}

float xpl_value_get_float( xpl_value* val )
{
    switch( val->type )
    {
        case XPL_INTEGERVAL:
            return (float)val->value.i;
        case XPL_FLOATVAL:
            return val->value.f;
        case XPL_STRINGVAL:
            return (float)atof( val->value.s );

        default:
            break;
    }

    return 0.0;
}

char* xpl_value_get_string( xpl_value* val )
{
    char    buf     [ 128 + 1 ];
    char*   p;

    val->strval = xpl_free( val->strval );

    switch( val->type )
    {
        case XPL_INTEGERVAL:
            sprintf( buf, "%d", val->value.i );
            val->strval = xpl_strdup( buf );
            return val->strval;
        case XPL_FLOATVAL:
            sprintf( buf, "%f", val->value.f );

            /* Remove trailing zeros to make values look nicer */
            for( p = buf + strlen( buf ) - 1; p > buf; p-- )
            {
                if( *p == '.' )
                {
                    *p = '\0';
                    break;
                }
                else if( *p != '0' )
                    break;

                *p = '\0';
            }

            val->strval = xpl_strdup( buf );
            return val->strval;
        case XPL_STRINGVAL:
            return val->value.s;

        default:
            break;
    }

    return "";
}
```

9. Appendix II: unicc.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
UniCC LALR(1) Parser Generator
Copyright (C) 2006-2016 by Phorward Software Technologies, Jan Max Meyer
http://unicc.phorward-software.com ++ unicc<at>phorward<dash>software<dot>com
All rights reserved. See LICENSE for more information.

File:           unicc.dtd
Author:         Jan Max Meyer
Usage:          Document Type Definitions for Parser Definition Files
-->

<!ELEMENT begin-of-match EMPTY >

<!ELEMENT character-class ( range+ ) >
<!ATTLIST character-class count NMTOKEN #REQUIRED >

<!ELEMENT code ( begin-of-match | command | raw | return-value | variable )* >
<!ATTLIST code defined-at NMTOKEN #IMPLIED >

<!ELEMENT command EMPTY >
<!ATTLIST command action NMTOKEN #REQUIRED >
<!ATTLIST command symbol NMTOKEN #REQUIRED >

<!ELEMENT copyright ( #PCDATA ) >

<!ELEMENT description ( #PCDATA ) >

<!ELEMENT dfa ( state+ ) >

<!ELEMENT epilogue ( #PCDATA ) >

<!ELEMENT goto EMPTY >
<!ATTLIST goto by-production NMTOKEN #IMPLIED >
<!ATTLIST goto symbol-id NMTOKEN #REQUIRED >
<!ATTLIST goto to-state NMTOKEN #IMPLIED >

<!ELEMENT left-hand-side EMPTY >
<!ATTLIST left-hand-side offset ( 0 | 1 ) #REQUIRED >
<!ATTLIST left-hand-side symbol-id NMTOKEN #REQUIRED >

<!ELEMENT lexer ( state+ ) >
<!ATTLIST lexer id NMTOKEN #REQUIRED >

<!ELEMENT lexers ( lexer+ ) >

<!ELEMENT parser ( version, copyright, description, symbols, productions,
states, lexers, value-types, prologue, epilogue, pcb, source ) >
<!ATTLIST parser basename NMTOKEN #REQUIRED >
<!ATTLIST parser char-max NMTOKEN #REQUIRED >
<!ATTLIST parser char-min NMTOKEN #REQUIRED >
<!ATTLIST parser mode NMTOKEN #REQUIRED >
<!ATTLIST parser name NMTOKEN #REQUIRED >
<!ATTLIST parser prefix NMTOKEN #REQUIRED >
<!ATTLIST parser source NMTOKEN #REQUIRED >
<!ATTLIST parser target-language NMTOKEN #REQUIRED >
<!ATTLIST parser unicc-version NMTOKEN #REQUIRED >

<!ELEMENT pcb ( #PCDATA ) >
```

```
<!ELEMENT production ( code | left-hand-side | right-hand-side |
semantic-right-hand-side )* >
<!ATTLIST production defined-at NMTOKEN #IMPLIED >
<!ATTLIST production id NMTOKEN #REQUIRED >
<!ATTLIST production length NMTOKEN #REQUIRED >

<!ELEMENT productions ( production+ ) >

<!ELEMENT prologue ( #PCDATA ) >

<!ELEMENT range EMPTY >
<!ATTLIST range from NMTOKEN #REQUIRED >
<!ATTLIST range to NMTOKEN #REQUIRED >

<!ELEMENT raw ( #PCDATA ) >

<!ELEMENT regex ( #PCDATA ) >

<!ELEMENT return-value EMPTY >
<!ATTLIST return-value value-type CDATA #REQUIRED >
<!ATTLIST return-value value-type-id NMTOKEN #REQUIRED >

<!ELEMENT right-hand-side EMPTY >
<!ATTLIST right-hand-side named CDATA #IMPLIED >
<!ATTLIST right-hand-side offset NMTOKEN #REQUIRED >
<!ATTLIST right-hand-side symbol-id NMTOKEN #REQUIRED >

<!ELEMENT semantic-right-hand-side EMPTY >
<!ATTLIST semantic-right-hand-side named NMTOKEN #IMPLIED >
<!ATTLIST semantic-right-hand-side offset NMTOKEN #REQUIRED >
<!ATTLIST semantic-right-hand-side symbol-id NMTOKEN #REQUIRED >

<!ELEMENT shift EMPTY >
<!ATTLIST shift symbol-id NMTOKEN #REQUIRED >
<!ATTLIST shift to-state NMTOKEN #REQUIRED >

<!ELEMENT shift-reduce EMPTY >
<!ATTLIST shift-reduce by-production NMTOKEN #REQUIRED >
<!ATTLIST shift-reduce symbol-id NMTOKEN #REQUIRED >

<!ELEMENT source ( #PCDATA ) >

<!ELEMENT state ( goto | shift | shift-reduce | transition )* >
<!ATTLIST state accept NMTOKEN #IMPLIED >
<!ATTLIST state default-production NMTOKEN #IMPLIED >
<!ATTLIST state default-transition NMTOKEN #IMPLIED >
<!ATTLIST state derived-from-state NMTOKEN #IMPLIED >
<!ATTLIST state id NMTOKEN #REQUIRED >
<!ATTLIST state lexer NMTOKEN #IMPLIED >

<!ELEMENT states ( state+ ) >

<!ELEMENT symbol ( character-class | code | dfa | regex )* >
<!ATTLIST symbol defined-at NMTOKEN #IMPLIED >
<!ATTLIST symbol derived-from NMTOKEN #IMPLIED >
<!ATTLIST symbol id NMTOKEN #REQUIRED >
<!ATTLIST symbol name CDATA #REQUIRED >
<!ATTLIST symbol terminal-type ( character-class | string | regular-expression
| system ) #IMPLIED >
<!ATTLIST symbol type ( non-terminal | terminal ) #REQUIRED >
<!ATTLIST symbol value-type CDATA #IMPLIED >
```

