

# Using Compiler Engineering Algorithms for Building Payoff Languages

Oliver Caps  
oliver.caps@dkib.com

RMT Model Validation Rates  
Dresdner Bank



Introduction

Basic Concepts of Compiler Engineering

Using Compiler-Compilers

In-depth: Parsing LL(1) Grammars

Introduction

Basic Concepts of Compiler Engineering

Using Compiler-Compilers

In-depth: Parsing LL(1) Grammars

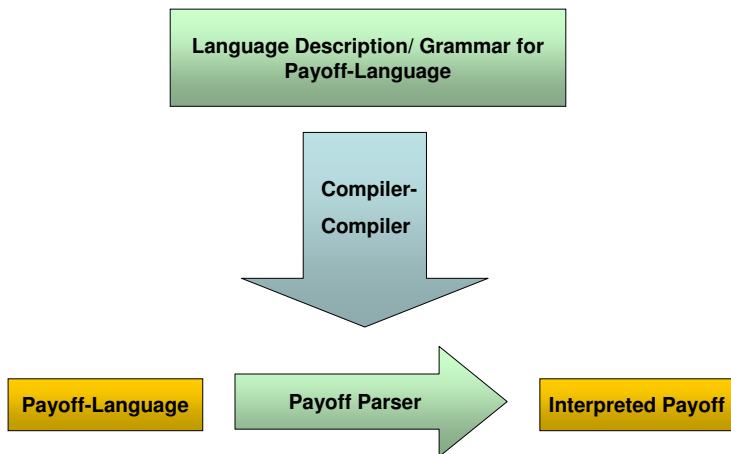
# What is Compiler Engineering ?

- A *compiler* or *parser* translates texts written in one language (source language) into another language (object language).
- Compiler Engineering is a discipline in computer sciences that systematically studies procedures for efficient parsing.
- In particular, there exist algorithms for parsing certain types of languages.
- But if there is an algorithm for compiling, one can write a computer program implementing this algorithm,
- and one ends up with a *compiler-compiler* or *parser generator*.
- Using compiler-compilers and a small input describing the language one can quickly generate powerful compilers.

# Compiler Engineering Techniques in Finance

- One of the main jobs of a Quant is to translate term sheets into coding languages.
- In the last years payoffs have been got much more complicated, think of
  - hybrid payoffs depending on several assets
  - trigger, TARN or Cliquet features
  - callability on-top
- and the given timeframe for implementing products has strongly decreased.
- Idea that can manage these challenges:
- Develop a payoff language as an intermediate layer and use compiler engineering techniques for efficient parsing of these languages.

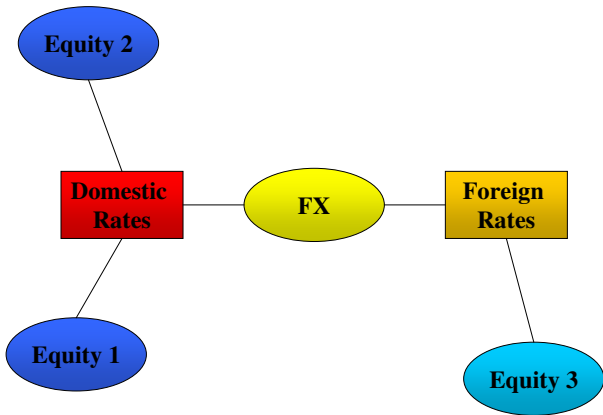
# Compiler Engineering Techniques in Finance



# Monte-Carlo Framework

- We assume that we have given a Monte-Carlo framework that provides for observation times  $t$  vectors of simulated values like
  - $\text{SwapRate}(t, 10Y)$
  - $\text{FX}(t, \text{EUR}, \text{USD})$
  - $\text{Equity}(t, \text{ALVG.DE})$
  - numeraires  $N(t)$
- The simulated values could come from any model like a BGM, Markovian HJM, or stochastic volatility asset model.
- Cf. [OC] how to construct a framework of this type.

