Syntax – Intro and Overview

CS331

Syntax

- Syntax defines what is grammatically valid in a programming language
  - Set of grammatical rules
  - E.g. in English, a sentence cannot begin with a period
  - Must be formal and exact or there will be ambiguity in a programming language
- We will study three levels of syntax
  - Lexical
    - Defines the rules for tokens: literals, identifiers, etc.
  - Concrete
    - Actual representation scheme down to every semicolon, i.e. every lexical token
  - Abstract
    - Description of a program’s information without worrying about specific details such as where the parentheses or semicolons go
BNF Grammar

• BNF = Backus-Naur Form to specify a grammar
  – Equivalent to a context free grammar
• Set of rewriting rules (a rule that can be applied multiple times) defined on a set of nonterminal symbols, a set of terminal symbols, and a start symbol
  – Terminals, $\Sigma$: Basic alphabet from which programs are constructed. E.g., letters, digits, or keywords such as “int”, “main”, “{”, “}”
  – Nonterminals, $N$: Identify grammatical categories
  – Start Symbol: One of the nonterminals which identifies the principal category. E.g., “Sentence” for English, “Program” for a programming language

Rewriting Rules

• Rewriting Rules, $\rho$
  – Written using the symbols $\rightarrow$ and $|$
    $|$ is a separator for alternative definitions, i.e. “OR”
    $\rightarrow$ is used to define a rule, i.e. “IS”
  – Format
    • LHS $\rightarrow$ RHS1 | RHS2 | RHS3 | …
    • LHS is a single nonterminal
    • RHS is any sequence of terminals and nonterminals
Sample Grammars

- Grammar for subset of English
  \[ \text{Sentence} \rightarrow \text{Noun Verb} \]
  \[ \text{Noun} \rightarrow \text{Jack | Jill} \]
  \[ \text{Verb} \rightarrow \text{eats | bites} \]
- Grammar for a digit
  \[ \text{Digit} \rightarrow 0 1 2 3 4 5 6 7 8 9 \]
- Grammar for signed integers
  \[ \text{SignedInteger} \rightarrow \text{Sign Integer} \]
  \[ \text{Sign} \rightarrow + | - \]
  \[ \text{Integer} \rightarrow \text{Digit | Digit Integer} \]
- Grammar for subset of Java
  \[ \text{Assignment} \rightarrow \text{Variable} = \text{Expression} \]
  \[ \text{Expression} \rightarrow \text{Variable | Variable + Variable | Variable – Variable} \]
  \[ \text{Variable} \rightarrow \text{X | Y} \]

Derivation

- Process of parsing data using a grammar
  - Apply rewrite rules to non-terminals on the RHS of an existing rule
  - To match, the derivation must terminate and be composed of terminals only
- Example
  \[ \text{Digit} \rightarrow 0 1 2 3 4 5 6 7 8 9 \]
  \[ \text{Integer} \rightarrow \text{Digit | Digit Integer} \]
  - Is 352 an Integer?
    \[ \text{Integer} \rightarrow \text{Digit Integer} \rightarrow \text{3 Integer} \rightarrow \]
    \[ 3 \text{ Digit Integer} \rightarrow 3 5 \text{ Integer} \rightarrow \]
    \[ 3 5 \text{ Digit} \rightarrow 3 5 2 \]

Intermediate formats are called **sentential forms**
This was called a Leftmost Derivation since we replaced the leftmost nonterminal symbol each time (could also do Rightmost)
Derivation and Parse Trees

• The derivation can be visualized as a parse tree

```
    Integer
     /   \
Digit  Integer
    /     / \
  3     Digit Integer
       /     /   \ 
       5     Digit
         / \
        2
```

Parse Tree Sketch for Programs

```
Program

void main (...) {
    Declarations
    Statements
}

Declaration

Type
    int

Identifiers
    x

Assignment

Statement

Expression
    Literal
    }
```
BNF and Languages

- The **language** defined by a BNF grammar is the set of **all** strings that can be derived
  - Language can be infinite, e.g. case of integers
- A language is **ambiguous** if it permits a string to be parsed into two separate parse trees
  - Generally want to avoid ambiguous grammars
  - Example:
    - Expr → Integer | Expr + Expr | Expr * Expr | Expr - Expr
    - Parse: 3*4+1
      - Expr * Expr → Integer * Expr →
      - 3 * Expr → 3 * Expr + Expr → ... 3 * 4 + 1
      - Expr + Expr → Expr + Integer → Expr + 1
      - Expr * Expr +1 → ... 3 * 4 + 1

