Principles of Programming Languages

Topic: Formal Languages I
Review

• This class will teach you
  – some common principles underlying most languages
  – some new ways of thinking about programs (new paradigms)

• A programming language must be
  – as easy as possible for people to
    • learn
    • read
    • write
  – as easy as possible for a compiler to translate into efficient machine code or an interpreter to execute
Defining a Language

• To define a computer language we need to say
  – How do we tell if a file of characters is a legal (grammatical) program in this language? (syntax)
  – How do we define what a program in this language means? (semantics)

• Semantics: several approaches but in practice we just use English to explain the meaning

• Syntax: defined by a formal grammar
Grammars

S \rightarrow NP \ VP
NP \rightarrow \text{Name} | \text{Det} \ Noun
VP \rightarrow \text{Verb} | \text{Verb} \ NP
Name \rightarrow john | mary
Det \rightarrow a | the
Det \rightarrow some | every
Noun \rightarrow boy | girl
Verb \rightarrow runs | likes

S
  \rightarrow
  NP
  \rightarrow
  Det | Noun
  VP
  \rightarrow
  Verb | NP
  Name

the boy likes Mary
A grammar, G, is a quadruple <T,N,P,S>, where

- T is a set of terminal symbols (e.g., john, mary, a, the, some, every, boy, girl, runs, likes);
- N is a set of nonterminal symbols (e.g., S, NP, VP, Name, Det, Noun, Verb);
- P is a set of productions, or rewrite rules (e.g., Det → a the);
- S is a special start symbol.

The language of G, L(G), is the set of all terminal sequences that can be produced by applying the rewrite rules, repeatedly, starting with S.
Grammars

• For Programming Languages:
  
  Stmt → Identifier := Digit
  
  Identifier → Letter | Identifier Letter | Identifier Digit
  
  Letter → a | b | c | ... | x | y | z
  
  Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

• Backus-Naur Form (BNF):

  <stmt> ::= <id> := <digit>
  <id> ::= <letter> | <id> <letter> | <id> <digit>
  <letter> ::= a | b | c | ... | x | y | z
  <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Q: Generate \( x2 := 0 \) in this grammar (call it \( G \))?

\[
\begin{align*}
\text{<stmt>} & \rightarrow \text{id} \ := \ <\text{digit}> \\
& \rightarrow \ \text{id} <\text{digit}> \ := \ <\text{digit}> \\
& \rightarrow \ \text{letter} <\text{digit}> \ := \ <\text{digit}> \\
& \rightarrow \ x <\text{digit}> \ := \ <\text{digit}> \\
& \rightarrow \ x \ 2 \ := \ <\text{digit}> \\
& \rightarrow \ x \ 2 \ := \ 0
\end{align*}
\]

Yes! This is a leftmost or \textbf{canonical derivation} in \( G \).
Q: Recognize \(x2:=0\) as a terminal sequence in \(L(G)\)?

\[
egin{align*}
x 2 & := 0 \
  & \rightarrow \ <\text{letter}>\ 2 \ :=\ 0 \
  & \rightarrow \ <\text{id}>\ 2 \ :=\ 0 \
  & \rightarrow \ <\text{id}>\ <\text{digit}>\ :=\ 0 \
  & \rightarrow \ <\text{id}>\ :=\ 0 \
  & \rightarrow \ <\text{id}>\ :=\ <\text{digit}> \
  & \rightarrow \ <\text{stmt}> \\
\end{align*}
\]

Yes! This is a parse of the sentence \(x2:=0\) in \(G\).
Parse Trees

Each internal node is a nonterminal; its children are drawn from the right-hand side of one of the productions for that nonterminal.
Grammars are not Unique

• Consider a grammar $G'$:
  
  $<\text{stmt}> ::= <\text{ident}> ::= <\text{digit}>$
  $<\text{ident}> ::= <\text{letter}> | <\text{ident}> <\text{char}>
  $<\text{char}> ::= <\text{letter}> | <\text{digit}>
  $<\text{letter}> ::= a \mid b \mid c \mid \ldots \mid x \mid y \mid z$
  $<\text{digit}> ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

