1

Syntax of Programming Languages

©1993 John Mylopoulos

2

Syntax of Programming Languages

How do we decide whether a symbolic expression is syntactically correct with respect to a given programming language?

For example,

X/Y is syntactically correct in PLs supporting arithmetic expressions

X/-+Y is not

❒ Need a formal notation for defining the syntax of a programming language

Use notion of grammar from Linguistics

❒ To define a grammar, we need to define a character set (alphabet) and a set of tokens or terminal symbols (vocabulary)

©1993 John Mylopoulos

3

Character Sets

The alphabet of a programming language consists of characters.

For example, APL has a character set full of special operators. (any terminal can be an APL terminal with the right key-caps and character generator, as long as the computer knows the character mapping).

There are two commonly accepted character sets:

ASCII	128 characters, 96 visible, others have special meaning needs 7 bits; **UNIX uses this**
EBCDIC	256 characters, used mainly by IBM, 8 bits

Extended Character Sets contain many useful character such asf mathematical symbols, accented characters etc.

Internationally Kanji character sets use 16 bit character set.

©1993 John Mylopoulos

4

Character Sets (continued)

The core character set includes upper, lower case Roman characters and ten Arabic digits. Old systems (card punches) didn't have lower-case characters. Some language implementations today don't internally distinguish between upper and lower case, or don't permit it. (IBM JCL, Assembler, external names)

Some languages (Pascal) used bold to indicate keywords, variables in Roman. Language implementors resolved this by not permitting keywords as variable names.

Algol 60 for example was specified with a character set incompatible with ASCII. For example, implementations of the character:

≠

is variously implemented as:

not=
/=
!=
<>
=/=
=|=

©1993 John Mylopoulos

5

Tokens - the Words of the Language

The vocabulary of a programming language is the set of words or tokens, that make it up. Tokens can be divided into classes.

• Operators or Delimiters -- These consist of tokens such as:

```
+    ,    =    .    :    ;
(    )    [    ]    /    *
{    }    -    |    &    $    @
```

These non-alphanumeric tokens have a special unambiguous role in the language as operators, or deliminaters. Sometimes, two or three together also have a special meaning. For example:

```
<>    :-   ->  =/=
```

Keywords

Most languages have words made of alphabetic characters, that are special to the language. For example:

var, begin, end, if, then, else

©1993 John Mylopoulos

6

Variables and Constants

This class of tokens is "made up" by the programmer. Instances of this class are like the names of people in natural languages.

Constant strings present some problems since they use up one of the characters to delimit the start and end of the string:

```
"this  ""  string contains one double quote"
'this " string contains one double quote"
"this ' string contains one single quote"
```

Numbers also can present problems in languages where the period represents an operator. For example:

(3.4)

may be a list consisting of one number, or a dotted-pair

©1993 John Mylopoulos

7

White Space

Variables and Constants generally can be arbitrarily long (up to some limit). The beginning or end of keywords, variables and constants requires a delimiter, so that we know where they start or end.

White space (spaces, tabs, or end-of-line), or a character that must be part of a different token, is used to separate tokens.

Note that in Fortran this wasn't true. Except in strings, blanks could be added or left out anywhere.

Blank Character as an operator

SNOBOL is a string-oriented language, with many of the "deep" features of LISP or PROLOG.

String concatenation is a frequently used feature. for which SNOBOL uses the blank character:

```
A =  concatenate("abc","def")

A =  "abc" || "def"

A =  "abc"  "def"                    --> SNOBOL
```


©1993 John Mylopoulos

8

Statement Layout

Fortran, Cobol, most assembly languages and SNOBOL made the compiler's job easier by requiring a line-oriented layout of statements.

For example the rules for FORTRAN are:

- statement must be between columns 7 and 72
- Optional label must be between column 1 and 5
- If there is any character in column 6, this line continues the previous one.
- At most one statement per line. Blank lines are OK.
- Comments are indicated by a C in column 1

Likewise, Cobol restricts the parts of a line in which a statement must start.

Many of the more recent languages are free-form:

- can break statements over lines
- can have more than 1 statement per line
- no restrictions on where statements can appear

(but neatly indented layout is even more important)

Compilers must be able to tell where statements start and end. Many languages use a statement-separator-character (usually **;**). Missing separators can confuse a compiler.

©1993 John Mylopoulos

9

Comment Layout

Some languages, such as FORTRAN, can't have comments on the same line as code.

Most other languages have one of two methods:

• A comment indicator (usually one or two characters) makes the rest of the line a comment. For example:

Turing , some Prologs	(%)
Lisp	(;)
Ada	(--)

(No code can follow a comment)

• The comment indicator makes everything a comment (possibly for several lines) until an end-of-comment indicator. For example:

PL/I, C, some Pascals	/*	*/
Other Pascals	{	}

If code unintentionally follows a comment, part of the program is "commented-out", and doesn't execute.

