Textual Syntax[1]

Introduction

Dr Artur Boronat

[1] Materials in these slides are borrowed from [1]

Do we need concrete syntax?

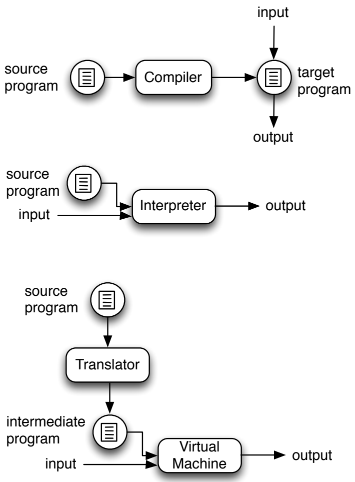
- A metamodel defines the abstract syntax for a language
- The concrete syntax is the "UI" for the language and is critical for DSL users
    - concise vs redundant
    - intuitive
    - simple to write and read
- Tool support matters
    - IDE integration
    - syntax highlighting
    - metamodel awareness

Strengths of Textual DSLs

- Textual languages have specific strengths compared to graphical languages
  - ideally there should be the option to have both
- Compact and expressive syntax
  - productivity for experienced users
  - IDE support softens users' learning curve
- Configuration management/versioning and integration into the "usual" development process
  - splitting a model into several files
  - concurrent work on a model, especially with a version control system: diff, merge

Approaches

- Compiler: program that reads a program in one (source) language and translates it into an equivalent program in a (target) language
- Interpreter: directly executes the operations specified in the source program on inputs supplied by the user
- Hybrid compiler: usually based on a virtual machine (e.g. Java)



Outline of the lecture

1. Compiler theory notions
    - Context-free grammar
    - Derivations
    - ASTs
    - Languages
    - Grammar design
2. Xtext Grammars
    - EBNF vs Ecore (EMF)
    - Xtext
    - Bridging Xtext (grammars) and EMF (metamodels)

Context-Free Grammars

- Concrete syntax: program representation, including lexical details such as the placement of keywords and punctuation marks
- Concepts:
  - Syntactic category or object symbol <E> (aka non-terminal): represents a set of strings being defined
  - Terminal symbol:
    - Characters (e.g. '+' or '(') or keywords (e.g. if or while)
    - Abstract symbols, such as 'id', 'string', 'number': placeholder for any string that can be defined as an 'id', 'string' or 'number' respectively
  - Production rule: H → B
    - H(ead): syntactic category that is (partially defined) by the rule
    - → means consists of
    - B(ody): string formed by zero or more terminals and non-terminals
  - Start symbol: syntactic category that represents the language being defined
- A context-free grammar is a tuple (N,T,P,S) where:
  - N is a set of non-terminal symbols
  - T is a set of terminal symbols
  - P is a set of production rules
  - S is the start symbol or sentential symbol of the grammar

Example

- A grammar to represent lists of digits separated by plus or minus signs

```
list -> list + digit
list -> list - digit
list -> digit
digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

  or

```
list -> list + digit | list - digit | digit
digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- What are the terminals of the grammar?
  + - 0 1 2 3 4 5 6 7 8 9

Notational Conventions

- Terminals
  - Lowercase letters early in the alphabet: a, b, c ...
  - Operator symbols: +, *, ...
  - Punctuation symbols: parentheses, comma, ...
  - The digits: 0,1,...,9.
  - strings in between quotes such as 'id' or 'if'
- Nonterminals
  - Uppercase letters early in the alphabet: A, B, C
  - The letter S for the start symbol
  - Lowercase names such expr or stmt
- Uppercase letters late in the alphabet, such as X, Y, ... Z, represent grammar symbols, i.e. either nonterminals or terminals.
- Lowercase Greek letters, such as α, β, γ, represent (possibly empty) strings of grammar symbols.
  - For example, a generic production can be written $A \to \alpha$
- Lowercase letters late in the alphabet, chiefly u,v,... z, represent (possibly empty) strings of terminals.

