

# Creating a programming language

Christian Klauser, 1 MNb

October 2007

```
var maxBeer = 99;

function subject(n) =
  if(n > 1)
    "$n bottles"
  else if(n == 1)
    "1 bootle"
  else
    "no more bottles";

function lyrics(n) =
  if(n > 0)
    "$(subject(n)) of beer on the wall, " +
    "$(subject(n)) of beer.\n" +
    "Take one down and pass it around, " +
    "$(subject(n-1)) of beer on the wall."
  else
    "No more bottles of beer on the wall, " +
    "no more bottles of beer.\n" +
    "Go to the store and buy some more, " +
    "$(maxBeer) bottles of beer on the wall.";

coroutine song does
  for(var i = maxBeer; i >= 0; i--)
    yield lyrics(i);

function main
{
  foreach(var verse in song)
    println("${verse}\n");
}
```

Supervised by: Tobias Bäumlín, applied mathematics  
Urs Zimmermann, english

# Contents

<b>1</b>	<b>Foreword</b>	<b>2</b>
<b>2</b>	<b>Introduction</b>	<b>2</b>
<b>3</b>	<b>Language design</b>	<b>4</b>
3.1	Goals . . . . .	4
3.2	Semantics . . . . .	5
3.3	Syntax . . . . .	8
3.4	Implementation . . . . .	9
<b>4</b>	<b>Scanner</b>	<b>10</b>
4.1	Regular expressions . . . . .	10
4.2	Implementation . . . . .	11
<b>5</b>	<b>Parser</b>	<b>13</b>
5.1	Context-free grammar . . . . .	13
5.2	Ambiguities . . . . .	16
5.3	Syntax-directed translation . . . . .	19
5.4	Constructing the AST . . . . .	21
5.5	Translation scheme . . . . .	21
5.6	Parsing . . . . .	23
<b>6</b>	<b>Code generation</b>	<b>26</b>
6.1	Simple optimizations . . . . .	26
6.2	Emitting code . . . . .	28
<b>7</b>	<b>Runtime environment</b>	<b>30</b>
7.1	Virtual stack machine . . . . .	30
<b>8</b>	<b>Closing comments</b>	<b>32</b>

# 1 Foreword

Programming has always fascinated me and so did solving of meta problems; optimizing the process of finding a solution. In practice, this means that most of what I code, are function libraries.

Over time, those got more and more abstract, more general. I was looking for an efficient way to solve problems. At some point, the expressiveness of the programming languages I used, was no longer sufficient to model those abstract optimizations. And so I started playing around with the idea of creating my own programming language. I think a new language is the most abstract solution to problems like the one of the *travelling salesman*.

# 2 Introduction

Computers are machines that process instructions. In the beginning, computers were programmed in their “native” language: the codes in their instruction set. This, of course, is a tedious process, as you have to dictate every little detail of a computation.

This is why programming languages were invented. They abstract away the unnecessary details of the underlying machine and let you focus on the problem, you want to solve. So instead of loading data into the correct arithmetic registers of your CPU before issuing the add instruction, you just add two numbers. Why would you want to explicitly specify the way a computation has to be performed, when computers can figure that out by themselves?

The very first programming language is generally considered to be Plankalkül, developed by Konrad Zuse during 1943-1945. Ironically, it remained a concept until the implementation of its first compiler in 2000 ([5]). In the 1950s, the first three modern programming languages, whose descendants are still in widespread use today, were designed: *FORTRAN*, *Lisp* and *COBOL* ([4]). Although the first FORTRAN compiler already utilized optimization techniques, many people were concerned about the speed of compiled programs compared to handwritten ones. However, in some fields the time it took to develop a program was more important than the runtime, which led to the acceptance of programming languages.

The definitive revolution came in the 1980s when processor manufacturers started designing their architectures for compilers instead of humans. IBM's *Power* architecture is the result of that movement. Ironically, the “traditional” Intel architecture is much more widely used in PCs today.

## Compilers

Since computers only process instructions that are part of their instruction set, programs written in programming languages have to be translated into functionally equivalent programs in machine language prior to their execution. This process is called *compilation* and is performed by *compilers*.

To cope with this task, the translation is commonly split up into multiple steps, also referred to as phases. A compiler starts with reading the input file byte by byte, character by character.

Like our eye splits up a text into individual words, the second step is to group meaningful characters together and remove those without meaning (e.g., whitespaces). This step is called *lexical analysis* and results in a stream of *tokens*: short, categorized strings of characters.

In the next phase, the compiler determines the relationship between the tokens. It applies the *syntax* of the programming language and is therefore called *syntactical analysis*. It results in a tree structure that represents the program.

At this stage, some compilers apply additional transformations to the program to increase performance before finally generating code in the language of the target machine.

While real world compilers not always adhere to the strict separation of these steps, most of them distinguish clearly between reading (understanding) a program and generating code. Mostly because that makes it easier to adapt the compiler to multiple target machines.

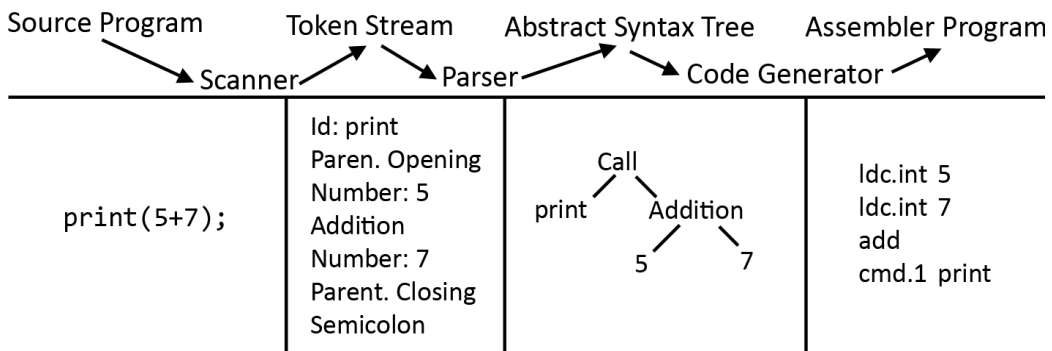


Figure 1: Process of translation

## 3 Language design

If you are thinking about creating your own programming language, you should clearly define its goals, behavior and appearance before even thinking about the actual implementation. I will answer all the questions asked in the following sections in the context of my own language, Prexonite Script.

### 3.1 Goals

The very first thing you have to do when designing a language, is to define its purpose, its mission. Why are you creating the language? Will it be specialized in a specific problem domain (e.g., requesting data from a relational database, like SQL does) or will it serve as a general purpose programming language? Will your language be used by hardcore system programmers, who want total control and speed; by enterprise application architects, who have to model complex business processes within a usually very limited time frame; by hobbyists, who want to quickly write backup scripts and simple games; or by complete programming novices, who will most likely have no concepts of “variables” and “functions”? Which aspects are important to you and/or your users? What is your language's advantage over comparable languages?

**Why?**

- I want to find my “perfect syntax”. There are many good programming languages out there but there is always something I don't like about every language I have tried so far.
- I like inventing programming languages

**Specialization?** As its name says, Prexonite Script is a scripting language and therefore highly abstracted from the underlying machine. This also means that it requires specialized libraries for anything other than reading from and writing to a console window. The Prexonite scripting engine is supposed to be hosted by custom applications and relies on interfaces they expose to script code. Also, I want my users to focus more on the problem they're trying to solve and less on the details of their tool.