©1993 John Mylopoulos

10

Defining the Grammar of a Programming Langiage: Phrase Structure Grammars

A phrase structure grammar (hereafter grammar) is a 4-tuple

$$G = <T, N, R, S>$$

where

T - set of terminal symbols (or terminals); this is the vocabulary of the language defined by the grammar.

N - set of nonterminal symbols (or nonterminals); this set defines word and grammatical categories of the language

R - set of grammatical rules; each rule has the form

$$A_1A_2...A_n \rightarrow B_1B_2...B_m$$

where A_i, B_j are terminal or nonterminal symbols

S - the starting symbol, has to be a nonterminal

Example

$G_{simple} = <\{0,1\}, \{S\}, \{S \rightarrow 0S, S \rightarrow 1S, S \rightarrow 0\}, S>$

G_{simple} has two terminals (0 and 1), one nonterminal (S) and three rules.

©1993 John Mylopoulos

11

Phrase Structured Grammar Derivations

A string is a sequence of terminal and/or non-terminal symbols.

String w is directly derivable from string u with respect to grammar G, $u \Rightarrow w$ wrt G, if $u = u_1 \alpha u_2$, $w = u_1 \beta u_2$ and there is a rule in G of the form $\alpha \rightarrow \beta$

String w is derivable from string u with respect to grammar G, $u \stackrel{*}{\Rightarrow} w$ wrt G if there is a sequence of strings $w_0, w_1, \ldots, w_n$ such that $w_0 = u$, $w_n = w$ and

$w_0 \Rightarrow w_1$, $w_1 \Rightarrow w_2, \ldots, w_{n-1} \Rightarrow w_n$,

all wrt G

The language defined by grammar G, L(G), is given by

L(G) = { w | w consists of terminal symbols in G and $S \stackrel{*}{\Rightarrow} w$ in G, where S is the starting symbol of G}

©1993 John Mylopoulos

12

Example

G_{simple} = <{0,1}, {S}, {S → 0S, S → 1S, S →0}, S>

Here are some derivations with respect to G_{simple}

$S \Rightarrow 0S \Rightarrow 00S \Rightarrow 000$

$S \Rightarrow 1S \Rightarrow 10S \Rightarrow 101S \Rightarrow 1010$

What is the language defined by G_{simple}?

©1993 John Mylopoulos

13

...And Another Example

Consider grammar $G_{identifier}$ with terminals a, b, c, d,..., z, 0, 1, 2,..., 9. Also nonterminals Letter, Digit, Identifier, with Identifier as starting symbol. Also the rules

Letter → a, Letter → b, ..., Letter → z
Digit → 0, Digit → 1, ..., Digit → 9
Identifier → Identifier Letter
Identifier → Identifier Digit
Identifier → Letter

Here is a derivation of ab7:

Identifier ⇒ Identifier Digit ⇒ Identifier Letter Digit
⇒ Letter Letter Digit ⇒ a Letter Digit ⇒
⇒ a b Digit ⇒ a b 7

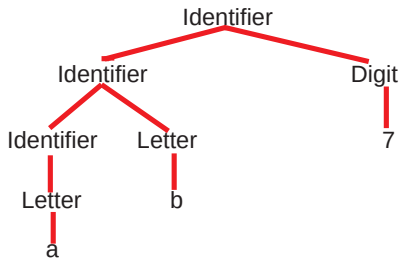
Prove that this grammar generates strings which begin with a letter and continue with zero or more letters or digits.

©1993 John Mylopoulos

14

Derivation Trees

Another way to look at derivations



©1993 John Mylopoulos

15

Another Example

Consider the grammar G_{ab} with terminals {a, b}, non-terminals {S, A, B} and rules:

$S \rightarrow aA$
$A \rightarrow aA, A \rightarrow aB$
$B \rightarrow bB, B \rightarrow b$

Lemma: Prove that B generates strings of one or more b's.

Lemma: Prove that A generates strings of one or more a's followed by one or more b's.

Theorem: The language generated by this grammar consists of strings consisting of two or more a's followed by one or more b's.

©1993 John Mylopoulos

16

Let's Simplify the Notation!

$\{ a_1 a_2 ... a_n \}$ - repetition of enclosed string zero or more times
$[a_1 a_2 ... a_n]$ - repetition of enclosed string zero or one times
$a_1 a_2 ... a_n \mid b_1 b_2 ... b_m$ - either $a_1 a_2 ... a_n$ or $b_1 b_2 ... b_m$

The rules of G_{simple} can then be rewritten as

$S \rightarrow 0S \mid 1S \mid 0$

or, even more simply

$S \rightarrow \{0 \mid 1\} 0$ (...why?)

