Syntax of Programming Languages

Reading:

- Sebesta, sections 3.1–3.4, sections 4.3–4.5

©Diane Horton 2000;
Modified by Sheila McIlraith 2004.

Translation Process Summary

1. Lexical Analysis:
   Converts source code into sequence of tokens

2. Syntactic Analysis:
   Structures tokens into initial parse tree

3. Semantic Analysis:
   Annotates parse tree with *semantic actions*

4. Code Generation:
   Produces final machine code

What is a Programming Language?

We tend to think of a *compiler* or an *IDE* as a programming language.
E.g., JDF, Java Workshop.

But these things are not Java. The language is an abstract entity, which these pieces of software implement.

Specification:
vs
Implementation:

Formal notion of a "language": a set of strings of symbols from some alphabet.

Language Specification

Two parts: syntax and semantics.

Syntax

Definition*: (1) The way in which words are put together to form phrases and sentences. (2) Analysis of the grammatical arrangement of words, to show their relation.
Root: means "arrange".

The syntax of a language tells us two things: what's legal, and what the relationships are in a legal sentence.

Example of relationships:
"used kids clothing store"

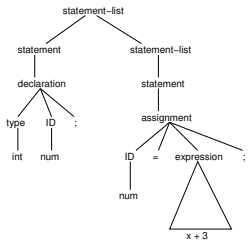
*Definitions are paraphrased from Webster's and the OED.

For a programming language, the units are not words but "tokens". Example:

```
int num;
num = x + 3;
```

Tokens:

Structure:



Specifying syntax informally

Example: "Everything between "/*" and "*/" is a comment and should be ignored."

Code:

```
/* Do such and such, watching out for problem fleep.
   Store the result in y.  */
x = 3;  */
y = x * 17.2;
```

When syntax is defined informally, incompatible dialects of the language may evolve.

Specifying syntax formally

The state of the art is to define programming language syntax formally.

There are a number of well-understood formalisms for doing so.

We'll talk about this in some detail.

Semantics

Definition: The study or science of meaning in language forms. Root: means "signify".

The semantics of a language defines the meaning of the legal sentences of the language.

Specifying semantics informally

Example: *The Java Language Specification* by Gosling, Joy, and Steele, page 93:
"The meaning of a name classified as a *PackageName* is determined as follows:
(1) If the package name consists of a single *Identifier*, then this identifier denotes a top-level package named by that identifier. If no packages of that name is accessible, then a compile-time error occurs.
(2) If a package name is of the form *Q.Id*, then ..."

Problems with informal specification of semantics?

Unfortunately

Defining semantics is inherently harder than defining syntax.

There are several formalisms for specifying programming language semantics (see Sebesta section 3.5), but they are hard to use and have not been widely adopted.

The state of the art is to define programming language semantics informally, in English.

Intended Audience

A language specification is written for three categories of people:

- Implementers,
  i.e., programmers writing a compiler for that language.
- Users,
  i.e., programmers writing in that language.
- Potential future users,
  during development of the language.

Want: What properties do we want a good language specification to have?

Specifying PL syntax

Two parts: Lexical rules, and syntax.

Lexical rules

Specify the form of the building blocks of the language:

- what's a token
- how tokens are delimited
- where can white space go
- syntax of comments

This is often described informally, in English.

Trickier parts (*e.g.*, syntax of real numbers) are sometimes described more formally.

Syntax

Specifies how to put the building blocks together.

Grammars

Informal idea of grammar: A bunch of rules.

- Don't end a sentence with a preposition.
- Subject and verb must agree in number.

A Formal grammar is a different concept.

A "language" is a set of strings; A grammar "generates" a language — it specifies which strings are in the language.

A grammar can be used to define *any* language: Java, Spanish, Unix commands.

There are many kinds of formal grammar.

Chomsky's Hierarchy

There are several categories of grammar, ordered by expressiveness (the last one is the least expressive):

- Phrase-Structure Grammars
- Context-Sensitive Grammars
- Context-Free Grammars
- Regular Grammars (can be described by regular expressions)

This hierarchy (circa 1950) is named after linguist (and political activist) Noam Chomsky, who researched grammars for natural language.

Regular Expressions

Kleene's language definition for Regular Languages.

Examples:

- $(0+1)^*$
- $1^+(:+;)^*$
- $(a+b)^*aa(a+b)^*$

Notation:

Kleene Closure: * superscript denotes 0 or more repetitions

Positive Closure: $^+$ superscript denotes 1 or more repetitions

Alternation: binary "+" denotes choice. It is also denoted by |, i.e., $(0|1)^*$.

"(" and ")" are used for grouping

ϵ (epsilon) denotes the empty or "null" string.

$\emptyset$ denotes the language with *no* strings.

