CS 1622:
Syntax Analysis

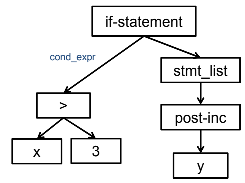
Jonathan Misurda
jmisurda@cs.pitt.edu

Parsing

Input: Sequence of tokens
Output: Abstract Syntax Tree

Example:

IF	(	ID('x')	>	NUM('3')	)	{	ID('y')	INCREMENT	;	}



Parsing

The lexing phase has left us with a set of tokens.

We now need to determine the role of those tokens in context.

We'll use a parser to produce a **parse tree** that represents the structure of the input.

A tree is used because the rules of a programming language are usually recursive.

For example:

```
if-statement = if ( condition ) statement;

statement = if-statement | while-statement | …
```

Can We Use REs for Parsing?

Quintessential example of the lack of power of REs: Matching parenthesis.

Alphabet: (and)
Language: All strings that contain properly matched and nested parenthesis

Describe strings with pattern: $(^i\ )^i$ $(i \geq 1)$:

Our finite automata would need to have states that represent each number of currently open parenthesis. (That is, a state for "(", "((", "(((", …)

That number could be infinite. REs are converted into **finite** state automata. This is a contradiction.

More Power

If regular expressions and finite state automata are insufficient for parsing, we will need a more powerful formalism.

To do this, we will use the concept of a **Context Free Language**.

Now that we have multiple categories of languages, let us generalize this notion first.

Grammar

Recall the definition of a language:

Language: set of strings over alphabet
Alphabet: finite set of symbols
Null string: ε

Sentences: strings in the language

It is possible to describe a language using a **grammar**

- Define English using English grammar (as we learn in school)

Grammars

A **grammar** consists of 4 components **(T, N, s, δ):**

T — set of **terminal** symbols
- Essentially tokens — appear in the input string

N — set of **non-terminal** symbols
- Categories of strings impose hierarchical language structure
- Useful for analysis. Examples: declaration, statement, loop, ...

s — a special non-terminal **start symbol** that denotes every sentence is derivable from it

δ — a set of **production** rules:
"LHS → RHS": left-hand-side produces right-hand-side

Derivation

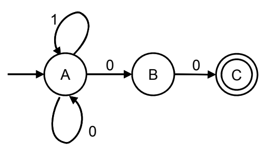
"LHS → RHS"
- Replace LHS with RHS
- Specifies how to transform one string to another

$\beta \Rightarrow \alpha$: string β derives α
- $\beta \Rightarrow \alpha$ — 1 step
- $\beta \overset{*}{\Rightarrow} \alpha$ — 0 or more steps
- $\beta \overset{+}{\Rightarrow} \alpha$ — 1 or more steps

Example

Language L = { any string with "00" at the end } (`/0{2}$/`)

Grammar G = (T, N, s, δ)
T = {0, 1}
N = {A, B}
s = A
δ = { A → 0A | 1 A | 0 B,
B → 0
}



Derivation: from grammar to language
- A ⇒ 0A ⇒ 00B ⇒ 000
- A ⇒ 1A ⇒ 10B ⇒ 100
- A ⇒ 0A ⇒ 00A ⇒ 000B ⇒ 0000
- A ⇒ 0A ⇒ 01A ⇒ ...

Chomsky Hierarchy of Languages

A classification of languages based on the form of grammar rules
- Classify not based on how complex the language is
- Classify based on how complex the grammar (the describe the language) is

Four types of grammars:
- Type 0 — recursive grammar
- Type 1 — context sensitive grammar
- Type 2 — context free grammar
- Type 3 — regular grammar

Regular Languages

Form of rules:
A → α
or
A → αB

where A,B ∈ N, α ∈ T

Regular grammars define REs.

Example:
A → 1A
A → 0

Context Free Languages

Form of rules:
A → γ

where A ∈ N, $\gamma \in (N \cup T)^+$

A can be replaced by γ at any time.

Proper CFLs have no "erase rule" where a production is replaced by ε.
- If there are rules deriving empty string, rewrite to remove empty rule (Such as in Chomsky Normal Form)

Example:
S → SS
S → (S)
S → ε

Context Sensitive Languages

Form of rules:

$\alpha A \beta \rightarrow \alpha \gamma \beta$

where $A \in N^+$; $\alpha, \beta \in (N \cup T)$; $\gamma \in (N \cup T)^+$; $|A| \leq |\gamma|$

Replace A by γ only if found in the context of α and β.

No erase rule.

Example:

aAB → aCB

Unrestricted/Recursive Languages

Form of rules:

$\alpha \rightarrow \beta$

where $\alpha \in (N \cup T)^+$, $\beta \in (N \cup T)^*$

The erase rule is allowed.

No restrictions on form of grammar rules.

Example:

aAB → aCD
aAB → aB
A → ε

Are CFGs enough for PLs?

We've determined that because of nesting and recursive relationships in programming languages that REs (type 3 grammars) are insufficient.

What about Context Free (type 2) grammars?

Imagine we want to describe the grammar of valid C or Java programs that have the declaration of a variable before their use:

S → DU
D → int identifier;
U → identifier '=' expr;

Are CFGs enough for PLs?

The CFG allows for the following derivations:

S ⇒ DU ⇒ int x; x=0;
S ⇒ DU ⇒ int x; y=0;
S ⇒ DU ⇒ int y; x=0;
S ⇒ DU ⇒ int x; y=0;

You would need a Context Sensitive grammar (type 1) to match the definition to the use.

So why do we seem to want to use CFGs?

- Some PL constructs are context free: If-stmt, declaration
- Many are not: def-before-use, matching formal/actual parameters, etc.
- We'll like CFGs because they are powerful and easily understood.
- But we'll need to add the checks that CFGs miss in later phases of the compiler.

Language Classification Summary

Regular Grammar ⊆ CFG ⊆ CSG ⊆ Recursive Grammar