# Monte-Carlo Framework



# Monte-Carlo Framework

- Most of the existing products can be described by
  - a time-scale  $t_1 < t_2 < \dots < t_n$ .
  - specifications  $C_i$  of cashflows occurring at time  $t_i$  that may depend on earlier cashflows and on trigger conditions.
  - Bermudan/American cancel rights terminating the product
- For example, take a hybrid TARN product:

$$C_i = \begin{array}{l} \text{If} \\ \text{then} \\ \text{else} \end{array} \begin{array}{l} \left( \sum_{j=0}^{i-1} C_j < 0.15 \right) \\ \max \left[ \left( \frac{FX(t_i, EUR, USD)}{FX(t_{i-1}, EUR, USD)} - 1 \right), 2 \cdot \text{SwapRate}(t_i, 10Y) \right] \\ 0 \end{array}$$

# Monte-Carlo Framework

- Goal: Translate  $C_i$  *algorithmically* into vectors of simulated values  $c_i$  where values like  $FX(t_i, EUR, USD)$  or  $SwapRate(t_i, 10Y)$  are taken from the Monte-Carlo framework.
- Then, we can obtain the present value of the product by

$$N(0)E \left[ \sum_{i=1}^n \frac{c_i}{N(t_i)} \right]$$

- Cancel rights can be integrated using least-square regression and duality methods on the simulated cashflows  $c_i$ .

Introduction

Basic Concepts of Compiler Engineering

Using Compiler-Compilers

In-depth: Parsing LL(1) Grammars

# Grammars

- Informally, a grammar is
  - a set of rules
  - that describes how the different symbols (words) of a language
  - can be composed to a valid statement in that language.
- Grammars do not describe the semantics, i.e. the meaning.
- For example, consider the grammar for a language with symbols a, b, c and the rules:

$$S \rightarrow aSb$$

$$S \rightarrow c$$

- Then e.g.  $S \rightarrow aSb \rightarrow aaSbb \rightarrow aacbb$  is a valid statement.
- For this simple grammar one can specify all valid statements:

$c, acb, aacbb, aaacbbb, aaaacbbbbb, \dots$

# Grammars

A context-free grammar is a tuple  $G = (T, N, P, S)$  consisting of

- a finite set of *terminal symbols*  $T$ :  
elementary language components like keywords or numbers.
- a finite set of *non-terminal symbols*  $N$ :  
structured components like sentences, loops, sub-routines.  
 $\Gamma := T \cup N$  is called alphabet and  $\Gamma^*$  denotes the set of all finite sequences with values in  $\Gamma$ .
- a set of *production rules*  $P \subseteq N \times \Gamma^*$  describing how non-terminal symbols can be built from non-terminal and terminal symbols.
- the *start symbol*  $S \in N$ , a non-terminal symbol from which any valid expression can be derived, e.g. a valid text.

# Grammars

- Production rules  $(A, \alpha) \in N \times \Gamma^*$  are also denoted by  $A \rightarrow \alpha$ .
- We write  $A \rightarrow \alpha_1 | \alpha_2$  as shortcut for the two production rules  $A \rightarrow \alpha_1$  and  $A \rightarrow \alpha_2$ .
- One says that  $\beta \in \Gamma^*$  can be derived from  $\alpha \in \Gamma^*$  and writes

$$\alpha \Rightarrow \beta \quad \text{if} \quad \alpha = \alpha_1 A \alpha_3 \quad \text{and} \quad \beta = \alpha_1 \alpha_2 \alpha_3.$$

for suitable  $\alpha_1, \alpha_2, \alpha_3 \in \Gamma^*$  and a production rule  $A \rightarrow \alpha_2$ .

- For example  $aaSbb \Rightarrow aaaSbbb$ .
- We write

$$\alpha \Rightarrow^* \beta$$

if  $\beta$  can be derived from  $\alpha$  in  $n$  steps with  $n \geq 0$ .

