

Translation Process Summary

CSC 324: Principles of Programming Languages

Formal Specifications

READING: Sebesta 3.1–3.5

©Suzanne Stevenson 2001,
revised by Diana Inkpen 2001

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

1

2

Syntax and Semantics of Programming Languages

Syntax:

Describes what a legal program may look like—its **form**.

- What are the components of the language? (*words*)
- How may they be combined with each other? (*grammar*)

3

Syntax and Semantics of Programming Languages

Semantics:

Defines the meaning of a legal program.

- Defines meaning of each program construct
- Normal and abnormal termination; nontermination

Inherently harder than syntax.

State-of-the-art is to use English descriptions (though see Sebesta 3.6 for formal approaches under investigation).

4

Syntax and Semantics of Programming Languages

Example semantic description from
Common Lisp: The Reference:

USAGE

(equal x y)

DESCRIPTION

Returns true if arguments x and y are isomorphic. (By 'isomorphic' we mean that the arguments denote structurally similar objects with equivalent values.) Objects with the same printed representation are isomorphic, and thus equal, with the exceptions and additions noted below.
...

5

Backus-Naur Form (BNF)

BNF: a formal notation for describing syntax—how components can be combined to form a valid program.

- To specify which programs are legal, and which are not
- To describe the structure of programs

7

Tokens, or Terminal Symbols

- Smallest “atomic” units of syntax
- Used to build all the other constructs

- E.g., Pascal:

```
program begin if then ...
```

```
= * / - < > = <= >= <>
```

```
( ) [ ] ; := . , ...
```

```
3.14 28 ...
```

```
square addEntry ...
```

6

Example BNF Grammar for a small language

```
<program> ::= begin <stmt_list> end
```

```
<stmt_list> ::= <stmt> | <stmt> ; <stmt_list>
```

```
<stmt> ::= <var> := <expression>
```

```
<var> ::= A | B | C
```

```
<var> ::= <var> + <var> |  
          <var> - <var> |  
          <var>
```

Each symbol can be expanded by any of its rules until only tokens are left, in which case a valid statement results.

8

Parsing in a Grammar (\mathcal{L})

Derivation in a Grammar (\mathcal{L})

Can $X2 := 0$ be **generated** from \mathcal{L} ?

Terminology: leftmost (or canonical) derivation.

13

Can we **recognize** $X2 := 0$ as being in \mathcal{L} ?

14

Parse Trees (in \mathcal{L})

A *parse tree* of $X2 := 0$ in \mathcal{L} :

Each internal node is a nonterminal; its children are the RHS of a production for that NT.

The parse tree demonstrates that the grammar generates the terminal string on the frontier.

15

Grammars are not Unique

Consider \mathcal{L}' :

Terminals letters, digits, $:=$

Nonterminals $\langle \text{letter} \rangle$ $\langle \text{digit} \rangle$ $\langle \text{ident} \rangle$
 $\langle \text{stmt} \rangle$ $\langle \text{letterordigit} \rangle$

Productions

- $\langle \text{stmt} \rangle ::= \langle \text{ident} \rangle := 0$
- $\langle \text{ident} \rangle ::= \langle \text{letter} \rangle |$
 $\langle \text{ident} \rangle \langle \text{letterordigit} \rangle$
- $\langle \text{letterordigit} \rangle ::= \langle \text{letter} \rangle | \langle \text{digit} \rangle$
- $\langle \text{letter} \rangle ::= A | B | C | \dots | Z$
- $\langle \text{digit} \rangle ::= 0 | 1 | 2 | \dots | 9$

Start Symbol $\langle \text{stmt} \rangle$

16

\mathcal{L} and \mathcal{L}' generate the same language, but yield different parse trees.

Grammars and Programming Languages

Many grammars may correspond to one programming language.

Good grammars:

- capture the logical structure of the language,
- use meaningful names,
- are easy to read,
- are as unambiguous as possible,
- ...

What does this list of desirable properties remind you of?