©1993 John Mylopoulos

17

The Backus-Naur Form (BNF)

To avoid having to specify explicitly the nonterminal and terminal symbols of a grammar, we'll also use angular brackets (<S>) to indicate nonterminals.

Finally, we'll use "::=" instead of "→". Grammars defined through this notation will be said to be in Backus-Naur Form (BNF)

$G_{identifier}$ Revisited

We can rewrite the rules of $G_{identifier}$ using the above conventions:

<Letter> ::= a | b | ... | z
<Digit> ::= 0 | 1 | 2 | ... | 9
<Identifier> ::= <Letter> { <Letter> | <Digit>}

©1993 John Mylopoulos

18

Classification of Phrase Structure Grammars

Regular Grammars (Type 3 Grammars): Rules are of the form
<A> ::= a or <A> ::= a<B> or <A> ::= <B> a

Context-Free Grammars (Type 2 Grammars): Rules are of the form
$$<A> ::= B_1B_2..B_n$$
where B_i , i = 1, 2..,n is a terminal or nonterminal

Context-Sensitive Grammars (Type 1 Grammars): Rules are of the form
$$\mathbf{A_1A_2...A_n} ::= \mathbf{B_1B_2...B_m}$$
where n ≤ m.

Type 0 Grammars: Arbitrary rules allowed

Clearly every regular grammar is also context-free; every context-free grammar is also context-sensitive; every context-sensitive grammar is also type 0.

It turns out that context-free grammars are particularly useful in defining the syntax of programming languages.

©1993 John Mylopoulos

19

Context-Free Grammars

• More powerful than regular grammars: Consider the language

L = { w | w = aa..abb..b, with equal number of a's, b's}

• Used heavily in defining the syntax of programming languages

• Less powerful than context-sensitive grammars: Consider the language

L = { w | w = aa...abb...bcc...c, with equal number of a's, b's c's}

Can't be defined by a context-free grammar, but it can be defined in terms of a context-sensitive grammar.

©1993 John Mylopoulos

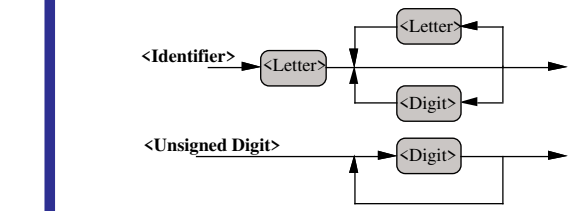
20

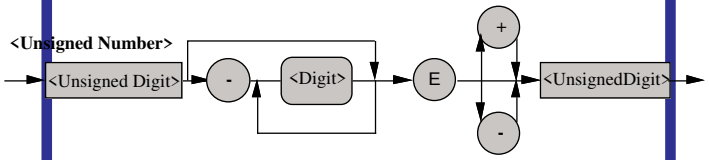
Syntax Diagrams

Syntax Diagrams are a pictorial way of representing BNF notation. For example the BNF representation of <identifier> in:

<Identifier> ::= <Letter> { <Letter> | <Digit> }
<Unsigned Digit> ::= <Digit> {<Digit>}
<Unsigned Number> ::=
<Unsigned Digit> [. <Unsigned Digit>]
E [+ | -] <Unsigned Digit>

can be represented using the following syntax diagrams.





©1993 John Mylopoulos

21

Syntax Diagrams

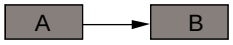
t — Terminal symbol

<N> — Nonterminal defined in terms of terminals
e.g., <N> ::= t1 | t2 | ...

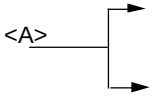
<N> — Nonterminal defined in terms of another syntax diagram

Let ▭ represent a box of any of the above kinds

A B

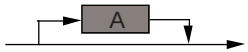


<A> ::= ... | ...



{ A }

[A]



©1993 John Mylopoulos

22

Syntax Diagrams (continued)

For example the BNF representation of the following fragment defining a for-loop:

<statement> ::= <assignment> | <for-loop> | ...
<for-loop> ::= for < range> {< statement>} end for
<range> ::= ...

can be represented using the following syntax diagram:

???

©1993 John Mylopoulos

23

ALGOL 60 Syntax

Context free grammars were used for the first time in the definition of the programming language ALGOL 60. Here is a fragment of the ALGOL 60 report:

....
2.4 Identifiers

2.4.1 Syntax
<Identifier> ::= <Letter>
| <Identifier> <Letter>
|<Identifier> <Digit>

2.4.2 Examples
q soup V12a
marilyn a324na

2.4.3 Semantics
Identifiers have no inherent meaning...The same identifier cannot be used to denote two different quantities except when these have different scopes...

Backus and Naur were members of the committee that defined ALGOL 60.