• The grammar $G'$ generates the same language as $G$, but it has different parse trees.
Grammars are not Unique

Parse Tree for G

Parse Tree for G’
Types of Grammars

• Context Free Grammars:
  – Every production has a single nonterminal on the left-hand side: \( A \rightarrow \ldots \)
  – Disallowed: \( X A \rightarrow X a \)

• Regular Grammars:
  – Productions take the form: \( A \rightarrow c \), or are all either left-linear: \( A \rightarrow B a \), or right-linear: \( A \rightarrow a B \)
  – Disallowed: \( S \rightarrow a S b \)
  – Cannot generate the language \( \{ a^n b^n \mid n = 1,2,3,\ldots \} \)
Types of Grammars

• Context Free Grammars (CFGs) are used to specify the overall structure of a programming language:
  – if/then/else, ...
  – brackets: ( ), { }, begin/end, ...

• Regular Grammars (RGs) are used to specify the structure of tokens:
  – identifiers, numbers, keywords, ...

• Note: The recognition problem for CFGs and RGs requires a different computational model (more on this later).
Ambiguity

S → NP VP
NP → Name | Det Noun | NP PP
PP → Prep NP
VP → Verb | Verb NP
Name → john | mary
Det → a | the | some | every
Prep → on | with | under | ...
Noun → man | hill | telescope | ...
Verb → saw | runs | likes | ...

CS 314, LS.LTM: L2, Formal Languages I
**Ambiguity**

... a man on a hill with a telescope
```
... a man on a hill with a telescope
```
Dangling Else

Here is a simplified grammar for Pascal:

<stmt> ::= <if-stmt> | <assign> | ...
<if-stmt> ::= if <expr> then <stmt> |
            if <expr> then <stmt> else <stmt>
<assign> ::= <id> := <digit>
<expr> ::= <id> = 0
<id> ::= a | b | c | ... | x | y | z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

How are compound “if” statements parsed using this grammar?
if x = 0 then if y = 0 then z := 1 else w := 2

Parse Tree 1
if \( x = 0 \) then if \( y = 0 \) then \( z := 1 \) else \( w := 2 \)

Q: which tree is correct?
How to Fix the Dangling Else?

- **Algol60**: use block structure
  
  ```
  if x = 0 then begin if y = 0 then z := 1 end else w := 2
  ```

- **Algol68**: use statement begin/end markers
  
  ```
  if x = 0 then if y = 0 then z := 1 fi else w := 2 fi
  ```

- **Pascal**: change the grammar of “if” statement to disallow the second parse tree, i.e., *always associate an “else” with the closest “if”*. 
How to Fix the Dangling Else?

Here is a revised grammar for Pascal:

<stmt> ::= <stmt1> | <stmt2>
<stmt1> ::= if <expr> then <stmt1> else <stmt1> | <assign> | ...
<stmt2> ::= if <expr> then <stmt> | if <expr> then <stmt1> else <stmt2>
<assign> ::= <id> := <digit>
<expr> ::= <id> = 0
:id> ::= a | b | c | ... | x | y | z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
if $x = 0$ then if $y = 0$ then $z := 1$ else $w := 2$

In the new grammar there is only one parse tree!
Arithmetic Expressions

Here is a grammar for arithmetic expressions:

\[
<\text{expr}> ::= <\text{expr}> + <\text{expr}> \mid <\text{expr}> - <\text{expr}> \mid <\text{expr}> * <\text{expr}> \mid <\text{expr}> / <\text{expr}> \mid <\text{var}> \mid <\text{num}>
\]

\[
<\text{var}> ::= a \mid b \mid c \mid \ldots \mid x \mid y \mid z
\]

\[
<\text{num}> ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

Using this grammar, how would we parse: \(x + 3 \ast y\)?
Two Parse Trees

Two Parse Trees

\[
\begin{align*}
\text{<expr>} & + \text{<expr>} \\
\text{<var>} & \\
\text{x} & \\
\text{<expr>} & \times \text{<expr>} \\
\text{<var>} & \\
\text{<num>} & \\
3 & \\
\text{<expr>} & \times \text{<expr>} \\
\text{<var>} & \\
\text{<num>} & \\
3 & \\
\end{align*}
\]
Precedence

