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.

A Brief History of Programming Languages

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!

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)

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 $\longrightarrow$ Tokens
2. *Parser*: Tokens $\longrightarrow$ Parse Tree
3. *Intermediate code generation*: Parse Tree $\longrightarrow$ Intermed. Representation (IR)
4. *Optimization I*: IR $\longrightarrow$ IR
5. *Target code generation*: IR $\longrightarrow$ assembly/machine language
6. *Optimization II*: target language $\longrightarrow$ 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 (Σ, N, S, δ), where:

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

$$\mathrm{ABC} \ldots \rightarrow \mathrm{XYZ} \ldots$$

  where $\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{X}, \mathrm{Y}, \mathrm{Z}$ 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.

Grammars

Consider the following grammar G:

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

Some sample derivations:

- $S \to b$
- $S \to XbY \to abY \to abc$
- $S \to XbY \to aXbY \to aaXbY \to aaabY \to aaabc$

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 preceed 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 $\alpha A \beta \to \alpha \gamma \beta$,
  - A is a non-terminal, and α, β, γ are arbitrary strings (α and β may be empty, but not γ)
  - 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

Backus-Naur Form (BNF) is a notation for context-free grammars:

- alternation: $\mathrm{Symb} ::= \mathrm{Letter} \mid \mathrm{Digit}$
- repetition: $\mathrm{Id} ::= \mathrm{Letter}\ \{\mathrm{Symb}\}$

  or we can use a Kleene star: $\mathrm{Id} ::= \mathrm{Letter}\ \mathrm{Symb}^*$

  for one or more repetitions: $\mathrm{Int} ::= \mathrm{Digit}^+$
- option: $\mathrm{Num} ::= \mathrm{Digit}^+[.\ \mathrm{Digit}^*]$

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*:

$$\mathrm{E} ::= \mathrm{E}\ +\ \mathrm{E} \mid \mathrm{E}\ *\ \mathrm{E} \mid \mathrm{Id}$$

Two possible parse trees for $\mathrm{A} + \mathrm{B} * \mathrm{C}$:

- $((\mathrm{A} + \mathrm{B}) * \mathrm{C})$
- $(\mathrm{A} + (\mathrm{B} * \mathrm{C}))$

One solution: rearrange grammar:

$$\mathrm{E} ::= \mathrm{E}\ +\ \mathrm{T} \mid \mathrm{T}$$
$$\mathrm{T} ::= \mathrm{T}\ *\ \mathrm{Id} \mid \mathrm{Id}$$

Why is ambiguity bad?

Ambiguity

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

$$\mathrm{E} ::= \mathrm{E}\ +\ \mathrm{E} \mid \mathrm{E}\ *\ \mathrm{E} \mid \mathrm{Id}$$

Two possible parse trees for $\mathrm{A} + \mathrm{B} * \mathrm{C}$:

- $((\mathrm{A} + \mathrm{B}) * \mathrm{C})$
- $(\mathrm{A} + (\mathrm{B} * \mathrm{C}))$

One solution: rearrange grammar:

$$\begin{aligned} \mathrm{E} &::= \mathrm{E}\ +\ \mathrm{T} \mid \mathrm{T} \\ \mathrm{T} &::= \mathrm{T}\ *\ \mathrm{Id} \mid \mathrm{Id} \end{aligned}$$

Why is ambiguity bad?

Dangling else problem

Consider:

$$\mathrm{S} ::= \mathrm{if\ E\ then\ S}$$
$$\mathrm{S} ::= \mathrm{if\ E\ then\ S\ else\ S}$$

The string

$$\mathrm{if\ E1\ then\ if\ E2\ then\ S1\ else\ S2}$$

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

Solutions:

- 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:

$$\mathrm{S} ::= \mathrm{if\ E\ then\ S}$$
$$\mathrm{S} ::= \mathrm{if\ E\ then\ S\ else\ S}$$

The string

$$\mathrm{if\ E1\ then\ if\ E2\ then\ S1\ else\ S2}$$

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

Solutions:

- PASCAL rule: else matches most recent if
- grammatical solution: different productions for balanced and unbalanced if-statements
- grammatical solution: introduce explicit end-marker