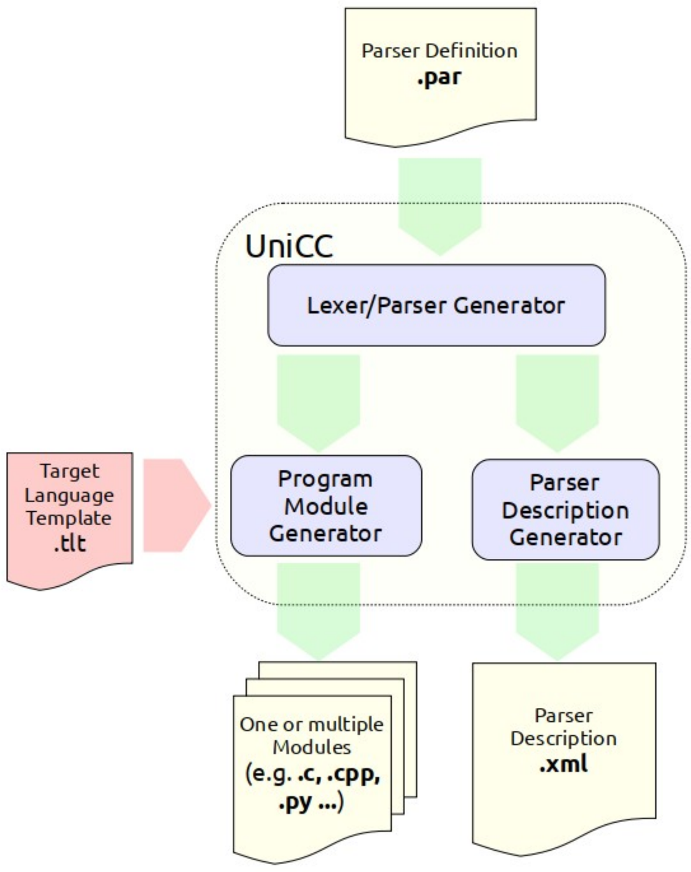
```
<!ATTLIST symbol value-type-id NMTOKEN #IMPLIED >

<!ELEMENT symbols ( symbol+ ) >

<!ELEMENT transition ( character-class ) >
<!ATTLIST transition goto NMTOKEN #REQUIRED >

<!ELEMENT value-type ( #PCDATA ) >
<!ATTLIST value-type c_name CDATA #REQUIRED >
<!ATTLIST value-type id NMTOKEN #REQUIRED >

<!ELEMENT value-types ( value-type+ ) >

<!ELEMENT variable EMPTY >
<!ATTLIST variable offset NMTOKEN #IMPLIED >
<!ATTLIST variable target ( left-hand-side | right-hand-side ) #REQUIRED >
<!ATTLIST variable value-type CDATA #IMPLIED >
<!ATTLIST variable value-type-id NMTOKEN #IMPLIED >

<!ELEMENT version ( #PCDATA ) >
```

UniCC: A universal LALR(1) Parser Generator
User Manual

UniCC is a target-language independent parser generator. It compiles an augmented grammar definition into a program source code. Because UniCC is intended to be target-language independent, it can be configured via target language templates to emit compileable and running parsers in probably any programming language.

This manual gives a detailed introduction into the software, and serves both as a reference guide and a practical information resource for using UniCC in real-world applications.

UniCC provides out-of-the box support for the programming languages **C, C++** and **Python** so far. More target languages can easily be integrated and made available. Further developments both in the parser generator and its targets is active and ongoing. Enjoy!



Phorward Software Technologies

https://phorward.info

...with best wishes from high above...