Ambiguity

- Example for
  - AmbExp → Integer | AmbExp – AmbExp
  - 2-3-4

![Parse Tree Diagram](attachment:image.png)
Ambiguous IF Statement

Dangling ELSE:

\[
\text{if (x<0) }
\]
\[
\text{if (y<0) \{ y=y-1 \}}
\]
\[
\text{else \{ y=0 \};}
\]

Does the else go with the first or second if?

\[
\text{IfStatement} \rightarrow \text{if (Expression) Statement} \mid \\
\text{if (Expression) Statement else Statement}
\]

\[
\text{Statement} \rightarrow \text{Assignment} \mid \text{IfStatement}
\]

Dangling Else Ambiguity

![Diagram of ambiguity in dangling else statement]
How to fix ambiguity?

• Use explicit grammar without ambiguity
  – E.g., add an “ENDIF” for every “IF”
  – Java makes a separate category for if-else vs. if:
    \[\text{IfThenStatement} \rightarrow \text{If (Expr) Statement}\]
    \[\text{IfThenElseStatement} \rightarrow \text{If (Expr) StatementNoShortIf else Statement}\]
    \[\text{StatementNoShortIf} \text{ contains everything except}\]
    \[\text{IfThenStatement}, \text{ so the else always goes with the}\]
    \[\text{IfThenElse statement not the IfThenStatement}\]

• Use precedence on symbols

Alternative to BNF

• The use of regular expressions is an alternate way to express a language

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>A character (stands for itself)</td>
</tr>
<tr>
<td>&quot;xyz&quot;</td>
<td>A literal string (stands for itself)</td>
</tr>
<tr>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>M N</td>
<td>M followed by N (concatenation)</td>
</tr>
<tr>
<td>M*</td>
<td>Zero or more occurrences of M</td>
</tr>
<tr>
<td>M+</td>
<td>One or more occurrences of M</td>
</tr>
<tr>
<td>M?</td>
<td>Zero or one occurrence of M</td>
</tr>
<tr>
<td>[a-zA-Z]</td>
<td>Any alphabetic character</td>
</tr>
<tr>
<td>[0-9]</td>
<td>Any digit</td>
</tr>
<tr>
<td>.</td>
<td>Any single character</td>
</tr>
</tbody>
</table>
Regex to EBNF

- The book uses some deviations from “standard” regular expressions in Extended Backus Naur Format (defined in a few slides)
  - `{ M }` means zero or more occurrences of M
  - `( M | N)` means one of M or N must be chosen
  - `[ M ]` means M is optional

Use “{“ to mean the literal { not the regex {

RegEx Examples

- Booleans
  - “true” | “false”
- Integers
  - `(0-9)+`
- Identifiers
  - `(a-zA-Z){a-zA-Z0-9}`
- Comments (letters/space only)
  - “/”`(a-zA-Z )“(\n” | “\n” | “\n\n”)
- Regular expressions seem pretty powerful
  - Can you write one for the language a^n b^n? (i.e. n a’s followed by n b’s)
Extended BNF

- EBNF – variation of BNF that simplifies specification of recursive rules using regular expressions in the RHS of the rule
- Example:
  - BNF rule
    \[
    \begin{align*}
    \text{Expr} & \rightarrow \text{Term} | \text{Expr} + \text{Term} | \text{Expr} - \text{Term} \\
    \text{Term} & \rightarrow \text{Factor} | \text{Term} \times \text{Factor} | \text{Term} / \text{Factor}
    \end{align*}
    \]
  - EBNF equivalent
    \[
    \begin{align*}
    \text{Expr} & \rightarrow \text{Term} \{ [+|-] \text{Term} \} \\
    \text{Term} & \rightarrow \text{Factor} \{ [*|/] \text{Factor} \}
    \end{align*}
    \]
- EBNF tends to be shorter and easier to read

EBNF

- Consider:
  \[
  \begin{align*}
  \text{Expr} & \rightarrow \text{Term} \{ [+|-] \text{Term} \} \\
  \text{Term} & \rightarrow \text{Factor} \{ [*|/] \text{Factor} \} \\
  \text{Factor} & \rightarrow \text{Identifier} | \text{Literal} | (\text{Expr})
  \end{align*}
  \]
Parse for X+2*Y
BNF and Lexical Analysis