**Who?** I expect users of Prexonite to be hobbyists like me. So they will want to write small scripts to automate common tasks.

### Special advantages?

- Code can be added to already compiled applications on-the-fly.
- Extendable type system
- Support for closures, nested functions and coroutines.

## 3.2 Semantics

The next step is to define your languages behaviour, its meaning. This includes but is not limited to the choice of a programming paradigm as well as the definition of the language's behaviour. The expression `2 + "thirteen"` might result in the number 2 if "thirteen" is treated as 0, the number 15 if the string is "read" and interpreted, the string "2thirteen" if 2 is converted to a string first or just an error.

### Programming paradigm

Over the time, computer scientists and compiler writers came up with many different views on "programs". I will introduce the most important ones in this section.

**Imperative programming** defines computation as a series of statements that change a program state.

**Declarative programming** is writing *what* you want to be computed instead of the actual algorithm to compute it (e.g., functional languages and relational database query languages).

**Functional programming** defines a program as the evaluation of a mathematical function (i.e., a function that does not have side effects and whose result only depends on the functions argument). In *Haskell* you would write a function that computes  $n!$  using a recursive definition.

```
factorial 1 = 1
factorial n = n * factorial (n-1)
```

While this may seem practical, typical programs usually do have side effects (like printing the results to the console window) and therefore functional programming languages provide mechanics for dealing with actions. Actions are functions that do have side effects and/or depend more than just their argument. Haskell uses the concept of *Monads*<sup>1</sup> to handle side effects.

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Monads\\_in\\_functional\\_programming](http://en.wikipedia.org/wiki/Monads_in_functional_programming)

**Flow-driven programming** is the idea of code following a straight path until it reaches a branching point, where the programs state is used to determine which path to follow from that point on.

**Event-driven programming** is to write code that waits for specific events to occur. Highly reactive systems like user interfaces or hardware drivers may use this concept. Event-driven programming can be done in a flow-driven language by writing a loop that continuously fetches an event from a queue and calls the corresponding subroutine/branches to the corresponding code path. You might include facilities for event-driven programming into your language like *TCL* does.

**Procedural programming** is a more structured way of writing imperative code by splitting it up into subroutines. In contrast to the mathematical definition given in the *Haskell* example above, procedural code has to explicitly define the algorithm used to compute the factorial.

```
int factorial(int n)
{
    int result = 1;
    for(int i = n; i != 0; i--)
        result = result * i;
    return result;
}
```

**Object oriented programming** is the idea of modelling the real world by packing data and operations on that data, the functions, into objects. To “open the door” on a `Car` data structure, you would write `someCar.OpenDoor()` instead of `openCarDoor(someCar)`. Usually, the structure of an object comes from either a formal definition(a *class*, like in C++ and Java) or from an already existing object(a *prototype*, like in ECMAScript<sup>2</sup>).

For Prexonite Script I have chosen to use the *Procedural programming* paradigm as it is most common and easy to implement. Actually Prexonite Script can consume class based objects defined in CLI (C#) libraries but covering this would exceed the scope of this paper.

---

<sup>2</sup>ECMAScript is a scripting language standardized by the ECMA. Its most popular applications are JavaScript for websites and ActionScript for flash movies.

## Type system

The type system defines how data is stored and accessed. Every programming language has one, even if it just says “There are no types. Everything is a string.” Such a type system would force you to turn every computed result into a sequence of characters. Simple calculations like  $1 + 2$  would already cause 3 operations on strings (interpreting "1" and "2" as numbers and turning the resulting number back into a string). Additionally, the compiler can not warn you, when you are about to add a boolean value to an integer number. This type system is very easy to implement, as neither the compiler nor the runtime system has to worry about types.

A richer type system allows a language to store its computed results in their “native” format. Floating point decimal numbers are stored as floating point numbers, integers are stored as integers and strings are stored as a sequence of characters. While integers are normally represented using the *two’s complement*, floating point numbers are stored according to the *IEEE 754* standard so you can not apply the “integer addition”-operation on floating point values.

A programming language could trust its user to always write correct code but programmers too are just humans and make mistakes. Most programming language implementations therefore check whether the types of expressions are *compatible* and convert them if possible. This can either happen during the execution of the program (i.e., at *runtime*) or during the process of translation (i.e., at *compile time*). The latter is called *static typing* because the types of variables, functions and values are defined before the program runs. Such compilers can prove that a given input program is free of typing errors and therefore remove type checks at runtime. Having to define types of expressions before running the program however puts serious constraints on your code.

*Dynamic typing* on the other hand allows expressions, functions and variables to be of any type and delays type checks to the point where they are actually needed: at runtime. A variable might contain a string the first time, and an integer the second time the program is run. Since the compiler cannot make any assumptions on the contents of this variable, the type of the value stored in this particular variable has to be checked every time it is accessed which takes up precious<sup>3</sup> processor time.

---

<sup>3</sup>With today’s processor speeds this is less of a concern than it was ten years ago.

Values in Prexonite Script do have types but they are not known at compile time (*dynamic typing*). In order to represent types defined in external CLI (C#) \*.dlls, the system is required to support parameterized data types. In the scope of this paper however, I restrict the type system to simple types: *Integer*, *Real*, *Bool*, *String*, *Null*, *List* and *Hash*. *Null* represents the absence of a value, *List* is what it's name implies and accepts values of *any* type and *Hash* finally is a Dictionary that uses hash values to quickly find stored records.

### 3.3 Syntax

Once the semantics of the language are set, you have to find a way to express algorithms in a way, that both computers (more specifically: compilers) and humans can understand. You have to define the language's grammar and it's meaning. You are relatively free when designing the syntax as long as you can write the algorithm that is needed to understand code written in your language. I will introduce one class of parsers as well their advantages and disadvantages and disadvantages in section 5.

Normally you want your language to be as intuitive as possible. One way of accomplishing this, is to mimic an existing language. *C#*, *Java*, *ECMAScript* and *PHP* are all inspired by C syntax for example. They inherited operators symbols like `!=` for  $\neq$  and `&&` for  $\wedge$  as well as the semicolon for statement termination. Unlike *Pascal* which uses the semicolon as a statement *separator*.

Even though I started programming in *Visual Basic*, I grew more comfortable with C-family syntax. By ignoring whitespaces and especially line breaks, it allows me to split complicated statements over multiple lines. Also *Visual Basic* is much more verbose and redundant than equivalent C-family languages. While this actually helps code readability, it is a pain to write without the support of an intelligent editor oder IDE. Since there won't be any smart editor for my scripting language, I chose a C-family syntax.

I, however, do not believe in case-sensitivity as the majority of C-family language users do. Humans do not naturally differentiate between the variations `buzzword` and `BuzzWord`. Additionally, I prefer the words `And`, `Or`, `Xor` and `mod` to `&&`, `||`, `^` and `%` respectively, because the latter do not make sense to someone who does not know the language. But since `&&` and `||` are quite frequent operators, people with a C-family background might be comfortable with those keywords, so my compiler also accepts those two C-family-operators.

### 3.4 Implementation

There are numerous ways to implement a programming language. Traditionally you wrote a compiler that translated a program written in your language into a functionally equivalent program in machine or assembler code. Another popular method is to directly interpret the source file as it is read, which eliminates the need for separate compilation at the cost of execution speed. Java uses a hybrid approach: It's source code is compiled to a lower level byte code, that is then interpreted<sup>4</sup> by a virtual machine. A machine that is simulated by another machine.