# Grammars

- The set  $L(G) = \{\alpha \in \Gamma^* : S \Rightarrow^* \alpha\}$  is called the formal language defined by the grammar  $G$ .
- The basic question of compiler engineering is to decide efficiently whether

$$\alpha \in L(G) \quad \text{for a given} \quad \alpha \in \Gamma^*.$$

- A derivation  $S \Rightarrow^* \alpha$  is called left-most derivation if in any step the left-most non-terminal symbol is expanded first.
- Similarly, in a right-most derivation the right-most non-terminal symbol is expanded first.
- Once we have found the derivation/syntax tree it is rather easy to incorporate actions triggered by the different rules.

## A Simplified Grammar for Payoffs

- We want to specify a grammar for payoff descriptions like

$$\max(0.09 - 2 * (\text{SwapRate}(t, 10Y) - \text{SwapRate}(t, 2Y)), 0)$$

- Terminal symbols are

numbers, +, -, \*, /, (, ), max, SwapRate

- A suitable grammar is

$$S \rightarrow E$$

$$E \rightarrow T | E + T | E - T$$

$$T \rightarrow F | T * F | T / F$$

$$F \rightarrow \text{number} | \text{SwapRate}(\text{number}, \text{number}Y) | (E) | \max(E, E)$$

- Non-terminals are given by  $E$  (for expressions),  $T$  (for terms),  $F$  (for factors),  $S$  (the start-symbol)

## A Simplified Grammar for Payoffs

- For the sake of simplicity, we will work with a reduced set of rules (without  $-$ ,  $/$ , SwapRate, max)

$$P = \{1 : S \rightarrow E, 2 : E \rightarrow T, 3 : E \rightarrow E + T, 4 : T \rightarrow F, \\ 5 : T \rightarrow T * F, 6 : F \rightarrow \text{number}, 7 : F \rightarrow (E)\}$$

- Then, e.g. for  $1 + 1$  the left-most derivation is given by

$$S \Rightarrow_1 E \Rightarrow_3 E + T \Rightarrow_2 T + T \Rightarrow_4 F + T \Rightarrow_6 1 + T \Rightarrow_4 1 + F \Rightarrow_6 1 + 1$$

- Similarly, the right-most derivation is given by

$$S \Rightarrow_1 E \Rightarrow_3 E + T \Rightarrow_4 E + F \Rightarrow_6 E + 1 \Rightarrow_2 T + 1 \Rightarrow_4 F + 1 \Rightarrow_6 1 + 1$$

## LL(k) and LR(k) grammars

- When parsing a grammar containing rules of the form

$$A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$$

it is not clear which rule  $A \rightarrow \alpha_i$  has to be chosen.

- Choosing the wrong production rule will lead to a dead-end, i.e. the derivation cannot be done in this way.
- Although a systematic trial-and-error testing of all alternatives of the production rules is possible in theory (back-tracking), it is by far too time-consuming.
- A little better is the so-called CKY algorithm for context-free grammars (cf. [ASU]), but its cubic complexity is still too bad for practical usage.

## LL(k) and LR(k) grammars

- Therefore, one restricts to sub-classes of all context-free grammars that can be efficiently (linear complexity) parsed.
- *Idea:* Use additional information that allow for a deterministic choice of the correct alternative of the production rules.
- This information is given by the so-called lookahead:  
We use not only the current terminal symbol of the input stream but look ahead to the following terminal symbols.
- A grammar is called LL(k) if one can deterministically parse the left-most derivation using a lookahead of  $k$  symbols.
- Similarly, an LR(k) grammar parses a right-most derivation.
- For these subclasses there exist efficient parsing algorithms.

Introduction

Basic Concepts of Compiler Engineering

Using Compiler-Compilers

In-depth: Parsing LL(1) Grammars

# Compiler Compiler

- Since there exist algorithms for parsing LL(k) or LR(k) grammars one can implement these parsing algorithms to obtain a compiler-compiler.
- As input for a compiler-compiler one only needs a grammar.
- Then, the compiler-compiler will generate the (C or C++) source code for parsing this grammar.
- This source code can be changed and has to be compiled again with a usual C/C++ compiler to obtain the final parser.
- Usually, compiler-compilers allow for user-defined call-backs to trigger actions (like calculations) if a rule is recognized.