17

18

Simple Statement Grammar

$\langle \text{start} \rangle ::= \langle \text{stmt} \rangle$

$\langle \text{stmt} \rangle ::= \langle \text{if-stmt} \rangle \mid \langle \text{assgn} \rangle$

$\langle \text{if-stmt} \rangle ::= \mathbf{if} \langle \text{expr} \rangle \mathbf{then} \langle \text{stmt} \rangle \mid$
 $\mathbf{if} \langle \text{expr} \rangle \mathbf{then} \langle \text{stmt} \rangle \mathbf{else} \langle \text{stmt} \rangle$

$\langle \text{assgn} \rangle ::= \langle \text{id} \rangle := \langle \text{d} \rangle$

$\langle \text{expr} \rangle ::= \langle \text{id} \rangle = 0$

$\langle \text{d} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$\langle \text{id} \rangle ::= a \mid b \mid c \mid \dots \mid z$

19

Dangling Else Ambiguity

How are compound if statements parsed with this grammar?

$\mathbf{if} \ x = 0 \ \mathbf{then} \ \mathbf{if} \ y = 0 \ \mathbf{then} \ z := 1 \ \mathbf{else} \ w := 2$

20

Dangling Else Ambiguity

```
if x = 0 then if y = 0 then z := 1 else w := 2
```

21

Changing the Language to Include Delimiters

Algol 68 if-statement grammar:

```
<start> ::= <stmt>
<stmt> ::= <if-stmt> | <assgn>

<if-stmt> ::= if <expr> then <stmt> fi |
           if <expr> then <stmt>
           else <stmt>
           fi
<assgn> ::= <id> := <d>
<expr> ::= <id> = 0

<d> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<id> ::= a | b | c | ... | z
```

23

Ambiguity

A grammar is *ambiguous* when a sentence in $L(G)$ has more than one canonical derivation (therefore, more than one valid parse tree).

How to deal with ambiguity?

1. Change the language to include **delimiters**
2. Change the grammar to impose **associativity** and **precedence**

22

Arithmetic Expression Grammar

```
<start> ::= <expr>
<expr> ::= <expr> + <expr> |
           <expr> - <expr> |
           <expr> * <expr> |
           <expr> / <expr> |
           <d> | <l>
<d> ::= 0 | 1 | 2 | 3 | ... | 9
<l> ::= a | b | c | ... | z
```

24

Ambiguous Parse Trees

Parse "8 - 3 * 2":

Changing the Language to Include Delimiters

$\langle \text{expr} \rangle ::= (\langle \text{expr} \rangle) - (\langle \text{expr} \rangle) \mid$

$(\langle \text{expr} \rangle) * (\langle \text{expr} \rangle) \mid$

$\langle l \rangle \mid \langle d \rangle$

$(8) - ((5) * (2))$

$((8) - (5)) * (2)$

25

26

Changing the Grammar to Impose Precedence

Grouping In Parse Tree Now Reflects Precedence

27

28

Precedence

- Low Precedence:

Addition + and Subtraction –

- Medium Precedence:

Multiplication * and Division /

- Higher Precedence:

Exponentiation ^

- Highest Precedence:

Parenthesized expressions (<expr>)

⇒ Ordered lowest to highest in grammar.

29

Still Have Ambiguity...

3 – 2 – 1 still a problem:

- Grouping of operators of same precedence not disambiguated.
- Non-commutative operators: only one parse tree correct.
- Operators may be **left** or **right** associative.

30

Imposing Associativity

Simple grammars with left/right recursion for –:

$$\langle \text{expr} \rangle ::= \langle \text{d} \rangle - \langle \text{expr} \rangle \mid \langle \text{d} \rangle$$
$$\langle \text{d} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid \dots \mid 9$$
$$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle - \langle \text{d} \rangle \mid \langle \text{d} \rangle$$
$$\langle \text{d} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid \dots \mid 9$$

31

Associativity

- Deals with operators of same precedence
- Implicit grouping or parenthesizing
- Left associative: *, /, +, –
- Right associative: ^