• **Lexicon** of a programming language – set of all nonterminals from which programs are written
• Nonterminals – referred to as **tokens**
  – Each token is described by its **type** (e.g. identifier, expression) and its **value** (the string it represents)
  – Skipping whitespace or comments

Categories of Lexical Tokens

• Identifiers
• Literals
  Includes Integers, true, false, floats, chars
• Keywords
  bool char else false float if int main true while
• Operators
  = || && == != < <= > >= + - * / % ! [ ]
• Punctuation
  ; . { } ()

Issues to consider: Ignoring comments, role of whitespace, distinguishing the < operator from <=, distinguishing identifiers from keywords like “if”
A Simple Lexical Syntax for a Small Language, Clite

Primary → Identifier [ "["Expression"]" ] | Literal [ "("Expression")" ] | Type "("Expression")"

Identifier → Letter { Letter | Digit }
Letter → a | b | … | z | A | B | … | Z
Digit → 0 | 1 | 2 | … | 9
Literal → Integer | Boolean | Float | Char
Integer → Digit { Digit }
Boolean → true | false
Float → Integer . Integer
Char → ‘ ASCIICHAR ‘

Major Stages in Compilation

• Lexical Analysis
  – Translates source into a stream of Tokens, everything else discarded
• Syntactic Analysis
  – Parses tokens, detects syntax errors, develops abstract representation
• Semantic Analysis
  – Analyze the parse for semantic consistency, transform into a format the architecture can efficiently run on
• Code Generation
  – Use results of abstract representation as a basis for generating executable machine code
Lexical Analysis & Compiling Process

Difficulties:
- 1 to many mapping from HL source to machine code
- Translation must be correct
- Translation should be efficient

Lexical Analysis of Clite

- Lexical Analysis – transforms a program into tokens (type, value). The rest is tossed.
- Example Clite program:

```c
// Simple Program
int main() {
  int x;
  x = 3;
}
```

Result of Lexical Analysis:
Lexical Analysis (2)

Result of Lexical Analysis:

1 Type: Int Value: int
2 Type: Main Value: main
3 Type: LeftParen Value: (
4 Type: RightParen Value: )
5 Type: LeftBrace Value: {
6 Type: Int Value: int
7 Type: Identifier Value: x
8 Type: Semicolon Value: ;
9 Type: Identifier Value: x
10 Type: Assign Value: =
11 Type: IntLiteral Value: 3
12 Type: Semicolon Value: ;
13 Type: RightBrace Value: }
14 Type: Eof Value: <<EOF>>

Lexical Analysis of Clite in Java

```java
public class TokenTester {
    public static void main (String[] args) {
        Lexer lex = new Lexer (args[0]);
        Token t;
        int i = 1;
        do {
            t = lex.next();
            System.out.println(i++ " Type: " + t.type() + "\tValue: " + t.value());
        } while (t != Token.eofTok);
    }
}
```

The source code for how the Lexer and Token classes are arranged is the topic of chapter 3.
Lexical to Concrete

- From the stream of tokens generated by our lexical analyzer we can now parse them using a concrete syntax

Concrete EBNF Syntax for Clite

Program \rightarrow int main ( ) \{ Declarations Statements \}
Declarations \rightarrow \{ Declaration \}
Declaration \rightarrow Type Identifier [ "["Integer"]" ] \{ , Identifier ["["Integer"]"] \};
Type \rightarrow int | bool | float | char
Statements \rightarrow\{ Statement \}
Statement \rightarrow ; | Block | Assignment | IfStatement | WhileStatement
Block \rightarrow \{ Statements \}
Assignment \rightarrow Identifier ["["Expression"]"] = Expression ;
IfStatement \rightarrow if "(" Expression ")" Statement [ else Statement ]
WhileStatement \rightarrow while "("Expression")" Statement

Concrete Syntax;
Higher than lexical syntax!
Concrete EBNF Syntax for Clite

Expression → Conjunction [ || Conjunction ]
Conjunction → Equality [ && Equality ]
Equality → Relation [ EquOp Relation ]
EquOp → == | !=
Relation → Addition [ RelOp Addition ]
RelOp → < | <= | > | >=
Addition → Term [ AddOp Term ]
AddOp → + | -
Term → Factor [ MulOp Factor ]
MulOp → * | / | %
Factor → [ UnaryOp ] Primary
UnaryOp → - | !
Primary → Identifier [ "["Expression"]" ] | Literal | "("Expression")" | Type "(" Expression ")"

