CS5142 Scripting Languages
Fall 2013
Context-Free Grammars, Parsing
Acknowledgment

These slides are based on slides and lecture notes of Clark Barrett and Robert Grimm.
The first computer programs were written in *machine language*. Machine language is just a sequence of ones and zeroes. The computer interprets sequences of ones and zeroes as *instructions* that control the *central processing unit* (CPU) of the computer. The length and meaning of the sequences depends on the CPU.

**Example**

On the 6502, an 8-bit microprocessor used in the Apple II computer, the following bits add 1 plus 1: 10101001000000010110100100000001.

Or, using base 16, a common shorthand: A9016901.

Programming in machine language requires an extensive understanding of the low-level details of the computer and is extremely tedious if you want to do anything non-trivial.

But it *is* the most straightforward way to give instructions to the computer: no extra work is required before the computer can run the program.
A Brief History of Programming Languages

Before long, programmers started looking for ways to make their job easier. The first step was *assembly language*.

Assembly language assigns meaningful names to the sequences of bits that make up instructions for the CPU.

A program called an *assembler* is used to translate assembly language into machine language.

**Example**

The assembly code for the previous example is:

- `LDA #$01`
- `ADC #$01`

**Question:** How do you write an assembler?

**Answer:** in machine language!
A Brief History of Programming Languages

Before long, programmers started looking for ways to make their job easier. The first step was assembly language.

Assembly language assigns meaningful names to the sequences of bits that make up instructions for the CPU.

A program called an assembler is used to translate assembly language into machine language.

**Example**

The assembly code for the previous example is:

```
LDA #$01
ADC #$01
```

Question: *How do you write an assembler?*

Answer: in machine language!
As computers became more powerful and software more ambitious, programmers needed more efficient ways to write programs.

This led to the development of high-level languages, the first being FORTRAN.

High-level languages have features designed to make things much easier for the programmer.

In addition, they are largely machine-independent: the same program can be run on different machines without rewriting it.

But high-level languages require a compiler. The compiler’s job is to convert high-level programs into machine language. More on this later...

Question: How do you write a compiler?

Answer: in assembly language (at least the first time)
A Brief History of Programming Languages

As computers became more powerful and software more ambitious, programmers needed more efficient ways to write programs.

This led to the development of *high-level* languages, the first being FORTRAN.

High-level languages have features designed to make things much easier for the programmer.

In addition, they are largely *machine-independent*: the same program can be run on different machines without rewriting it.

But high-level languages require a *compiler*. The compiler’s job is to convert high-level programs into machine language. More on this later...

Question: *How do you write a compiler?*

Answer: in assembly language (at least the first time)
Compilation overview

Major phases of a compiler:

1. **Lexer**: Text $\rightarrow$ Tokens

2. **Parser**: Tokens $\rightarrow$ Parse Tree

3. **Intermediate code generation**: Parse Tree $\rightarrow$ Intermed. Representation (IR)

4. **Optimization I**: IR $\rightarrow$ IR

5. **Target code generation**: IR $\rightarrow$ assembly/machine language

6. **Optimization II**: target language $\rightarrow$ target language
Syntax and Semantics

Syntax refers to the structure of the language, i.e. what sequences of characters are well-formed programs.

- Formal specification of syntax requires a set of rules
- These are often specified using grammars

Semantics denotes meaning:

- Given a well-formed program, what does it mean?
- Meaning may depend on context

We now look at grammars in more detail.
Grammars

A grammar $G$ is a tuple $(\Sigma, N, S, \delta)$, where:

- $N$ is a set of *non-terminal* symbols
- $S \in N$ is a distinguished non-terminal: the *root* or *start* symbol
- $\Sigma$ is a set of *terminal* symbols, also called the *alphabet*. We require $\Sigma$ to be disjoint from $N$ (i.e. $\Sigma \cap N = \emptyset$).
- $\delta$ is a set of rewrite rules (productions) of the form:

\[ ABC \ldots \rightarrow XYZ \ldots \]

where $A, B, C, D, X, Y, Z$ are terminals and non-terminals.

Any sequence consisting of terminals and non-terminals is called a *string*.

The *language* defined by a grammar is the set of strings containing *only* terminal symbols that can be generated by applying the rewriting rules starting from $S$. 