# Compiler Compiler

- There exist several popular compiler-compilers:
- YACC is the classic LALR(1) parser (subclass of LR(1)) that
  - has been developed further in several directions like GNU Bison
  - is capable of handling large languages
- ANTLR is somehow similar to YACC, but parses  $LL(\infty)$  grammars and generates code in C, C++, C#, Java, Python.
- SPIRIT is also an  $LL(\infty)$  parser that
  - is part of the C++ Boost Library
  - generates the compiler using template meta programming techniques without the need of two compiling processes.
  - grammars can be specified as valid C++ code
  - is only suited for small grammars like those for payoffs.

# Payoff Grammar in SPIRIT

```
struct calculator : public grammar<calculator>
{
    template <typename ScannerT> struct definition
    {
        definition(calculator const& self)
        {
            G = '(' >> E >> ')';
            F = real_p | G | M | S;
            T = F >> *(('*' >> F) | ('/' >> F));
            E = T >> *(('+' >> T) | ('-' >> T));
            M = str_p("max(") >> E >> ',' >> E >> ')';
            S = str_p("SwapRate(") >> *(alnum_p | ',' >> ')');
        }
        rule<ScannerT> E, T, F, G, M, S;
        rule<ScannerT> const& start() const { return E; }
    };
};
```

# Input for Compiler-Compiler

- When specifying a grammar one has to make sure that it satisfies the prerequisites of a given compiler-compiler.
- E.g. one has to decide if a grammar is LL(1) or LR(1).
- Often one can transform a grammar violating the prerequisites into an equivalent one satisfying the prerequisites.
- This transformation work cannot be done algorithmically.
- Therefore, the intellectual work is to specify the grammar in a clever way,
- and one needs to have at least basic theoretical knowledge on LL(k) and LR(k) grammars.

Introduction

Basic Concepts of Compiler Engineering

Using Compiler-Compilers

In-depth: Parsing LL(1) Grammars

## Some Properties of LL(1) Grammars

- One can show that a grammar with left recursions

$$A \rightarrow \alpha | A\beta$$

cannot be LL(1).

- But: One can always remove a left recursion by introducing the equivalent production rules (with the empty word  $\varepsilon$ )

$$A \rightarrow \alpha B$$

$$B \rightarrow \beta B | \varepsilon$$

- For a context-free grammar one cannot decide algorithmically if there exists an equivalent LL(1) grammar.
- Moreover, there exist context-free grammars that do not have equivalent LL(1) grammars.

# First and Follow Sets

- For  $\alpha \in \Gamma^*$  define the first set

$$\text{first}(\alpha) = \{t \in T : \alpha \Rightarrow^* t\beta \text{ with } \beta \in \Gamma^*\} \cup \{\varepsilon\} \quad \text{if } \alpha \Rightarrow^* \varepsilon$$

$$\text{first}(\alpha) = \{t \in T : \alpha \Rightarrow^* t\beta \text{ with } \beta \in \Gamma^*\} \quad \text{otherwise}$$

- For a non-terminal symbol  $A \in N$  define the follow set

$$\text{follow}(A) = \{t \in T : S \Rightarrow^* \alpha_1 A t \alpha_2 \text{ with } \alpha_1, \alpha_2 \in \Gamma^*\}$$

- For a production rule  $A \rightarrow \alpha$  define the predict set

$$\text{predict}(A \rightarrow \alpha) = \text{first}(\alpha \text{follow}(A))$$

- A grammar is LL(1) if and only if for any rule

$$A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$$

we have  $\text{predict}(\alpha_i) \cap \text{predict}(\alpha_j) = \emptyset$  for  $i \neq j$ .

# LL(1) parsing of arithmetic expressions

- The reduced grammar

$$P = \{1 : S \rightarrow E, 2 : E \rightarrow T, 3 : E \rightarrow E + T, 4 : T \rightarrow F, \\ 5 : T \rightarrow T * F, 6 : F \rightarrow \textit{number}, 7 : F \rightarrow (E)\}$$

contains left recursions and is not LL(1).