©1993 John Mylopoulos

24

Another example: A Fragment of a PL Grammar

<Statement> ::= [<Label> :] <Unlabelled Statement>

<Unlabelled Statement> ::= <Simple Statement> |
<Structured Statement>

<Simple Statement> ::= <Assignment Statement> |
<Procedure Call> | <Goto Statement>

<Structured Statement> ::= <Block Statement> |
<Conditional Statement> | <Iterative Statement>

<Block Statement> ::=
begin <Statement> { ; <Statement> } end

<Conditional Statement> ::= if <Boolean Expression>
then <Statement> [else <Statement>]

<Iterative Statement> ::=
for <Identifier> ;= <Expression> to <Expression>
do <Statement>

...

©1993 John Mylopoulos

25

Syntactic Ambiguity

• A sequence of symbols w is well formed with respect to grammar G if S *⇒ w, where S is the starting symbol of G.

• A sequence of symbols w is syntactically ambiguous with respect to grammar G if there are two different derivations of w with respect to G.

Example: Consider the grammar of the previous slide and the sequence of symbols:

if <Condition> then
if <Condition> then <Statement>
else <Statement>

There are two derivations of this from the nonterminal <Conditional Statement> (which ones?).

• Syntactic vs Semantic Ambiguity

if A then if B then x := 1 else x := 2
(meaning of symbols unimportant)

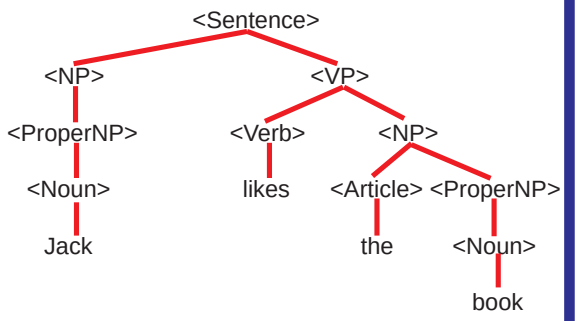
I asked Jack to give John $50. He refused
(who refused?)

©1993 John Mylopoulos

26

The Syntax of Natural Languages: A Simple Grammar for English

<Sentence> ::= <NP> <VP>
<NP> ::= <Article> <ProperNP> | <ProperNP>
<ProperNP> ::= <Adjective> <ProperNP> | <Noun>
<Article> ::= a | the
<Adjective> ::= tall | short
<Noun> ::= Jack | Jill | book
<VP> ::= <Verb> <NP>
<Verb> ::= likes | hits



"Jack likes the book"

Nonsense sentences will also be generated by this grammar

©1993 John Mylopoulos

27

Problems with BNF

In Turing, a variable name may have at most 50 characters. This limitation must be explicitly *rolled* out. For example:

```
<identifier>    ::= <letter>    [<letter> | <digit>]
                                [<letter> | <digit>]
                                    ...
                                [<letter> | <digit>]
                                    (49 times)
```

In many programming languages, a variable has to be declared before it is used. One can't say that in BNF at all.

Why?

Context-free grammars aren't powerful enough.

Programming language designers generally use a combination of context-free grammars and other techniques to define the syntax of their programming language

©1993 John Mylopoulos

28

Syntactic Analysis

• A compiler or interpreter can use a BNF description of a language to analyze a program, determine its structure, in order to generate code, or execute it directly.

• Top-Down vs Bottom-Up Analysis:

In top-down analysis, the compiler considers the grammar for a program, and tries to match against it the input program.

In bottom-up, the system starts with the input, tries to find possible syntactic structures consistent with a given grammar

• Compiler generators are programs that automatically create a new syntactic analyzer for a compiler, given a description of the syntax of a language in BNF. Many are bottom up and table-driven.

• Within a compiler, the tokenizer does the hard work of detecting the beginning and end of tokens, quoted characters, comments, and white space.

©1993 John Mylopoulos

29

User-Defined Syntax

Some languages permit the programmer to change the underlying syntax of the language.

For example, some Prolog implementations permit the partial redefinition of the priority of unary and binary operators used in expressions:

```
a+b*c  --> a+(b*c)        * higher priority
       --> (a+b)* c       + higher priority
```

These implementations of Prolog also allow binary functions to be declared as prefix or infix:

```
is_older_than(presley, maddona).

presley  is_older_than  maddona.
```


©1993 John Mylopoulos

30

A Description of BNF in BNF

```
<Bnf>      ::= <Lhs>  "::=" <Rhs>

<Lhs>      ::= <Variable>

<Rhs>      ::=  <Expr>  { "|" <Expr>}

<Expr>     ::= <Term>  {   <Term>}

<Term>     ::=  <Atom>
             |    "{"  <Rhs>  "}"
             |    "["  <Rhs>  "]"

<Atom>     ::= <Constant>
             |    <Variable>

<Variable>     ::= "<" Constant ">"

<Constant> ::= ...
```


©1993 John Mylopoulos