32

Dealing with Ambiguity

1. Can't *always* remove an ambiguity from a grammar by restructuring productions
2. An inherently ambiguous language does not possess an unambiguous grammar
3. There is no algorithm that can examine an arbitrary context-free grammar and tell if it is ambiguous, i.e., detecting ambiguity in context-free grammars is an *undecidable* problem

Complete, Unambiguous Arithmetic Expression Grammar

$\langle \text{start} \rangle ::= \langle e \rangle$
 $\langle e \rangle ::= \langle e \rangle + \langle t \rangle \mid \langle e \rangle - \langle t \rangle \mid \langle t \rangle$
 $\langle t \rangle ::= \langle t \rangle * \langle f \rangle \mid \langle t \rangle / \langle f \rangle \mid \langle f \rangle$
 $\langle f \rangle ::= \langle g \rangle ^ \langle f \rangle \mid \langle g \rangle$
 $\langle g \rangle ::= (\langle e \rangle) \mid \langle n \rangle \mid \langle i \rangle$
 $\langle n \rangle ::= 0 \mid 1 \mid 2 \mid \dots \mid 9$
 $\langle i \rangle ::= a \mid b \mid c \mid \dots \mid z$

33

An Inherently Ambiguous Language

Suppose we want to generate the following language:

$$\mathcal{L} = \{a^i b^j c^k \mid i, j, k \geq 1, i = j \text{ or } j = k\}$$

Grammar:

35

Two Parse Trees for $a^i b^i c^i$

34

36

Extended BNF (EBNF)

1. If no confusion can result, terminals are sometimes just written as they are.
 - Terminals that are reserved words are quoted, 'if', or in boldface, **begin**.
 - Terminals that are the same as EBNF symbols are quoted, e.g., '(').
2. (...) is used for grouping, usually with the alternative symbol |.
3. [...] means the enclosed is optional (i.e., 0 times or 1 time).
4. {...} means repeat the enclosed 0 or more times.

37

Formalisms for Lexical and Syntactic Analysis

First steps of translation process:

1. Lexical Analysis: Converts source code into sequence of tokens.
2. Syntax Analysis: Structures tokens into initial parse tree.

For (2), we use **context-free grammars** and associated **parsing algorithms**.

For (1), we use **regular grammars** and **finite state automata**.

39

Examples:

$\langle \text{Ident} \rangle ::= \langle \text{Letter} \rangle \{ \langle \text{Letterordigit} \rangle \}$

$\langle \text{Letterordigit} \rangle ::= \langle \text{Letter} \rangle \mid \langle \text{Digit} \rangle$

$\langle \text{Expr} \rangle ::= \langle \text{Term} \rangle \{ ('+' \mid '-') \langle \text{Term} \rangle \}$

$\langle \text{Term} \rangle ::= \langle \text{Ident} \rangle \{ ('*' \mid '/') \langle \text{Ident} \rangle \}$

$\langle \text{Ifstm} \rangle ::= \mathbf{if} \langle \text{Expr} \rangle \mathbf{then} \langle \text{Stm} \rangle$
 $\quad \quad \quad [\mathbf{else} \langle \text{Stm} \rangle]$

$\langle \text{Whilestm} \rangle ::= \mathbf{while} \text{'('} \langle \text{Expr} \rangle \text{')' } \langle \text{Stm} \rangle$

$\langle \text{Block} \rangle ::= \mathbf{begin} \langle \text{Stm} \rangle \{ ; \langle \text{Stm} \rangle \} \mathbf{end}$

$\langle \text{Arrayelem} \rangle ::= \langle \text{Id} \rangle \text{'['} \langle \text{Id} \rangle \text{' , ' } \langle \text{Id} \rangle \text{']'}$

38

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.

40

Regular Grammars

Defined over alphabet Σ , using non-terminals and grammar rules, analogous to terminals (words), NTs, and production rules of CFGs, but **more restricted**.

Left-recursive:

$\langle N \rangle ::= \langle X \rangle a b$
 $\langle X \rangle ::= a \mid \langle X \rangle b$

Right-recursive:

$\langle N \rangle ::= b \mid b \langle Y \rangle$
 $\langle Y \rangle ::= a b \mid a b \langle Y \rangle$

41

Regular Expressions

A compact notation for regular languages.