Empty string and recursive rules

- Define a grammar that captures the list of parameters in a function call

```
call -> id ( optparams )
optparams -> params | ϵ
params -> params , param | param
```

  - ϵ stands for the empty string of symbols
  - `optparams` can be replaced with the empty string
- Production rules can be recursive

```
stmt -> if ( expr ) stmt else stmt
```

- Define a grammar for palindromes: 0, 1, 00, 11, 0110, 10101, 0101111010

What do you want to produce?

Recursive inference

- Infer if a string is in the language of a given non-terminal.
- Parsing: the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar.

Derivation

- Expand the start symbol using one of its productions.
  - Usually the start symbol is defined by the first production rule
- Language generation: a grammar derives strings by beginning with the start symbol and by repeatedly replacing a nonterminal with the body of a production for that nonterminal.

Derivations in a CFG

- Given a CFG $G = (N, T, P, S)$, we use productions in G from head to body starting with the start symbol of the grammar.
- The language of the grammar is all the strings that we can generate in this way.
- One derivation step:
  - mathematically[2]: Let $\alpha A \beta$ be a string of terminals and variables such that $A \in N$ and $\alpha, \beta \in (N \cup T)^{\star}$. Let $A \rightarrow \gamma$ be a production of G. Then we say $\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta$.
  - idea: one derivation step replaces any variable anywhere in the string by the body of one of its productions.
- The relationship $\underset{G}{\Rightarrow}$ can be extended to represent zero, one, or many derivation steps, denoted $\underset{G}{\Rightarrow}^{*}$.
- Ex: $P \underset{G}{\Rightarrow}^{*} 01110 = P \underset{G}{\Rightarrow} 0P0 \underset{G}{\Rightarrow} 01P10 \underset{G}{\Rightarrow} 01110$

[2] _* corresponds to the Kleene star
http://en.wikipedia.org/wiki/Kleene_star

Leftmost and Rightmost Derivations

- To reduce the number of choices we have in deriving a string using a CFG:
  - Leftmost derivation: at each step we replace the leftmost variable by one of its production bodies
  - Rightmost derivation: at each step we replace the rightmost variable by one of its production bodies
- Ex: given the grammar $E \rightarrow E\,`+'\,E | E\,`*'\,E | `-'\,E | (E) | id$, define the leftmost derivation for the expression $-(id + id)$:
  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)$.
  and its rightmost derivation:
  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + id) \Rightarrow -(id + id)$.
- A sentential form α of the grammar is a string of terminals and nonterminals or ϵ.
- A sentence of G is a sentential form that has no nonterminals.

Parse trees

- Data structure of choice to represent a source program in a compiler.
- The parse trees for a grammar $G = (N, T, P, S)$ are trees with the following conditions:
  1. Each interior node is labeled by a nonterminal in N.
  2. Each leaf is labeled by either a nonterminal, a terminal, or ϵ. If the leaf is labeled ϵ, then it must be the only child of its parent.
  3. If an interior node is labeled A, and its children are labeled

     $$X_1, X_2, \ldots, X_k$$

     respectively, from the left, then $A \rightarrow X_1 X_2 \ldots X_k$ is a production in P.
- The yield of a parse tree, which is a string formed by the leaves of the tree from left to right. Parse trees whose yield is always a sentence are of great importance.
- The root node of a parse tree is the start symbol of the grammar.
- The set of yields of parse trees of a grammar form the language of the grammar.

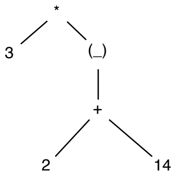
Abstract Syntax Trees

- Parse trees are very detailed: every step in a derivation is a node.
  - interior nodes represent nonterminals
- A semantic analyzer removes intermediate productions to create an (abstract) syntax tree.
  - interior nodes represent programming constructs
- Transformation:
  1. Atomic operands are condensed to a single node labelled by that operand.
  2. Operators are moved from leaves to their parent node.
  3. Interior nodes that remain labelled by a syntactic category have their label removed.