- But we can replace the grammar by the equivalent grammar

$$P = \{1 : S \rightarrow E, 2 : E \rightarrow TE', 3 : E' \rightarrow +TE', 4 : E' \rightarrow \varepsilon \\ 5 : T \rightarrow FT', 6 : T' \rightarrow *FT', 7 : T' \rightarrow \varepsilon \\ 8 : F \rightarrow (E), 9 : F \rightarrow \textit{number}\}$$

## LL(1) parsing of arithmetic expressions

- The predict sets can be easily calculated:

$A \rightarrow \alpha$	$\text{predict}(A \rightarrow \alpha)$	semantic action
$S \rightarrow E$	$\{number, (\}$	
$E \rightarrow TE'$	$\{number, (\}$	
$E' \rightarrow +TE'$	$\{+\}$	add
$E' \rightarrow \varepsilon$	$\{\varepsilon, )\}$	
$T \rightarrow FT'$	$\{number, (\}$	
$T' \rightarrow *FT'$	$\{*\}$	mul
$T' \rightarrow \varepsilon$	$\{\varepsilon, +, )\}$	
$F \rightarrow (E)$	$\{( \}$	
$F \rightarrow number$	$\{number\}$	number

- This shows, that the reformulated grammar is LL(1).

# LL(1) parsing of arithmetic expressions

- This leads to the parsing table

	<i>number</i>	( )	+	*	$\epsilon$
<i>S</i>	<i>E</i>	<i>E</i>			
<i>E</i>	<i>TE'</i>	<i>TE'</i>			
<i>E'</i>			$\epsilon$	<i>TE'add</i>	$\epsilon$
<i>T</i>	<i>FT'</i>	<i>FT'</i>			
<i>T'</i>			$\epsilon$	$\epsilon$	<i>FT'mul</i>
<i>F</i>	<i>number</i>	( <i>E</i> )			

- The grammar can be parsed with a stack and an input queue:
  - One starts with the start symbol on the stack.
  - Depending on the next terminal symbol in the input queue the top non-terminal symbol on the stack is replaced by the symbols given in the table.
  - The input is recognized if both the stack and the input queue are empty.
  - If we don't find a substitution in the table, we have an error.

# LL(1) parsing of arithmetic expressions

For illustration we use the parsing table

	<i>number</i>	<i>( )</i>	<i>+</i>	<i>*</i>	$\epsilon$
<i>S</i>	<i>E</i>	<i>E</i>			
<i>E</i>	<i>TE'</i>	<i>TE'</i>			
<i>E'</i>			$\epsilon$	<i>TE'add</i>	$\epsilon$
<i>T</i>	<i>FT'</i>	<i>FT'</i>			
<i>T'</i>			$\epsilon$	$\epsilon$	<i>FT'mul</i>
<i>F</i>	<i>number</i>	<i>(E)</i>			

to parse the expression  $1 + 2 * 3$ :

$[1 + 2 * 3, S, .] \rightarrow [1 + 2 * 3, E, .] \rightarrow [1 + 2 * 3, TE', .] \rightarrow$   
 $[1 + 2 * 3, FT'E', .] \rightarrow [+2 * 3, T'E', 1] \rightarrow [+2 * 3, E', 1] \rightarrow$   
 $[2 * 3, TE'add, 1] \rightarrow [2 * 3, FT'E'add, 1] \rightarrow [*3, T'E'add, 21] \rightarrow$   
 $[3, FT'mulE'add, 21] \rightarrow [., T'mulE'add, 321] \rightarrow [., mulE'add, 321] \rightarrow$   
 $[., E'add, 61] \rightarrow [., add, 61] \rightarrow [., ., 7]$

# References

- [A] ANTLR: <http://www.antlr.org>.
- [ASU] A. Aho, R. Sethi, J. Ullman.  
Compilers - Principles, Techniques, and Tools  
Addison-Wesley, 1986
- [OC] O. Caps.  
On the Valuation of Power-Reverse Duals and Equity-Rates  
Hybrids.  
<http://conference.mathfinance.com/2007/abstracts.php>
- [G] GNU Bison: <http://www.gnu.org/software/bison>.
- [S] Spirit: <http://spirit.sourceforge.net>.