I too decided to implement my language with the help of a virtual machine, so I can focus the compiler on the act of translation instead of runtime issues. I even chose to write the machine myself so it would perfectly fit the needs of my language. Theoretically, others could write their own compilers that target my virtual machine.

**Stack machine** A machine that uses a stack to store values. Operations are performed on the values that currently are “on top” of the stack. Since stack machine instructions do not need to specify any source and target registers, they are smaller and computed faster. But as stacks can be infinite in theory, they cannot be implemented in hardware and have to reside in the main memory as data structures. Thus every operation involves reading from and writing to memory, which is slow.

**Register machine** A machine that uses named “slots” to store values. Although it's instructions are more complicated, the fact that registers can be implemented in hardware eliminates the need for main memory access. At least as long as all required data is loaded into the registers.

As I am not bound to hardware limitations, I can go with the stack machine design as it is easier to implement and my compiler won't have to worry about registers. Unless the machine is supposed to just evaluate mathematical expressions, a second stack is commonly used to store the return addresses of subroutines. So my virtual stack machine uses 2 separate stacks: One for data and one for function calls.

---

<sup>4</sup>Today's java virtual machines can compile important portions of byte code to machine code to improve performance.

## 4 Scanner

The first phase of the compilation process is called *lexical analysis* and performed by the *scanner* (sometimes also called “*lexer*”). Its purpose is to filter and group the stream of characters from the input file into more meaningful units. These units are called *tokens* and typically consist of a *token code* and the character sequence they represent. Characters with no meaning to compiler are usually dropped during this phase.

The scanner tries to recognize patterns in the input stream and turns matching character sequences into tokens with the token code being determined by the pattern that has been applied.

### 4.1 Regular expressions

Regular expressions are patterns written in a formal language originally invented by *Stephen Kleene* ([3]) to describe automata. The terms *symbol*, *word*, *alphabet* and *language* are well defined in language theory:

**Symbol:** an element in the input stream, in our case a character.

**Word:** a finite sequence (concatenation) of *symbols*. Also called *String*.

**Alphabet:** a finite set of *symbols*. For compilers this is usually the set of characters defined in either ASCII oder Unicode.

**Language:** the set of all *words* that can be formed from *symbols* in the *alphabet*.

Understanding the definition of regular languages is not very difficult, as there are only three operations:  $RS$  is the concatenation (“ $R$  is followed by  $S$ ”),  $R|S$  is the alternation (“Either  $R$  or  $S$ ”) and  $R^*$  means “any number of  $R$ s concatenated”. Brackets have the same meaning as in mathematics

language	in english	possible words
$ab$	exactly $a$ follwed by $b$	$ab$
$a b$	just $a$ or just $b$	$a$ or $b$
$a^*$	any number of $a$ 's	$\epsilon$ , $a$ , $aa$ , $aaa...$
$(a b)^*$	any number of $a$ 's or $b$ 's	$a$ , $aa$ , $ab$ , $aaabaabbabba...$

The  $\epsilon$  (epsilon) stands for the *empty string*.

To deal more easily with multiple regular expressions, I am going to name them and use those names like symbols in other expressions. Be aware though that this is just for convenience: Regular language grammars cannot have cycles or recursion.

```

letter → A|B|⋯|Z|a|b|⋯|z
digit  → 9|8|⋯|0
id     → letter(letter|digit)*

```

id is the *language* that includes all *strings* that consist of one letter followed by any number of letters and numbers. This might be the definition of an identifying name (an *identifier*) in your programming language.

## 4.2 Implementation

Implementing a scanner by hand is very tedious and unless you know exactly what you are doing, automatically generated scanners are probably faster. The most widely known scanner generator is called *lex*. Although the current Prexonite Script compiler uses *C# Flex*<sup>5</sup>, I will use the scanner generator that comes with the “compiler-compiler” *Coco/R* for demonstration purposes.

```

DIGIT  → 9|8|⋯|0
LETTER → A|B|⋯|Z|a|b|⋯|z
number → DIGITDIGIT * |DIGITDIGIT * '.' DIGITDIGIT *
id     → LETTER(LETTER|LETTER) *
plus   → '+'|'-'
times  → '*'|'/'
assign → '='

```

Such a formal definition would be written for *Coco/R* as shown in listing 1.

---

<sup>5</sup>Licensed under the GPL. Available at <http://sourceforge.net/projects/csflex/> Apparently no longer in development.

Listing 1: A scanner definition for *Coco/R*

```
CHARACTERS
  letter      = 'A'..'Z' + 'a'..'z'.
  digit       = '0'..'9'.
TOKENS
  number      = digit { digit }
              | digit { digit } "." digit { digit }.
  id          = letter { letter | digit }.
  plus        = "+" | "-".
  times       = "*" | "/".
  assign      = "=".
```

## 5 Parser

Once the scanner has grouped the character input into a stream of tokens, a compiler needs to determine the relationship between individual tokens (besides their chronological order). This second step is called *syntactic analysis*. Its goal is to create an *abstract syntax tree* (or *AST*), a rooted tree graph where the leafs represent literal constants or identifiers and the nodes represent operations.

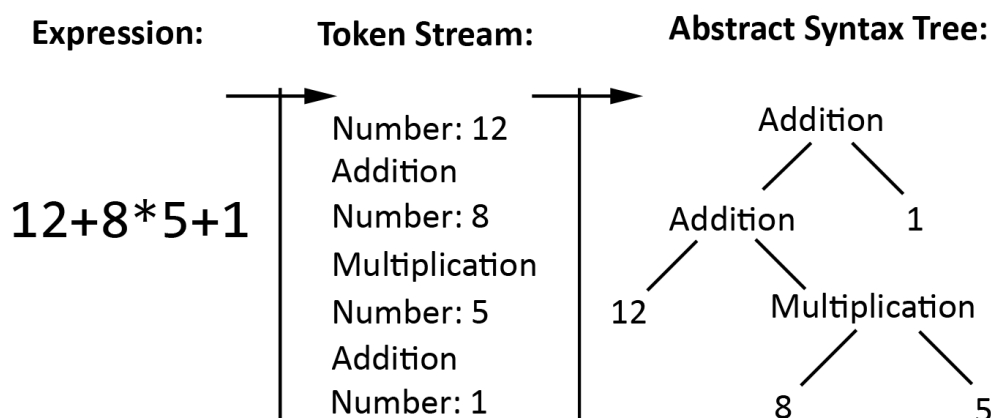


Figure 2: Translation of the expression  $12+8*5+1$  to an AST

### 5.1 Context-free grammar

If you, as a human, read  $1 + 2$ , you will recognize this as the addition of one and two because you know that  $+$  is an operator that has two operands, one on the left and one on the right side. You try to apply certain rules to the words (tokens) that you see on the paper. That is exactly what parsers do, except that their rules do not come from basic education but your language's formal grammar.

I used a common approach called *syntax-directed translation* that lets me derive my parser from the context-free grammar of my language. Like its name implies, the rules in a context-free grammar do not depend on the context of the tokens to be parsed. This "restriction" has the advantage that it is easy to define such a grammar formally.

