Specifying Syntax

Language Specification
Syntax
Form of phrases
Physical arrangement of symbols
Semantics
Meaning of phrases
Result of executing a program
Pragmatics
Usage of the language
Features of implementations

Components of a Grammar
1. Terminal symbols or terminals, $\Sigma$
2. Nonterminal symbols or syntactic categories, $N$
   Vocabulary $= \Sigma \cup N$
3. Productions or rules, $\alpha ::= \beta$, where $\alpha, \beta \in (\Sigma \cup N)^*$
4. Start Symbol (a nonterminal)

Types of Grammars
Type 0: Unrestricted grammars
$\alpha ::= \beta$
where $\alpha$ contains a nonterminal
Type 1: Context-sensitive grammars
$\alpha B \gamma ::= \alpha \beta \gamma$
Type 2: Context-free grammars
$A ::= \text{string of vocabulary symbols}$
Type 3: Regular grammars
$A ::= a$ or $A ::= aB$

An English Grammar
<sentence> ::= <noun phrase> <verb phrase>

<noun phrase> ::= <determiner> <noun> | <determiner> <noun> <prepositional phrase>

<verb phrase> ::= <verb> | <verb> <noun phrase> | <verb> <noun phrase> <prepositional phrase>

<prepositional phrase> ::= <preposition> <noun phrase>

<noun> ::= boy | girl | cat | telescope | song | feather
<determiner> ::= a | the
<verb> ::= saw | touched | surprised | sang
<preposition> ::= by | with
Concrete Syntax for Wren

Phrase-structure syntax:

<program> ::= program <ident> is <block>
<brack> ::= <declaration seq> begin <cmd seq> end
<brack> ::= ε
<declaration> ::= var <var list> : <type> ;
<type> ::= integer | boolean
<var list> ::= <var> | <var>, <var list>
<cmd seq> ::= <cmd> | <cmd> ; <cmd seq>
<cmd> ::= <var> ::= <expr> | skip
| read <var> | write <int expr>
| while <bool expr> do <cmd seq> end while
| if <bool expr> then <cmd seq> end if
| if <bool expr> then <cmd seq> else <cmd seq> end if

Lexical syntax:

<relation> ::= <= | < | = | > | >= | ~> <weak op> <strong op>
<weak op> ::= + | –
<strong op> ::= * | /
<ident> ::= <letter> | <ident> <letter>
| <ident> <digit>
<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m
| n | o | p | q | r | s | t | u | v | w | x | y | z
<numeral> ::= <digit> | <digit> <numeral>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Tokens:

<token> ::= <identifier> | <numeral>
| <reserved word> | <relation>
| <weak op> | <strong op>
| := | ( | ) , , ; : 

<reserved word> ::= program | is | begin | end
| var | integer | boolean | read
| write | skip | while | do | if | then
| else | and | or | true | false | not

A Derivation (for Figure 1.4)

<sentence> ⇒ <noun phrase> <verb phrase>.
⇒ <determiner> <noun> <verb phrase>.
⇒ the <noun> <verb phrase>.
⇒ the girl <verb phrase>.
⇒ the girl touched <noun phrase>.
⇒ the girl touched <determiner>
<noun> <prep phrase>.
⇒ the girl touched the <noun> <prep phrase>.
⇒ the girl touched the cat <preposition>
<noun phrase> .
⇒ the girl touched the cat with <noun phrase> .
⇒ the girl touched the cat with <determiner> <noun> .
⇒ the girl touched the cat with a <noun> .
⇒ the girl touched the cat with a feather .

Note example of a context-sensitive grammar.
Sample Wren Program

```wren
program prime is
    var num, divisor : integer
    var done : boolean
    begin
        read num;
        while num > 0 do
            divisor := 2;  done := false
            while divisor <= num/2 and not done do
                done := num = divisor*(num/divisor);
                divisor := divisor + 1
            end while
            if done then write 0
            else write num
        end if
        read num
    end while
end
```

Ambiguity

One phrase has two different derivation trees, thus two different syntactic structures.