Syntax Diagram

• Alternate way to specify a language
• Popularized with Pascal
• Not any more powerful than BNF, EBNF, or regular expressions
Linking Syntax and Semantics

• What we’ve described so far has been concrete syntax
  – Defines all parts of the language above the lexical level
    • Assignments, loops, functions, definitions, etc.
    • Uses BNF or variant to describe the language
  • An abstract syntax links the concrete syntax to the semantic level

Abstract Syntax

• Defines essential syntactic elements without describing how they are concretely constructed
• Consider the following Pascal and C loops

\[
\begin{align*}
\text{Pascal} & \quad \text{C} \\
\text{while } i < n \text{ do begin} & \quad \text{while } (i < n) \{ \\
& \quad i := i + 1 \\
& \quad \text{end} & \quad i = i + 1; \\
& \quad \} \\
\end{align*}
\]

Small differences in concrete syntax; identical abstract construct
Abstract Syntax Format

• Defined using rules of the form
  – LHS = RHS
    • LHS is the name of an abstract syntactic class
    • RHS is a list of essential components that define the class
      – Similar to defining a variable. Data type or abstract syntactic class, and name
      – Components are separated by ;

• Recursion naturally occurs among the definitions as with BNF

Abstract Syntax Example

• Loop
  Loop = Expression test ; Statement body
  – The abstract class Loop has two components, a test which is a member of the abstract class Expression, and a body which is a member of an abstract class Statement

• Nice by-product: If parsing abstract syntax in Java, it makes sense to actually define a class for each abstract syntactic class, e.g.
  class Loop extends Statement {
    Expression test;
    Statement body;
  }
Abstract Syntax of Clite

Program = Declarations decpart; Statements body;
Declarations = Declaration*
Declaration = VariableDecl | ArrayDecl
VariableDecl = Variable v; Type t
ArrayDecl = Variable v; Type t; Integer size
Type = int | bool | float | char
Statements = Statement*
Statement = Skip | Block | Assignment |
Conditional | Loop
Skip =
Block = Statements
Conditional = Expression test;
Statement thenbranch, elsebranch
Loop = Expression test; Statement body
Assignment = VariableRef target; Expression source
Expression = VariableRef | Value | Binary | Unary

VariableRef = Variable | ArrayRef
Binary = Operator op; Expression term1, term2
Unary = UnaryOp op; Expression term
Operator = BooleanOp | RelationalOp | ArithmeticOp
BooleanOp = && | ||
RelationalOp = = | ! | != | < | <= | > | >=
ArithmeticOp = + | - | * | /
UnaryOp = ! | -
Variable = String id
ArrayRef = String id; Expression index
Value = IntValue | BoolValue | FloatValue | CharValue
IntValue = Integer intValue
FloatValue = Float floatValue
BoolValue = Boolean boolValue
CharValue = Character charValue
Java Abstract Syntax for Clite

class Loop extends Statement {
    Expression test;
    Statement body;
}

Class Assignment extends Statement {
    // Assignment = Variable target; Expression source
    Variable target;
    Expression source;
}

... 

Much more… see the file (when available)

Abstract Syntax Tree

- Just as we can build a parse tree from a BNF grammar, we can build an abstract syntax tree from an abstract syntax

- Example for:  x+2*y
  
Expression = Variable | Value | Binary

Binary = Operator op ; Expression term1, term2

![Abstract Syntax Tree Diagram](image-url)
Sample Clite Program

- Compute nth fib number

```c
// compute result = the nth Fibonacci number
void main () {
    int n, fib0, fib1, temp, result;
    n = 8;
    fib0 = 0;
    fib1 = 1;
    while (n > 0) {
        temp = fib0;
        fib0 = fib1;
        fib1 = fib0 + temp;
        n = n - 1;
    }
    result = fib0;
}
```

Abstract Syntax for Loop of Clite Program
Concrete and Abstract Syntax

• Aren’t the two redundant?
  – A little bit
• The concrete syntax tells the programmer exactly what to write to have a valid program
• The abstract syntax allows valid programs in two different languages to share common abstract representations
  – It is closer to semantics
  – We need both!

What’s coming up?

• Semantic analysis
  – Do the types match? What does this mean?
    char a='c';
    int sum=0;
    sum = sum = a;
• Can associate machine code with the abstract parse
  – Code generation
  – Code optimization