The most common grammar notation is called *Backus Naur Form* or *BNF* for short. It consists of a set of productions, each one of the form

$$V \rightarrow r$$

where  $V$  is a non-terminal symbol, also called the "left hand side (of the production)", and  $r$  is a string of terminal and/or non-terminal symbols.

A terminal symbol refers to a certain type of token and is matched when a token of the corresponding type appears in the stream while a non-terminal symbol is just the "name" of a production that has to be expanded (replaced by the terminal and/or non-terminal symbols it consists of) in order to be matched.

A grammar uses one of the productions, the so called *start symbol*, as the rule to match the whole input program against. By convention, this is always the first production in the definition of a grammar.

In order to match expressions like<sup>6</sup>  $9-5+2$ ,  $3-1$  or  $7$  you could design the corresponding grammar to treat such strings as "a list of numbers separated by either  $+$  or  $-$ ":

$$\text{List} \rightarrow \text{List} + \text{Digit} \tag{1}$$

$$\text{List} \rightarrow \text{List} - \text{Digit} \tag{2}$$

$$\text{List} \rightarrow \text{Digit} \tag{3}$$

$$\text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \tag{4}$$

The right hand sides of the three productions with List on the left hand side can be combined into the equivalent production

$$\text{List} \rightarrow \text{List} + \text{Digit} \mid \text{List} - \text{Digit} \mid \text{List} - \text{Digit} \mid \text{Digit}$$

The symbols  $+$ ,  $-$ ,  $0$ ,  $1$ ,  $\dots$  and  $9$  are treated as terminal symbols. The non-terminal *Digit* can therefore be expanded to any of our ten digit terminals.

When you read "the production *for* a non-terminal", the production with the non-terminal on it's left hand side is meant. Production **(3)** states that one digit alone already is a list. According to productions **(1)** and **(2)**, a list followed by one of the two operators and a digit is also a list.

---

<sup>6</sup>The examples in this chapter are taken from [1].

We can show that the expression  $9 - 5 + 2$  is a *word* in the *language* defined by the above grammar:

- $9$  is a *List* according to **(3)**, because  $9$  is also a *Digit*.
- $9 - 5$  is a *List* according to **(2)**, because  $9$  is a *List* and  $5$  a *Digit*.
- $9 - 5 + 2$  is a *List* according to **(1)**, because  $9 - 5$  is a *List* and  $2$  a *Digit*.

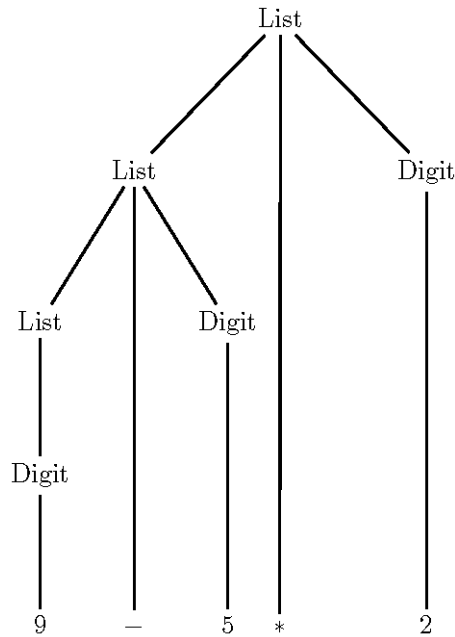


Figure 3: Parse tree of the input  $9 - 5 + 2$

Figure 3 is a way to visualize the way  $9 - 5 + 2$  can be recognized as a word of the language defined in our grammar. This tree structure is called *parse tree*. A node represents a non-terminal with its label begin the left hand side and its branches the right hand side while the leafs represent terminals and are all aligned on the same height to highlight the input string. As such a parse tree is ordered, it shows how a certain *word* ( $9 - 5 + 2$  in this case) can be *derived* from a given grammar. The reverse process, finding a parse tree for a given *word*, is called parsing.

The code blocks in Prexonite Script are similar to a list. They are defined as a sequence of statements, each one terminated by a semicolon, enclosed in braces. A corresponding grammar could start like this:

$$\begin{aligned} \text{Block} &\rightarrow \{ \text{OptionalStatementList} \} \\ \text{OptionalStatementList} &\rightarrow \text{StatementList} \mid \epsilon \\ \text{StatementList} &\rightarrow \text{Statement} ; \text{OptionalStatementList} \end{aligned}$$

Note that the second alternative for *OptionalStatement* is  $\epsilon$ , the empty string. This makes the *OptionalStatementList* optional since the production would also be matched for an input like "{}".

## 5.2 Ambiguities

Consider the following simplification of our example previous grammar:

$$\text{String} \rightarrow \text{String} + \text{String} \mid \text{String} - \text{String} \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \quad (5)$$

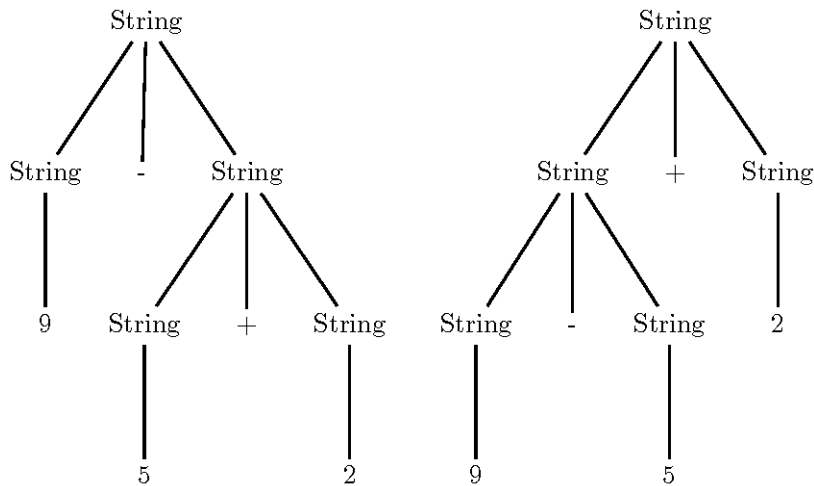


Figure 4: Two parse trees for  $9 - 5 + 2$

Figure 4 shows two possible ways to derive the word  $9 - 5 + 2$  from the grammar defined by (5). Now, there is nothing wrong with either of those two parse trees as a mathematically correct *abstract syntax tree* can be created from either but while the first one happens to be identical to the resulting AST, the second one requires additional transformations. Your parser gets a lot simpler if you design the grammar in such a way that the parse tree resembles the resulting AST as closely as possible.

## Operator associativity

To create a mathematically correct parse tree, we have to introduce operator associativity.

$$(x * y) * z \neq x * (y * z) \text{ for all } x, y, z \in S \quad (6)$$

An operator that satisfies **(6)** is called *associative* and as such it does not matter in which order its operands are processed. Now, the *subtraction* is clearly non-associative. In general, parentheses must be used to indicate the order of evaluation: In this example, you would either have to write  $(9-5)+2$  or  $9-(5+2)$ . There, however, is a convention for the subtraction operator: If parentheses are omitted, the expression must be evaluated from left to right. This is called *left-associativity*. An example for *right-associativity* would be  $x^{y^z} = x^{(y^z)} \neq (x^y)^z$ .