Modify the grammar to add precedence:

\[
\begin{align*}
<expr> &::= <expr> + <expr> | <expr> - <expr> | <term> \\
<term> &::= <term> * <term> | <term> / <term> | <factor> \\
<factor> &::= <var> | <num> | ( <expr> ) \\
<var> &::= a | b | c | ... | x | y | z \\
<num> &::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\end{align*}
\]

Using this grammar, how would we parse: \(x + 3 * y\) ?
Using this grammar, how would we parse: \(7 - 4 - 2\) ?
Only One Parse Tree

But there are two parse trees for the second example:

```
<e>
  -  2
    7  -  <e>
      4  -  2
```
Modify the grammar to add associativity:

\[
<expr> ::= <expr> + <term> \mid <expr> - <term> \mid <term>
\]

\[
<term> ::= <term> \ast <factor> \mid <term> / <factor> \mid <factor>
\]

\[
<factor> ::= <var> \mid <num> \mid ( <expr> )
\]

\[
<var> ::= a \mid b \mid c \mid \ldots \mid x \mid y \mid z
\]

\[
<num> ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

Using this grammar, how would we parse: 7 - 4 - 2?
Only One Parse Tree

expr
  /   \  
expr   -   term
  /   \    \   
factor  num
     2
expr - term
  /    
term  factor
  /    
factor num
     4
     7

num

expr

Concrete vs. Abstract Syntax

\[
\begin{align*}
\langle \text{expr} \rangle & \quad \langle \text{term} \rangle \\
\langle \text{term} \rangle & \quad \langle \text{factor} \rangle \\
\langle \text{factor} \rangle & \quad \langle \text{var} \rangle \\
\langle \text{var} \rangle & \quad x
\end{align*}
\]

\[
\begin{align*}
\langle \text{expr} \rangle & \quad + \\
\langle \text{term} \rangle & \quad \langle \text{term} \rangle \\
\langle \text{term} \rangle & \quad \langle \text{factor} \rangle \\
\langle \text{factor} \rangle & \quad \langle \text{var} \rangle \\
\langle \text{var} \rangle & \quad y
\end{align*}
\]

Abstract Syntax

\[
\begin{align*}
+ & \quad x \\
* & \quad 3 \quad y
\end{align*}
\]
Extended BNF (EBNF)

Write nonterminals as in BNF. (Variant: Write them with initial capital letters, or using a different font.)

Use additional metasymbols, as shortcuts:

- {...} means repeat the enclosed text zero or more times
- [...] means the enclosed text is optional
- (...) is used for grouping, usually with the alternation symbol, e.g., (... | ...).

If { }, [ ], or ( ) are used as terminal symbols in the language being defined, then they must be quoted. (Variant: They must be underlined.)
Extended BNF (EBNF)

Examples:

\[ <\text{expr}> ::= <\text{term}> \{ ( + | - ) <\text{term}> \} \]

\[ <\text{term}> ::= <\text{factor}> \{ ( * | / ) <\text{factor}> \} \]

\[ <\text{factor}> ::= <\text{var}> \mid <\text{num}> \mid \text{‘(’}<\text{expr}>\text{‘)} \]

\[ <\text{if-stmt}> ::= \text{if}<\text{expr}>\text{then}<\text{stmt}>\ [\text{else}<\text{stmt}>\ ] \]

\[ <\text{identifier}> ::= <\text{letter}> \{ ( <\text{letter}> \mid <\text{digit}> ) \} \]