Consider the following grammar $G$:

- $N = \{S, X, Y\}$
- $S = S$
- $\Sigma = \{a, b, c\}$
- $\delta$ consists of the following rules:
  - $S \rightarrow b$
  - $S \rightarrow XbY$
  - $X \rightarrow a$
  - $X \rightarrow aX$
  - $Y \rightarrow c$
  - $Y \rightarrow Yc$

Some sample derivations:

- $S \rightarrow b$
- $S \rightarrow XbY \rightarrow abY \rightarrow abc$
- $S \rightarrow XbY \rightarrow aXbY \rightarrow aaXbY \rightarrow aaabY \rightarrow aaabc$
The Chomsky hierarchy

• Regular grammars (Type 3)
  – All productions have a single non-terminal on the left and a terminal and optionally a non-terminal on the right
  – Non-terminals on the right side of rules must either always precede terminals or always follow terminals
  – Recognizable by finite state automaton

• Context-free grammars (Type 2)
  – All productions have a single non-terminal on the left
  – Right side of productions can be any string
  – Recognizable by non-deterministic pushdown automaton

• Context-sensitive grammars (Type 1)
  – Each production is of the form \( \alpha A \beta \rightarrow \alpha \gamma \beta \),
  – \( A \) is a non-terminal, and \( \alpha, \beta, \gamma \) are arbitrary strings (\( \alpha \) and \( \beta \) may be empty, but not \( \gamma \))
  – Recognizable by linear bounded automaton

• Unrestricted grammars (Type 0)
  – No restrictions
  – Recognizable by turing machine
Tokens

Tokens are the basic building blocks of programs:

- **keywords** (`begin, end, while`).
- **identifiers** (`myVariable, yourType`)  
- **numbers** (`137, 6.022e23`)  
- **symbols** (`+`, `−`)  
- **string literals** ("Hello world")  
- **described** (mainly) by regular grammars

**Example**: identifiers

\[
\text{Id} \rightarrow \text{Letter IdRest} \\
\text{IdRest} \rightarrow \epsilon \mid \text{Letter IdRest} \mid \text{Digit IdRest}
\]

Other issues: international characters, case-sensitivity, limit of identifier length
Backus-Naur Form (BNF) is a notation for context-free grammars:

- alternation: `Symb ::= Letter | Digit`
- repetition: `Id ::= Letter {Symb}`
  or we can use a Kleene star: `Id ::= Letter Symb*`
  for one or more repetitions: `Int ::= Digit+`
- option: `Num ::= Digit+[. Digit*]`

Note that these abbreviations do not add to expressive power of grammar.
Parse trees

A parse tree describes the way in which a string in the language of a grammar is derived:

- root of tree is start symbol of grammar
- leaf nodes are terminal symbols
- internal nodes are non-terminal symbols
- an internal node and its descendants correspond to some production for that non terminal
- top-down tree traversal represents the process of generating the given string from the grammar
- construction of tree from string is parsing
Ambiguity

If the parse tree for a string is not unique, the grammar is ambiguous:

\[ E ::= E + E \mid E \ast E \mid Id \]

Two possible parse trees for \( A + B \ast C \):  
- \( ((A + B) \ast C) \)  
- \( (A + (B \ast C)) \)

One solution: rearrange grammar:

\[ E ::= E + T \mid T \]
\[ T ::= T \ast Id \mid Id \]

Why is ambiguity bad?
Ambiguity

If the parse tree for a string is not unique, the grammar is *ambiguous*:

\[ E ::= E + E | E \times E | Id \]

Two possible parse trees for \( A + B \times C \):

- \((A + B) \times C\)
- \((A + (B \times C))\)

One solution: rearrange grammar:

\[
\begin{align*}
E ::= & \ E + \ T \mid \ T \\
T ::= & \ T \times \ Id \mid \ Id \\
\end{align*}
\]

*Why is ambiguity bad?*
Dangling else problem

Consider:

\[ S ::= \text{if } E \text{ then } S \]
\[ S ::= \text{if } E \text{ then } S \text{ else } S \]

The string

\[ \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2 \]

is ambiguous (Which \textit{then} does \textit{else }\textit{S2} match?)

Solutions:

- \textbf{PASCAL rule}: else matches most recent if
- grammatical solution: different productions for balanced and unbalanced if-statements
- grammatical solution: introduce explicit end-marker
Dangling else problem

Consider:

\[
S ::= \text{if } E \text{ then } S \\
S ::= \text{if } E \text{ then } S \text{ else } S
\]

The string

\[
\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2
\]

is ambiguous (Which \textit{then} does \textit{else } \textit{S2} match?)

Solutions:

- \textsc{Pascal} rule: else matches most recent if
- grammatical solution: different productions for balanced and unbalanced if-statements
- grammatical solution: introduce explicit end-marker