Most of the time it is easier to also treat associative operators like  $*$  or  $+$  as left-associative operators since the resulting grammar must no longer make a difference between Addition and Subtraction. The following grammar can be proven to define the exactly same language like **(5)** does, but makes the right parse tree from figure 4 impossible.

$$\begin{aligned} \text{Left} &\rightarrow \text{Left} + \text{Digit} \\ \text{Left} &\rightarrow \text{Left} - \text{Digit} \\ \text{Left} &\rightarrow \text{Digit} \\ \text{Digit} &\rightarrow 0|1|2|3|4|5|6|7|8|9 \end{aligned}$$

## Operator priority

Consider the expression  $9+5*2$ . It can be interpreted in two ways:  $(9+5)*2$  or  $9+(5*2)$ . The associativity of  $+$  and  $*$  doesn't help you in this case. They are both left-associative which would lead to the conclusion that  $(9+5)*2$  must be correct. This problem is solved by respecting operator precedence.

$$\begin{aligned} \text{left-associative: } &+ - \\ \text{left-associative: } &* / \end{aligned}$$

For those two priority levels we define the two non-terminals *Expr* and *Term* as well as the terminal symbol *digit*. The third non-terminal *Factor* is used as a building block for our grammar.

$$\text{Expr} \rightarrow \text{Expr} + \text{Term} \mid \text{Expr} - \text{Term} \mid \text{Term} \quad (7)$$

$$\text{Term} \rightarrow \text{Term} * \text{Factor} \mid \text{Term} / \text{Factor} \mid \text{Factor} \quad (8)$$

$$\text{Factor} \rightarrow \text{digit} \mid ( \text{Expr} ) \quad (9)$$

Figure 5 shows the only possible parse tree for  $9 * (5 + 2)$  and Figure 5 is an AST for this expression. Note, that after the syntactical analysis, parentheses are no longer needed.

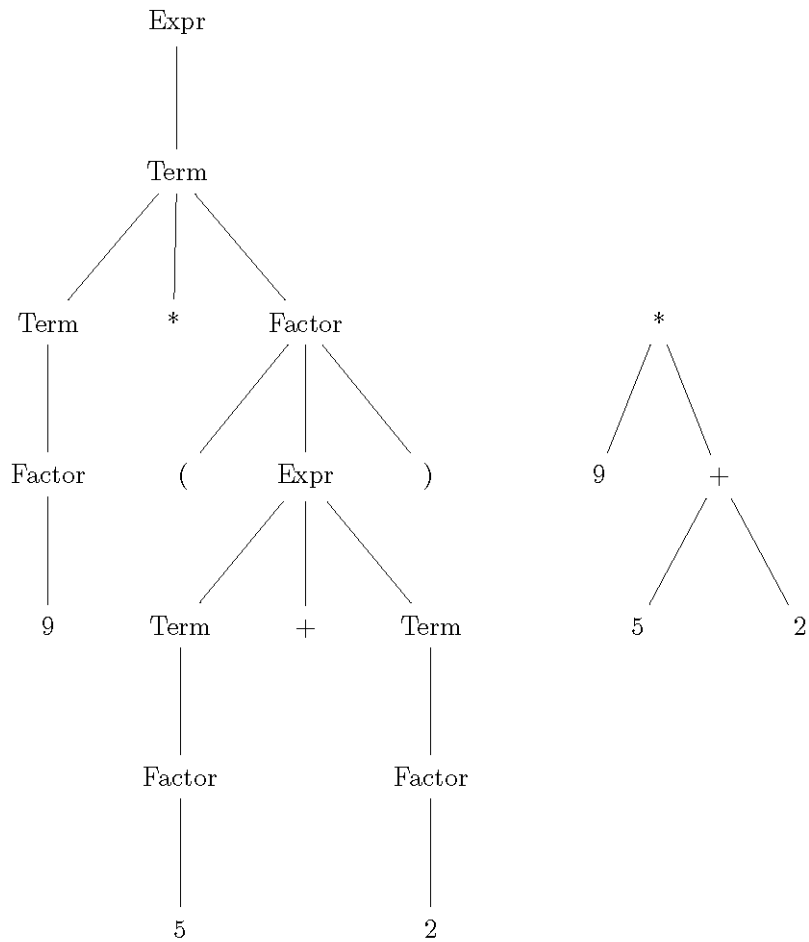


Figure 5: The parse tree and AST for  $9 * (5 + 2)$

### 5.3 Syntax-directed translation

The idea behind *syntax-directed translation* is to exploit the similarities between the parse tree and the AST to be generated, provided that you designed your grammar in such a way, that it tries to match the structure of the AST.

A syntax-directed *definition* describes the syntactical structure of the input with the help of a context-free grammar. Every grammar symbol receives a set of *attributes* and every production has a set of *semantic rules*. These semantic rules compute the attributes of the symbols a production consists of.

The context-free grammar and the semantic rules together, form a syntax-directed definition. If  $X$  is a grammar symbol and  $b$  is an attribute of  $X$ , the notation  $X.b$  represents the value of the attribute  $b$ . A parse tree is called an *attributed parse tree* if its nodes have attributes.

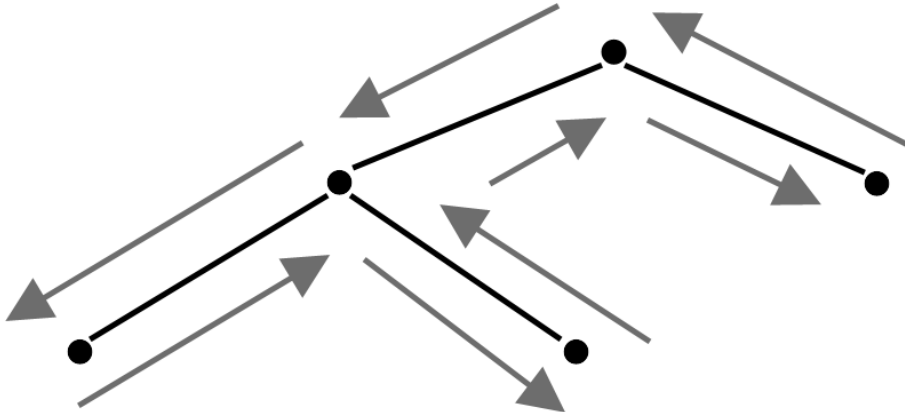


Figure 6: Sample of the order nodes are visited in a depth-first pass.

#### A simple calculator

A syntax-directed definition can be seen as a function that maps an input string  $x$  to an attributed parse tree. The following sample shows a syntax-directed definition that calculates an attribute  $val$  for each symbol in the grammar according to mathematical rules.

Given the syntax-directed definition from figure 7, we obtain the value of an expression like  $2 + 3$  by evaluating the attribute  $val$  of the attributed parse tree's root node. The root is derived from  $\text{Expr} \rightarrow \text{Expr}_1 + \text{Factor}_1$  and the rule that defines  $\text{Expr}.val$  depends on  $\text{Expr}_1.val$  and  $\text{Factor}_1.val$ .

Production	Semantic rule
$\text{Expr} \rightarrow \text{Expr}_1 + \text{Factor}_1$	$\text{Expr.val} := \text{Expr}_1.\text{val} + \text{Factor}_1.\text{val}$
$\text{Expr} \rightarrow \text{Expr}_1 - \text{Factor}_1$	$\text{Expr.val} := \text{Expr}_1.\text{val} - \text{Factor}_1.\text{val}$
$\text{Expr} \rightarrow \text{Factor}_1$	$\text{Expr.val} := \text{Factor}_1.\text{val}$
$\text{Factor} \rightarrow 0$	$\text{Factor.val} := 0$
$\text{Factor} \rightarrow 1$	$\text{Factor.val} := 1$
$\vdots$	$\vdots$
$\text{Factor} \rightarrow 9$	$\text{Factor.val} := 9$
$\text{Factor} \rightarrow (\text{Expr}_1)$	$\text{Factor.val} := \text{Expr}_1.\text{val}$

Figure 7: A syntax-directed definition that acts as a calculator.

