Writing a Simple DSL Compiler with Delphi
About me

- Primož Gabrijelčič
  - http://primoz.gabrijelcic.org
  - programmer, MVP, writer, blogger, consultant, speaker
  - Blog http://thedelphigeek.com
  - Twitter @thedelphigeek
  - Skype gabr42
  - LinkedIn gabr42
  - GitHub gabr42
  - SO gabr
  - Google+ Primož Gabrijelčič
WHY AM I HERE?
It all had started with a podcast ...

https://hanselminutes.com
https://interpreterbook.com
DSL
DSL?

- Damn Small Linux
- Danish Sign Language
- Dictionary of the Scots Language
- Dominican Summer League
- Domestic Substances List
- Domain Specific Language
  - A (computer) language designed for a specific problem domain
  - In short … a programming language

https://en.wikipedia.org/wiki/DSL_(disambiguation)
When?

- When presenting a special syntax helps certain class of users
- Most popular DSLs: SQL, html, LaTeX, BNF, VHDL
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  ( aclr : in std_logic;
    clk : in std_logic;
    a : in std_logic_vector;
    b : in std_logic_vector;
    q : out std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high-1 downto 0); -- extra bit wide
begin
  begin -- architecture
    assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
    q <= std_logic_vector(q_s);
    adding_proc:
      process (aclr, clk)
      begin
        if (aclr = '1') then
          q_s <= (others => '0');
        elsif rising_edge(clk) then
          q_s <= ('0'&signed(a)) + ('0'&signed(b));
        end if; -- clk'd
      end process;
  end signed_adder_arch;

Source: https://en.wikipedia.org/wiki/VHDL#/media/File:Vhdl_signed_adder_source.svg
FROM PROGRAM TO RESULT
From Program to Result

• Program = stream of characters

• Parsing
  • Lexical analysis [lexer/tokenizer]
    • Characters → tokens
    • Defined by regular expressions
  • Syntactical analysis [parser]
    • Tokens → internal representation [AST]
    • Defined by a grammar

• Execution
  • Interpreter: Walk over an AST + execute step by step
  • Cross-compiler: Walk over an AST + rewrite it as an equivalent textual output
  • Compiler: Walk over an AST + generate machine code (for some architecture)
  • [semantical analysis]
From program to result
Abstract Syntax Tree (AST)

- An abstract syntactic structure in a tree form
- Inessential stuff is removed
  - Punctuation
  - delimiters
- Can contain extra information
  - position in source code
- Specific for a single language

https://en.wikipedia.org/wiki/Abstract_syntax_tree
while \( b \neq 0 \)

if \( a > b \)
  \( a := a - b \)
else
  \( b := b - a \)

return \( a \)

DelphiAST

- [https://github.com/RomanYankovsky/DelphiAST](https://github.com/RomanYankovsky/DelphiAST)
  - One unit a a time
- [https://github.com/gabr42/DelphiAST](https://github.com/gabr42/DelphiAST)
  - Project indexer
- [https://github.com/gabr42/DelphiLens](https://github.com/gabr42/DelphiLens)
  - Research project
GRAMMARS FOR DUMMIES
Grammar

• Set of production rules
  • Left hand side → right hand side

• Symbols
  • Nonterminal [can be expanded]
  • Terminal [stays as it is]
  • Start

• Can be recursive or non-recursive
  • Non-recursive → not interesting

https://en.wikipedia.org/wiki/Recursive_grammar
Grammar Example

- Example
  - Terminals: \{a,b\}
  - Nonterminals: \{S, A, B\}
  - Rules:
    - S -> AB
    - S -> \(\varepsilon\)
    - A -> aS
    - B -> b

- Language
  - \(a^n b^n\)

- Example
  - S -> AB -> aSB -> aSb -> aAb -> aAb -> a\(\varepsilon\)
    -> aaSbb -> a\(\varepsilon\)bb

- Simpler version
  - S -> aSb
  - S -> \(\varepsilon\)

- Example
  - S -> aSb -> aaSbb -> a\(\varepsilon\)bb

https://en.wikipedia.org/wiki/Formal_grammar
## Chomsky Hierarchy

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Languages</th>
<th>Automaton</th>
<th>Production rules (constraints)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-0</td>
<td>Recursively enumerable</td>
<td>Turing machine</td>
<td>$\alpha \rightarrow \beta$ (no restrictions)</td>
</tr>
<tr>
<td>Type-1</td>
<td>Context-sensitive</td>
<td>Linear-bounded non-deterministic Turing machine</td>
<td>$\alpha A \beta \rightarrow \alpha \gamma \beta$</td>
</tr>
<tr>
<td>Type-2</td>
<td>Context-free</td>
<td>Non-deterministic pushdown automaton</td>
<td>$A \rightarrow \gamma$</td>
</tr>
<tr>
<td>Type-3</td>
<td>Regular</td>
<td>Finite state automaton</td>
<td>$A \rightarrow a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$A \rightarrow aB$</td>
</tr>
</tbody>
</table>
Context-free Grammars

- Base of program language design
  - Typically cannot satisfy all needs
    - Indentation-based languages
    - Macro- and template-based languages
  - Attribute grammar
  - Compiler = definition

https://en.wikipedia.org/wiki/Attribute_grammar
Syntax vs. semantics

• Not all syntactically correct programs compile!
  • Most of them don’t!

program Test;
begin
  a := 1;
end.