Ambiguous structure may lead to ambiguous semantics.

Dangling “else”

Associativity of expressions (ex 2, page 16)

Ada:

Is “a(2)” a subscripted variable
or a function call?

Classifying Errors in Wren

Context-free syntax — specified by concrete syntax
- lexical syntax
- phrase-structure syntax

Context-sensitive syntax (context constraints)
- also called “static semantics”

Semantic or dynamic errors

Context Constraints in Wren

1. The program name identifier may not be declared elsewhere in the program.

2. All identifiers that appear in a block must be declared in that block.

3. No identifier may be declared more than once in a block.

4. The identifier on the left side of an assignment command must be declared as a variable, and the expression on the right must be of the same type.

5. An identifier occurring as an (integer) element must be an integer variable.

6. An identifier occurring as a Boolean element must be a Boolean variable.

7. An identifier occurring in a `read` command must be an integer variable.

Syntax of Wren

- `<token>^` — arbitrary strings of tokens

- `<program>` — strings derived from the context-free grammar

Syntactically well formed Wren programs derived from `<program>` and also satisfying the context constraints

Semantic Errors in Wren

1. An attempt is made to divide by zero.

2. A variable that has not been initialized is accessed.

3. A `read` command is executed when input file empty.

4. An iteration command (`while`) does not terminate.
Regular Expressions Denote Languages

- \( \emptyset \)
- \( \varepsilon \)
- \( a \) for each \( a \in \Sigma \)
- \( (E \mid F) \) union
- \( (E \cdot F) \) concatenation
- \( (E^*) \) Kleene closure

May omit some parentheses following the precedence rules

Abbreviations

\[ E_1 E_2 = E_1 \cdot E_2 \]
\[ E^+ = E \cdot E^* \]
\[ E^n = E \cdot E \cdot \ldots \cdot E, \text{ repeated } n \text{ times} \]
\[ E^? = \varepsilon \mid E \]

Note: \( E^* = \varepsilon \mid E_1 \mid E_2 \mid E_3 \mid \ldots \)

Nonessential Ambiguity

Command Sequences:

\[
<\text{cmd seq}> ::= <\text{command}> \mid <\text{cmd seq}> ; <\text{cmd seq}>
\]

\[
<\text{cmd seq}> ::= \begin{align*}
&<\text{command}> \mid <\text{cmd seq}> ; <\text{command}> \\
&<\text{cmd seq}> := <\text{command}> ( ; <\text{command}> )^*
\end{align*}
\]

Or include command sequences with commands:

\[
<\text{command}> ::= <\text{cmd seq}> ; <\text{command}> \mid \text{skip} \mid \text{read}<\text{var}> \mid \ldots
\]

Language Processing:

\[
\text{scan} : \text{Character}^* \rightarrow \text{Token}^* \]
\[
\text{parse} : \text{Token}^* \rightarrow ????
\]

Concrete Syntax (BNF)

- Specifies the physical form of programs.

- Guides the parser in determining the phrase structure of a program.

- Should not be ambiguous.

- A derivation proceeds by replacing one nonterminal at a time by its corresponding right-hand side in some production for that nonterminal.

- A derivation following a BNF definition of a language can be summarized in a derivation (parse) tree.

- Although a parser may not produce a derivation tree, the structure of the tree is embodied in the parsing process.

read \( m \); if \( m=9 \) then \( m:=0 \) end if; write 22*\( m \)

Derivation Tree
Abstract Syntax

- Representation of the structure of language constructs as determined by the parser.
- The hierarchical structure of language constructs can be represented as abstract syntax trees.
- The goal of abstract syntax is to reduce language constructs to their “essential” properties.
- Specifying the abstract syntax of a language means providing the possible templates for the various constructs of the language.
- Abstract syntax generally allows ambiguity since parsing has already been done.
- Semantic specifications based on abstract syntax are much simpler.