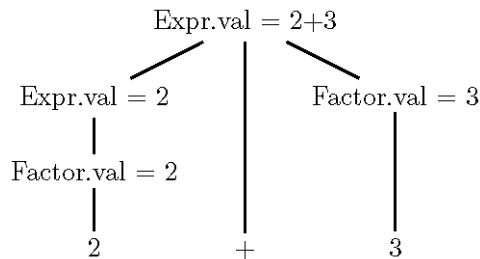


Figure 8: *val* attributes throughout the attributed parse tree of  $2 + 3$ .

$\text{Expr}_1$  in turn is derived from the production  $\text{Expr} \rightarrow \text{Factor}_1$  and its attribute *val* has the same value as  $\text{Factor}_1.\text{val}$ .  $\text{Expr}_1$ , however, has no such dependencies but instead sets *val* to 2. This is shown in figure 8. In other words, the attributed parse tree is traversed in a depth-first manner as illustrated in figure 6.

## Symbol table

Unlike that calculator, real world programming languages operate mostly on symbolic names: variables, functions, types and so on. Compilers normally keep track of the meaning of names (*identifiers*) using a symbol table. During code generation, the compiler will lookup the meaning of identifiers it encounters and emit instructions accordingly. The amount of data stored in a symbol table entry depends on the language. Compilers for statically typed languages add type information for instance.

<b>Id</b>	<b>Entry</b>
sum	local variable
abs	function
results	global variable
$\vdots$	$\vdots$

## 5.4 Constructing the AST

Your compiler is not supposed to evaluate expressions. What you need for the next phase is the AST. Instead of actually computing the result, you create a data structure that represents the steps that need to be taken in order to obtain the result. The principle is very similar: Every symbol in the grammar has an attribute  $n$  which contains an AST node. This AST node represents the process of computing the attribute val.

Production	Semantic rule
$\text{Expr} \rightarrow \text{Expr}_1 + \text{Factor}_1$	$\text{Expr}.n := \text{AstAdd}(\text{Expr}_1.n, \text{Factor}_1.n)$
$\text{Expr} \rightarrow \text{Expr}_1 - \text{Factor}_1$	$\text{Expr}.n := \text{AstSub}(\text{Expr}_1.n, \text{Factor}_1.n)$
$\text{Expr} \rightarrow \text{Factor}_1$	$\text{Expr}.n := \text{Factor}_1.n$
$\text{Factor} \rightarrow 0$	$\text{Factor}.n := \text{AstConst}(0)$
$\text{Factor} \rightarrow 1$	$\text{Factor}.n := \text{AstConst}(1)$
...	
$\text{Factor} \rightarrow 9$	$\text{Factor}.n := \text{AstConst}(9)$
$\text{Factor} \rightarrow (\text{Expr}_1)$	$\text{Factor}.n := \text{Expr}_1.n$

Figure 9: syntax-directed definition for AST generation.

So instead of evaluating  $2 + 3$ , we create an AST node that represents the addition of 2 and 3. figure 9 shows how the syntax-directed definition from the previous sample would have to be altered. When applying this transformation to  $9 - 5 + 2$ , we receive the following expression:

$\text{AstAdd}(\text{AstSub}(\text{AstConst}(9), \text{AstConst}(5)), \text{AstConst}(2))$

## 5.5 Translation scheme

Unless you are implementing your compiler in a purely functional language, you may want more control over the execution of the semantic rules in your parser. A *translation scheme* is a context-free grammar with small bits of code (*semantic actions*) integrated into the right hand side of productions. This way, the order in which the semantic rules of a syntax-driven definition are applied can be explicitly specified. Semantic actions are written between braces, as shown in

$\text{Decl} \rightarrow \text{var } id_1 \{ \text{declare}(id_1.val); \} = \text{Expr}_1 \{ \text{AstAssign}(id_1.val, \text{Expr}_1.n); \} ;$

This is a sample of a variable declaration and initialization. It might be vital, that the declare-action is executed before the assign node is created.

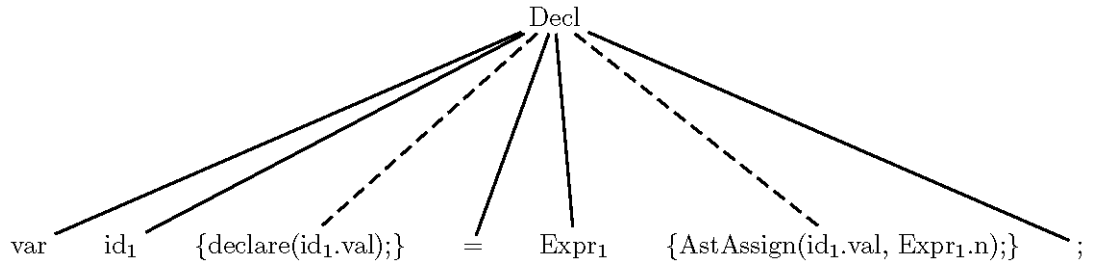


Figure 10: Semantic actions

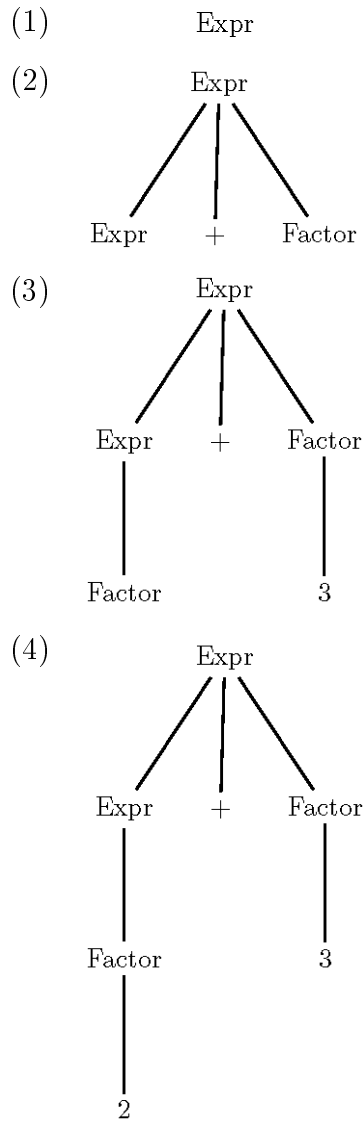


Figure 11: States of a top-down parser

## 5.6 Parsing

The process of constructing a parse tree from given token stream is called *parsing*. There are several algorithms of which most fall into one of two categories: *top-down* or *bottom-up*. Top-down<sup>7</sup> parsers start constructing the parse tree from the root (the top) and descend to the leaves while bottom-up parsers do the exact opposite. This paper focuses on top-down parsers as they are easier to understand and not as difficult to write by hand as bottom-up parsers.

Basically there exists a parser for every context-free grammar that takes at most  $O(n^3)$ <sup>8</sup> time to parse a string of  $n$  tokens. Cubic time, however, is too expensive for practical use, therefore algorithms have been developed, that run in linear time but introduce some restrictions on the set of context-free grammars they can parse.