• Set of **syntactically** correct programs = CFG (possibly)
• Set of **semantically** correct programs ≠ CFG (= CSG)
Documenting the grammar

- Backus-Naur form (BNF)
- Extended Backus-Naur form (EBNF)

Example – Pascal-like Language

```plaintext
program = 'PROGRAM', white space, identifier, white space,
    'BEGIN', white space,
    { assignment, ';', white space },
    'END.';
identifier = alphabetic character, { alphabetic character | digit };
number = [ '-' ], digit, { digit };
string = '"', { all characters - '"' }, '"';
assignment = identifier, ':=', ( number | identifier | string );

alphabetic character = "A" | "B" | "C" | "D" | "E" | "F" | "G"
    | "H" | "I" | "J" | "K" | "L" | "M" | "N"
    | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
    | "V" | "W" | "X" | "Y" | "Z";
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
white space = ? white space characters ?;
all characters = ? all visible characters ?;
```
Example – Delphi 5 EBNF (partial)

```
start= program | unit | library | package .

identifier_list= ID_NAME { ',' ID_NAME }.
unit_qualified_identifier= ID_NAME { '::' ID_NAME }.

type_name= TYPE_NAME | STRING | FILE .
unit_qualified_type_name= type_name [ '::' type_name ].

function_result_type= type_name .
constant_expression= F .
string_expression= ( STRING_NAME | STRING_LITTERAL )
{ '+' ( STRING_NAME | STRING_LITTERAL ) }.

variable_access= ( ACCESS_NAME | STRING ) { end_access_ }.
end_access_ = { array_access_ | record_access_ | '^' | function_parameters_ }.
array_access_ = '[' constant_expression { ',' constant_expression } ']' .
record_access_ = '.' variable_access .
function_parameters_ = '(' [ constant_expression { ',' constant_expression } ] ')' .

set_factor= '[' [ set_element { ',' set_element } ] ']' .
set_element= constant_expression [ '::' constant_expression ] .
constant_expression= simple_expression
{ '=' | '<' | '<=' | '>' | '>=' | 'IN' }.
```

Source: http://www.felix-colibri.com/papers/compilers/delphi_5_grammar/delphi_5_grammar.html
PARSING
Parsing in Practice

- **Lexer**
  - Typically DFA (regular expressions)
  - Generator
  - Custom

- **Parser**
  - Typically LR(0), LR(1), LALR(1), LL(k)
    - Lx  top-to-bottom
    - xL  Leftmost derivation
    - xR  Rightmost derivation
    - (n) lookahead
    - LALR Look-Ahead LR, a special version of LR parser
  - Generator
  - Custom

https://en.wikipedia.org/wiki/Lexical_analysis
https://en.wikipedia.org/wiki/Parsing#Computer_languages
**LL / LR**

<table>
<thead>
<tr>
<th>Leftmost</th>
<th>Rightmost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $S \rightarrow S + S$</td>
<td>Input: $1 + 1 + a$</td>
</tr>
<tr>
<td>2. $S \rightarrow 1$</td>
<td>$S \rightarrow S + S$</td>
</tr>
<tr>
<td>3. $S \rightarrow a$</td>
<td>$\rightarrow S + a$</td>
</tr>
</tbody>
</table>

- $S \rightarrow S + S$ (1)
  - $\rightarrow 1 + S$ (2)
  - $\rightarrow 1 + S + S$ (1)
  - $\rightarrow 1 + 1 + S$ (2)
  - $\rightarrow 1 + 1 + a$ (3)

- $S \rightarrow S + a$ (3)
  - $\rightarrow S + S + a$ (1)
  - $\rightarrow S + 1 + a$ (2)
  - $\rightarrow 1 + 1 + a$ (2)

---

https://en.wikipedia.org/wiki/LR_parser
https://en.wikipedia.org/wiki/LL_parser
https://en.wikipedia.org/wiki/LALR_parser
A SIMPLE PRIMER
A Simple Primer

- Language
  - Addition of non-negative numbers
    - 1
    - 1 + 2
    - 1 + 2 + 44 + 17 + 1 + 0
- AST
- Tokenizer
- Parser
- Interpreter
- Compiler
MY “TOY LANGUAGE”
fib(i) {
    if i < 3 {
        return 1
    } else {
        return fib(i-2) + fib(i-1)
    }
}
Specification

- C-style language
- Spacing is ignored
- One data type - integer
- Three operators: +, -, and <
  - $a < b$ returns 1 if $a$ is smaller than $b$, 0 otherwise
- Two statements - if and return
  - If statement executes then block if the test expression is not 0. Else block is required
  - Return statement just sets a return value and doesn't interrupt the control flow
- There is no assignment
- Every function returns an integer
- Parameters are always passed by value
- A function without a return statement returns 0
- A function can call other functions (or recursively itself)
function ::= identifier "(" [ identifier { "," identifier } ] ")" block

block ::= "{" statement ";" statement ";" ] "}"

statement ::= if | return

if ::= "if" expression block "else" block

return ::= "return" expression

expression ::= term | term operator term

term ::= numeric_constant | function_call | identifier

operator ::= "+" | "-" | "<"

function_call ::= identifier "(" [expression { "," expression } ] ")"
EXTENDING THE LANGUAGE
Attributes

- AST
- Tokenizer
- Parser
- Interpreter
- Compiler

```
fib(i) [memo] {
    if i < 3 {
        return 1
    } else {
        return fib(i-2) + fib(i-1)
    }
}
```
THANK YOU!