Designing Abstract Syntax

Consider concrete syntax for expressions:

\[ 
<\text{expr}> ::= <\text{int expr}> | <\text{bool expr}> 
\]

\[ 
<\text{integer expr}> ::= <\text{term}> 
| <\text{int expr}> <\text{weak op}> <\text{term}> 
\]

\[ 
<\text{term}> ::= <\text{element}> 
| <\text{term}> <\text{strong op}> <\text{element}> 
\]

\[ 
<\text{element}> ::= <\text{numeral}> | <\text{var}> 
| ( <\text{int expr}> ) | - <\text{element}> 
\]

\[ 
<\text{bool expr}> ::= <\text{bool term}> 
| <\text{bool expr}> \text{ or } <\text{bool term}> 
\]

\[ 
<\text{bool term}> ::= <\text{bool element}> 
| <\text{bool term}> \text{ and } <\text{bool element}> 
\]

\[ 
<\text{bool element}> ::= \text{true} | \text{false} | <\text{var}> 
| <\text{comparison}> | ( <\text{bool expr}> ) 
| \text{not}( <\text{bool expr}> ) 
\]

\[ 
<\text{comparison}> ::= <\text{int expr}> <\text{relation}> <\text{int expr}> 
\]

Abstract Syntax:

\[ 
\text{Expression} ::= \text{Numeral} | \text{Identifier} | \text{true} | \text{false} 
| \text{Expression} \text{ Operator } \text{Expression} 
| - \text{Expression} | \text{not}(\text{Expression}) 
\]

\[ 
\text{Operator} ::= + | - | * | / | \text{or} | \text{and} 
| <= | < | = | > | >= | <> 
\]

Assume we only have to deal with programs that are syntactically correct.

What basic forms can an expression take?

\[ 
<\text{int expr}> <\text{weak op}> <\text{term}> 
<\text{term}> <\text{strong op}> <\text{element}> 
<\text{numeral}> <\text{variable}> 
- <\text{element}> 
<\text{bool expr}> \text{ or } <\text{bool term}> 
<\text{bool term}> \text{ and } <\text{bool element}> 
\]

\[ 
\text{true} \quad \text{false} 
\]

\[ 
\text{not}( <\text{bool expr}> ) 
\]

\[ 
<\text{int expr}> <\text{relation}> <\text{int expr}> 
\]
Abstract Syntax

Abstract Syntactic Categories

<table>
<thead>
<tr>
<th>Program</th>
<th>Type</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>Command</td>
<td>Numeral</td>
</tr>
<tr>
<td>Declaration</td>
<td>Expression</td>
<td>Identifier</td>
</tr>
</tbody>
</table>

Abstract Production Rules

Program ::= program Identifier is Block
Block ::= Declaration* begin Command+ end
Declaration ::= var Identifier+ : Type ;
Type ::= integer | boolean
Command ::= Identifier ::= Expression | skip
| read Identifier | write Expression
| while Expression do Command+ |
| if Expression then Command+ |
| if Expression then Command+ else Command+ |
Expression ::= Numeral | Identifier | true | false
| Expression Operator Expression
| – Expression | not Expression
Operator ::= + | – | * | / | or | and
| <= | < | = | > | >= | <>

Alternative Abstract Syntax

Abstract Production Rules

Program ::= prog (Identifier, Block)
Block ::= block (Declaration*, Command+)
Declaration ::= dec (Identifier+, Type)
Type ::= integer | boolean
Command ::= assign (Identifier, Expression)
| skip | read (Identifier) | write (Expression)
| while (Expression, Command+)
| if (Expression, Command+)
| ifelse (Expression, Command+, Command+)
Expression ::= Numeral | Identifier | true | false
| not (Expression) | minus (Expression)
| expr (Operator, Expression, Expression)
Operator ::= + | – | * | / | or | and
| <= | < | = | > | >= | <>

Example

read m;
   if m=-9 then m:=0 end if;
   write 22*m

[ read (m),
   if (expr (equal, ide(m), minus (num(9))),
      [ assign (m, num(0)) ]),
   write (expr (times, num(22), ide(m)))
]