*Top-down* parsers basically begin with the start symbol and expand non-terminals according to the tokens found in the token stream. Figure 11 shows how the parse tree is created from  $2+3$  expansion by expansion. We assume that Expr is the start symbol.

### Left recursion

When it comes to the actual implementation of top-down parsers, we face a serious problem: algorithms that parse the token stream from left to right, run into infinite loops if the underlying grammar contains *left recursive productions*.

$$\text{Expr} \rightarrow \text{Expr} + \text{Factor}$$

The first step in the process of expanding Expr is to expand Expr, which in turn requires the expansion of Expr and so on. In order to parse our language in a top-down approach, we need to remove left recursion from our grammar. [1] presents an algorithm that transforms a given grammar without cycles and  $\epsilon$ -Productions (productions with empty right-hand sides) into an equivalent grammar without left-recursion.

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow ( E ) \mid \text{id} \end{aligned}$$

---

<sup>7</sup>Also called LL-parser

<sup>8</sup>Taken from [1]

Applying that algorithm to the above grammar, we get

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \epsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F T' \mid \epsilon \\ F &\rightarrow ( E ) \mid \text{id} \end{aligned}$$

We can call  $E'$  the *right-hand side of  $E$* .

### Alternatives

When expanding a production with multiple alternatives, a top-down parser has to decide, which one to apply. A naive approach is to just try the first alternative and see, if a parse tree can be built with the underlying token stream. Should the tokens match, the parser accepts the alternative, otherwise rejects it. In the latter case, the parser rewinds the token stream to the position, where the apparently wrong alternative has been chosen and tries the second alternative. Such a parser would be classified as  $LL(*)$ , as it parses from *Left* to *right*, uses the *Leftmost* derivation and looks an arbitrary (i.e.,  $*$ ) number of tokens ahead. Unfortunately, this approach is not very efficient, especially not with highly ambiguous grammars.

If we need a fast parser,  $LL(1)$  is the way to go. Such parsers only read one token ahead (the *lookahead token*) to choose which production to expand. That, however, puts serious constraints on the grammar, as it has to be designed in such a way, that all alternatives start with a different token.

$$\begin{aligned} \text{Stmt} &\rightarrow \text{if} ( \text{Expr} ) \text{Block} \\ \text{Stmt} &\rightarrow \text{if} ( \text{Expr} ) \text{Block} \text{ else } \text{Block} \end{aligned}$$

Here, the first terminal of both alternatives is *if*. With just one lookahead token, the parser can't decide, which alternative is the correct one. In many cases, such ambiguities can be removed by *left factoring* the productions. The concept is similar to factoring out in mathematics: We extract the largest common prefix, in this case *if*, and defer the decision.

In general,  $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$  becomes

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \\ A' &\rightarrow \beta_2 \end{aligned}$$

The main problem with LL(1) parsers is to find a grammar for your source language, that can be used to develop the parser. While it is relatively easy to apply the algorithms for the elimination of left recursion and for left factoring, the result is often more complex and difficult to understand. This can be a problem, since we want our parse tree to resemble the abstract syntax tree.

A solution is to vary the number of lookahead tokens depending on the production. The Prexonite Script parser for example uses one lookahead token for the majority of the productions in the grammar, while some constructs like variable declarations and labels require a second token to be read. Of course it would have been possible to merge labels ( $\text{Label} \rightarrow \text{id} \text{:}$ ) with variable assignments ( $\text{Assign} \rightarrow \text{id} = \text{Expr}$ ) but as they have nothing in common semantically, I decided to depend on the colon token in order to disambiguate those two statements.

### Parser generator

Most of the time, parsers are not written by hand, but by a specialized tool, a parser generator. It accepts some sort of grammar definition file, which normally also includes semantic actions, written in the implementation language, and transforms it into a class or a set of functions that does the parsing. This saves a lot of time, since such tools can also detect ambiguities and left recursion and warn the user.

I chose *Coco/R for C#*<sup>9</sup> for my project, mainly for its straightforward handling of LL(1) conflicts (productions that require more than one lookahead symbol).

Coco/R generates a separate function for every production, which might not be as efficient as using a table but helps you to find errors. The snippet on the next page is taken from the Prexonite Script attributed grammar file. It contains the definition of the explicit **goto** statement.

---

<sup>9</sup>Licensed under the GPL. Available at <http://www.ssw.uni-linz.ac.at/coco/>

```

ExplicitGoTo<AstBlock block>    (.  string id; .)
=
    goto
    Id<out id>
    (. block.Statements.Add(
        new AstExplicitGoTo(this, id)); .)
.

```

The code inside (. and .) is pure C# and will be injected into the resulting function as you can see in the next snippet. The **goto** on line 3 refers to a terminal symbol, while `Id<out id>` is a reference to the non-terminal *Id* with the attribute `id`. The following semantic action then adds a new AST node to the current block.

```

void ExplicitGoTo(AstBlock block) {
    string id;
    Expect(_goto);
    Id(out id);
    block.Statements.Add(new AstExplicitGoTo(this, id));
}

```

It would be possible to emit the target code directly from within the semantic actions of the attributed grammar, but that would make certain optimizations difficult to implement. I therefore decided to first explicitly create an abstract syntax tree.

## 6 Code generation

In this last step of translation, we walk the abstract syntax tree generated in the previous step and emit bytecode instructions for the virtual machine discussed in the next chapter. More sophisticated compilers would choose *three-address-code* as an intermediate representation as it is more suited for data flow analysis.

### 6.1 Simple optimizations

Optimizations are transformations of the program that make it “better” while retaining its functionality. What “better” is, depends on the situation: For most applications it means *faster*, but if the program runs in an environment with limited resources a smaller memory consumption and smaller executable size might be preferable.

Also, the transformation has to justify the cost of its implementation. Unless its effect is noticeable at runtime, it is waste of time. There, however, are a number of optimizations that are relatively simple to implement and can make a huge difference at runtime.

### Constant folding

It is sometimes convenient to write  $60*60$  than  $3600$  because the former suggests, that the number represents a time span. A compiler that supports constant folding can evaluate and substitute constant expressions at compile time. This way, the program won't have to compute  $60^2$  in every iteration at runtime.

```
for (var s = 0; s < 60*60; s++)
  println(s + " seconds\n" +
    " since midnight");
```

The implementation is pretty simple. In this case  $60*60$  is represented in the AST as a *multiplication* node with two *constant* nodes as its operands. When the compiler gets to the *multiplication* node, it checks whether both operands are *constant* nodes, in which case the *multiplication* node is replaced by a new *constant* node that represents  $3600$ . Figure 12 shows an AST before and after constant folding.