Give regular expressions for these languages:

1. All alphanumeric strings beginning with an upper-case letter.
2. All strings of a's and b's in which the third-last character is b.
3. All strings of 0's and 1's in which every pair of adjacent 0's appears before any pairs of adjacent 1's.
4. All the binary numbers with exactly six 1's.
5. What is another way of writing $0^+1^+2^+$

Limitations of Regular Expressions

Regular expressions are not powerful enough to describe some languages.

Examples:

- The language consisting of all strings of one or more a's followed by the same number of b's.
- The language consisting of strings containing a's, left brackets, and right brackets, such that the brackets match.

Research question: How can we be sure there is no regular expression for these languages?

Research question: Exactly what things can and cannot be expressed with a regular expression?

Context-Free Grammar

CFGs are more powerful than regular expressions.

Definition

A CFG has four parts:

- A set of tokens (or "terminals"):
  The atomic symbols of the language.
- A set of "non-terminals":
  Variables used in the grammar.
- A special non-terminal chosen as the "starting non-terminal" or "start symbol":
  It represents the top-level construct of the language.
- A set of rules (or "productions"), each specifying one legal way that a non-terminal could be constructed from a sequence of tokens and non-terminals.

Example

A CFG for real numbers:

- Terminals: `0 1 2 3 4 5 6 7 8 9 .`
- Non-terminals: real-number, part, digit.
- Productions:
  - A digit is any single token except ".".
  - A part is a digit.
  - A part is a digit followed by a part.
  - A real-number is a part, followed by ".", followed by a part.
- Start symbol: real-number.

Note that we use recursion to specify repeated occurrences.

We have defined this CFG using plain English. A notation might be more convenient.

Backus-Naur Form

A notation for writing down a CFG.

Example

```
<real-number> --> <part> . <part>
<part>        --> <digit> | <digit> <part>
<digit>       --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Notation

- Productions: Non-terminal, followed by "-->", then the list of tokens and non-terminals that it can be made of, without punctuation.
- Terminals: Just written within the rules.
- Non-terminals: enclosed with "<" and ">". (`<empty>` denotes the empty string.)
- Start symbol: Usually just the first non-terminal listed.

Note that this is a language for describing a language! We call this a "meta-language". ("meta" meaning "above" or "transcending".)

Write a CFG for each of the 3 languages we wrote regular expressions for a few slides ago.

More Examples

Write a CFG for each of these languages:

1. all non-empty strings containing only a's.
2. all strings of odd length containing only a's.
3. all strings of one or more a's followed by one more more b's.

CFGs Are More "Powerful" Than REs

That is, there are languages that cannot be described with a RE but can be described with a CFG.

Example: The language consisting of strings with one or more a's followed by the same number of b's.

There is no regular expression for this language.

CFG for the language:

Regular vs. Context-Free Languages

Regular languages are simpler than programming languages (e.g., numbers, identifiers).

- Context-free grammars can describe nested constructs, matching pairs of items.
- Regular grammars can only describe linear, not nested, structure.

Regular Grammars

Defined over alphabet Σ, using non-terminals and grammar rules, analogous to terminals (words), and production rules of Context-Free Grammars (which we'll see later), but **more restricted**. They are limited to productions of the form:

Left-recursive:

```
<N> ::= <X> a b
<X> ::= a | <X> b
```

Right-recursive:

```
N ::= b | b <Y>
Y ::= a b | a b <Y>
```


Extended BNF

There are extensions to BNF that make it more concise, but no more powerful (*i.e.*, there is no language that can be expressed with EBNF but not with BNF).

Examples:

- { *blah* } denotes zero or more repetitions of *blah*.
- [*blah*] denotes that *blah* is optional.
- a + superscript denotes one or more repetitions.
- a numeric superscript denotes a maximum number of repetitions.
- (and) are used for grouping.

There is no one standard EBNF; it just refers to any extension of BNF.

EBNF is more concise than BNF.

Example (Sebesta, p. 121)

BNF grammar:

```
<expr> --> <expr> + <term>  |
           <expr> - <term>  |
           <term>
<term> --> <term> * <factor>  |
           <term> / <factor>  |
           <factor>
```

EBNF grammar for the same language:

```
<expr> --> <term> { (+|-) <term> }
<term> --> <factor> { (*|/) <factor> }
```

Derivations

Example:

Definition: Beginning with the start symbol, apply rules until there are only terminals left.

A sentence is in the language generated by a grammar iff there is a derivation for it.

Parse Trees

Parse trees show the structure within a sentence of the language.

Example

Grammar:

```
<real-number> --> <part> . <part>
<part>        --> <digit> | <digit> <part>
<digit>       --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Parse tree for the sentence "97.123":

9 7 . 1 2 3

Definitions

Parse tree: A tree in which

- the root is the start symbol;
- every leaf is a terminal; and
- every internal node is a non-terminal, and its children correspond, in order, to the RHS of one of its productions in the grammar.

Parsing: The process of producing a parse tree.

A sentence is in the language generated by a grammar iff there is a parse tree for the sentence.