Abstract Syntax Trees

- Grammar:

```
number -> number digit
expr -> number | ( expr ) | expr ‘+’ expr |  expr ‘*’ expr
digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```



- Expression: 3 * (2+14)

Languages from Grammars

A grammar is an inductive definition involving sets of strings.
For each nonterminal N of a grammar we define a language $L(N)$:

- Basis case: assume that for each N, $L(N)$ is empty.
- Induction case:
  - Suppose the grammar has a production $N \rightarrow X_1 X_2 \ldots X_n$
  - For each X_i select a string s_i for X_i as follows:
    - If X_i is a terminal, then we may only use X_i as the string s_i
    - If X_i is a nonterminal, then select any string s_i that is known to be in $L(X_i)$. When there are several X_i's, we can pick a different string from $L(X_i)$ for each occurrence.
- Then the concatenation $s_1 s_2 \ldots s_n$ of those selected strings is a string in the language $L(N)$. If $n = 0$, then we put ϵ in the language.

Language for a Grammar

```
S -> while cond S | { L } | s;
L -> L S | ϵ
```

Define the language generated by this grammar:

	S	L
round 1	s;	ϵ
round 2	while cond s; { }	s;
round 3	while cond while cond s; while cond {} { s; }	while cond s; { } s; s; s; while cond s; s; {}

EBNF

- Problems with Backus-Naur Form:
  - It is too long.
  - We must use recursion to specify repeated occurrences
  - We must use separate alternatives for every option
- Extended BNF:
  - The standard metalanguage: Extended BNF
  - Terminal symbols of the language are quoted
  - = is the defining symbol
  - , denotes concatenation
  - [and] denote optional symbols
  - { and } denote repetition
  - (and) are used to group items.
    - In EBNF, we need to quote (and) literals as '(' ... ')'.
  - | denotes alternatives.
  - ; termination symbol

Grammar Design

- BNF:

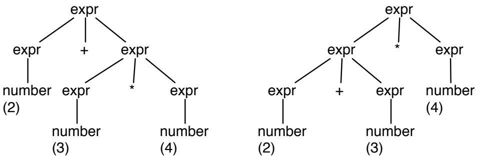
```
expr -> expr '+' expr | expr '-' expr | term
term -> term '*' factor | term '/' factor | factor
factor -> (expr) | id | number
```

- EBNF:

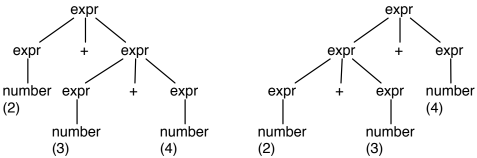
```
expr = term , { ('+' | '-') , term } ;
term = factor , { ('*' | '/') , term } ;
factor = '(' , expr , ')' | id | number ;
```

Ambiguous Grammars

- A grammar is ambiguous if there is more than one parse tree, whose root is labelled with the same symbol, for a valid sentence.
- Given the following production rules:
  $expr \rightarrow expr\,`+'\,expr | expr\,`*'\,expr | id | number$
    - How would you parse $2 + 3 * 4$?



    - How would you parse $2 + 3 + 4$?



Ambiguous Grammars: conflict resolution

- Replace multiple occurrences of the same nonterminal with a different nonterminal to enforce precedence
- Choose a replacement that gives correct associativity: $expr \rightarrow expr \text{ `+' } term$
- Add new rules in order to achieve the correct precedence:

```
expr -> expr '+' term | term
term -> term '*' factor | factor
factor -> ( expr ) | id | number
```

Writing a Grammar

1. Avoid ambiguities
2. Avoid left recursion
   - A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \Rightarrow A\alpha$ for some string α
   - Tools like Xtext do not admit left-recursive grammars
   - Technique to eliminate left-recursion
     - Group productions as $A ::= A\alpha_1 | \ldots | A\alpha_n | \beta_1 | \ldots | \beta_n$
     - Replace A-productions

$$A ::= \beta_1 A' | \ldots | \beta_n A'$$

$$A' ::= \alpha_1 A' | \ldots | \alpha_n A' | \epsilon$$

3. Apply left factoring
   - To remove ambiguities when deciding which rule to apply using a predictive parsing method
   - Ex: $A ::= \alpha\beta_1 | \alpha\beta_2$; could be left factored to $A ::= \alpha A'; A' ::= \beta_1 | \beta_2;$

What's next?

- EBNF vs Ecore
- Xtext
- Bridging Grammars and Metamodels

References

R. Sethi A. V. Aho, M. S. Lam and J. D. Ullman.
Compilers: Principles, Techniques, and Tools.
Pearson Education, Inc., 2nd edition, 2007.