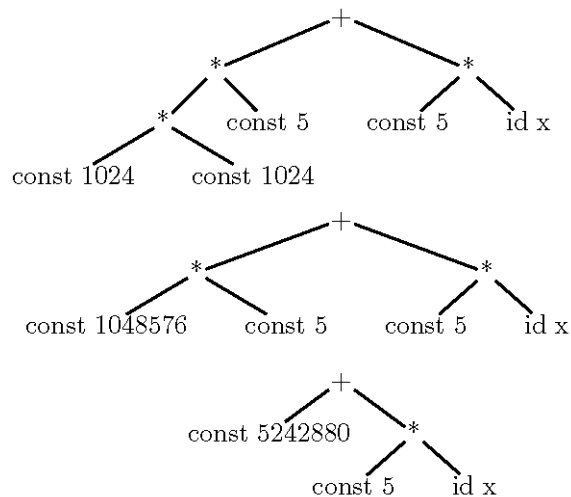


Figure 12: Constant folding of  $1024*1024*5 + 5*x$

Another very similar class of optimizations are algebraic transformations. Simple examples include the removal of redundant computations like in  $(x + 0) * 1 = x$ . However, you have to be careful when applying mathematical rules to programs. In a purely functional language it is ok to reduce  $1 + f(a)*0$  to 1. In C, on the other hand,  $f$  might have side effects like writing a log entry. By removing the call to  $f$  we change the effect of the program.

You can also take advantage of knowledge about the target machine to optimize computations. Some operations might take more time than others.  $x^2$ , for example, usually takes longer than  $x * x$ . The same is true for  $2 * x$  and  $x + x$ .

### Dead code elimination

If the compiler can prove, that a certain block of code is never executed, it is safe to remove that block. While this usually requires control flow analysis, there are cases where a quick look at the AST serves as proof: Should an **if** structure have a constant expression as its condition, the **if** structure can be replaced by the block it protects.

## 6.2 Emitting code

Generating code for a stack based machine from an AST is very simple in most cases. Basically, the compiler has to translate the program to a sort of postfix or *reverse polish notation* (RPN). In RPN, the operators follow their operands. For instance, to add two and five one would write  $2\ 5\ +$  and to multiply the sum of two and five by three one writes  $2\ 5\ +\ 3\ *$ .

To generate code, the compiler recursively traverses the abstract syntax tree, starting at the root as shown in Figure 13. The order in which the nodes are visited is

```
2
5
+
3
*
```

which just happens to be the order in which the expression would be written in postfix notation. In the Prexonite Script compiler, all AST nodes have a method called `EmitCode`. The snippet on the next page is taken from the `AstBinaryOperator` and demonstrates very well the depth first traversal.

```
public override void EmitCode(CompilerTarget target)
{
    LeftOperand.EmitCode(target);
    RightOperand.EmitCode(target);
    EmitOperator(target);
}
```

One challenge is left, though: jumps to labeled instructions. When the compiler comes across a **goto** statement, the actual target address might not be known. The compiler cannot predict how many instructions will be emitted after the **goto** statement. It has to wait until the whole block is translated and then insert the target addresses into not yet resolved jump instructions according to the label entries symbol table.

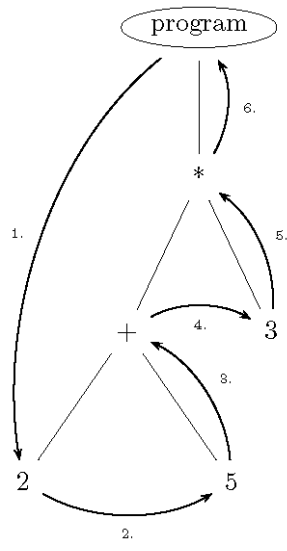


Figure 13: Depth-first traversal

## 7 Runtime environment

Runtime describes the operation of a computer program, the duration of its execution, from beginning to termination ([2]). The runtime environment includes everything from the executing machine over memory management to a standard library. In the case of Prexonite Script, programs are executed by the virtual stack machine, memory management is inherited from the implementations runtime environment and the standard library is mostly integrated into the virtual machine.

### 7.1 Virtual stack machine

A stack machine is a model of computation in which the computer's memory takes the form of one or more stacks. A stack machine has a number of stacks. A machine with just one stack is limited to the computation of expressions; it does not allow for recursion.

In order to implement a virtual machine that can be used for general purpose programming, at least two stacks are required: one for the evaluation of expressions and one for return addresses from procedures. Whenever a procedure is called, the address of the next instruction after the procedure call is pushed onto the *call stack*. Once program control has reached the end of the called procedure, it jumps back to the address stored on top of the call stack.

The actual implementation of a virtual stack machine is not that complicated. It is basically a huge **switch** statement, enclosed in a loop. An *instruction pointer* keeps track of the next instruction to execute. Additional features like global and local variables can be implemented in various ways. Prexonite Script uses a data structure it calls `StackContext` to keep track of local variables, parameters as well as the return address.

Listing 2: A very simple virtual stack machine.

```
int execute(Instruction [] code)
{
    int pointer = 0;
    Stack<int> data = new Stack<int>();
    Stack<int> addresses = new Stack<int>();
    while(pointer < code.Length)
    {
        Instruction ins = code[pointer];
        int left, right;
        switch(ins.OpCode)
        {
            case ADD:
                left = data.Pop();
                right = data.Pop();
                data.Push(left + right);
                pointer++;
                break;
            case MUL:
                left = data.Pop();
                right = data.Pop();
                data.Push(left * right);
                pointer++;
                break;
            // and so on
            case CALL:
                addresses.Push(pointer+1);
                pointer = ins.Operand;
                break;
        }
    }
    return data.Peek();
}
```

## 8 Closing comments

According to [6], the implementation of the first FORTRAN compiler took 18 man-years. The techniques that have been developed since then, enabled me to write a working implementation of Prexonite Script in a bit more than half a year. The project has been challenging at times since compiler theory involves concepts of mathematics and computer science not taught in high school. Fortunately, already very few compiler construction techniques suffice to successfully implement a simple language.

Prexonite Script turned out very well and provides a lot of features, that would exceed the scope of this paper like access to objects defined in C#, nested functions or coroutines.

I am, however, not content with the performance of programs written in Prexonite Script compared to equivalent programs in JavaScript or PHP. Especially when it comes to recursion, my implementation is about six times slower than the popular languages. Unfortunately, I did not have the time to profile the virtual machine to find the bottleneck, but I suspect that a completely rewritten execution engine would be necessary to improve performance significantly.

As much as I would have loved to include the source code for Prexonite Script in the paper, I doubt that anyone is interested in over 460 pages, containing more than 41.000 lines of code. Should you nonetheless want to look at the implementation of Prexonite Script, you can download the source code from:

**<http://www.sealedsun.ch/press/prexonite/>**

## References

- [1] Aho, Alfred V. and Sethi, Ravi and Ullmann, Jeffrey. 1999. Compilerbau. Oldenbourg. München.
- [2] Wikipedia, Runtime,  
<http://en.wikipedia.org/wiki/Runtime> (28.10.2007)
- [3] Wikipedia, Regular Expressions,  
[http://en.wikipedia.org/wiki/Regular\\_Expressions](http://en.wikipedia.org/wiki/Regular_Expressions) (28.10.2007)
- [4] Wikipedia, History of programming languages,  
[http://en.wikipedia.org/wiki/History\\_of\\_programming\\_languages](http://en.wikipedia.org/wiki/History_of_programming_languages)  
(28.10.2007)
- [5] Wikipedia, Plankalkül,  
<http://en.wikipedia.org/wiki/Plankalk%C3%BCl> (28.10.2007)
- [6] Backus, J.W., R.J. Beeber, S. Best, R. Goldberg, L.M. Haibt, H.L. Herrick, R.A. Nelson, D. Sayre, P.B. Sheridan, H. Stern, I. Ziller, R.A. Hughes and R. Nutt. 1957. The Fortran automatic coding system, Western Joint Computer Conference.