RE	Language
a	$\{a\}$
ϵ	$\{\epsilon\}$
$r \mid s$	$L(r) \cup L(s)$
rs	$\{rs \mid r \in L(r), s \in L(s)\}$
r^+	$r \cup rr \cup rrr \cup \dots$ (any number of r 's concatenated)
r^*	$\{\epsilon\} \cup r \cup rr \cup rrr \cup \dots$ ($r^* = r^+ \cup \{\epsilon\}$)
(s)	$L(s)$

(all left-assoc. in order of increasing prec.)

42

Examples of Regular Expressions

<u>RE</u>	<u>Language</u>
-----------	-----------------

a|bc

(a|b)c

a ϵ

a*|b

ab*

ab*|c+

(a|b)*

(0|1)*1

43

Regular Grammars and Regular Expressions

$\langle N \rangle ::= \langle X \rangle \mid \langle Y \rangle$
 $\langle X \rangle ::= a \mid \langle X \rangle b$
 $\langle Y \rangle ::= c \mid \langle Y \rangle c$

is equivalent to:

44

Regular Expressions for Programming Languages

Let letter stand for $A \mid B \mid C \mid \dots \mid Z$

Let digit stand for $0 \mid 1 \mid 2 \mid \dots \mid 9$

identifier:

integer constant:

real constant:

exponent part of

scientific notation:

Compare: $\text{digit}^* . \text{digit}^+$ with

$\langle \text{Real} \rangle ::= \langle \text{Num} \rangle . \langle \text{Frac} \rangle$

$\langle \text{Num} \rangle ::= \langle \text{Num} \rangle \langle \text{Digit} \rangle \mid \epsilon$

$\langle \text{Frac} \rangle ::= \langle \text{Digit} \rangle \langle \text{Frac} \rangle \mid \langle \text{Digit} \rangle$

$\langle \text{Digit} \rangle ::= 0 \mid 1 \mid 2 \mid \dots \mid 9$

45

Limitations of BNF/CFGs

Other things are simply awkward in BNF:

- An identifier cannot be longer than 50 characters.

These aspects are either specified informally, or by using **attribute grammars** (see Sebesta 3.5).

47

Limitations of BNF/CFGs

CFGs are more powerful than REs, but can't describe everything we need in PL syntax; e.g.:

- Must declare an identifier before using it.
- Cannot declare the same identifier twice in the same block.
- The number of parameters in a procedure call must equal the number of parameters in its definition.

46

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 is named after Noam Chomsky, who did research in grammars for natural language.

48

Abstract versus Concrete Syntax

Concrete Syntax:

representation of a construct in a particular language, including placement of keywords and delimiters

Abstract Syntax:

structure of meaningful components of each language construct

Same abstract syntax, different concrete syntax:

Pascal **while** x <> A[i] **do**
 i := i + 1
 end

Modula-2 **WHILE** x <> A[i] **DO**
 i := i + 1;
 END

C **while** (x != A[i])
 i = i + 1;

49

50

Example:

$\langle S \rangle ::= \langle E \rangle$
 $\langle E \rangle ::= \langle E \rangle - \langle T \rangle \mid \langle T \rangle$
 $\langle T \rangle ::= \langle T \rangle * \text{id} \mid \text{id}$

Consider A*B-C:

Parse Tree	Abstract Syntax Tree
$\langle S \rangle$	-
$\langle E \rangle$	* C
$\langle E \rangle$ - $\langle T \rangle$	A B
$\langle T \rangle$ id	"Essence" of construct
$\langle T \rangle$ * id C	
id B	
A	

51

Summary of Part I: Programming Language Theory

- Why have high-level languages?
- Issues in good language design
- Stages of translation process
- Compilation vs Interpretation
- Elements of syntax
- BNF and CFGs
- Derivation and parsing
- Resolution of ambiguity in grammars
- EBNF
- Regular expressions
- Limitations of REs and CFGs
- Abstract and concrete syntax

52