

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**, Σ

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** (**nonterminal**)

Types of Grammars

Type 0: **Unrestricted grammars**

$\alpha ::= \beta$

where α contains a nonterminal

Type 1: **Context-sensitive grammars**

$\alpha B \gamma ::= \alpha \beta \gamma$

Type 2: **Context-free grammars**

$A ::=$ string of vocabulary symbols

Type 3: **Regular grammars**

$A ::= a$ or $A ::= aB$

An English Grammar

$\langle \text{sentence} \rangle ::= \langle \text{noun phrase} \rangle \langle \text{verb phrase} \rangle .$

$\langle \text{noun phrase} \rangle ::= \langle \text{determiner} \rangle \langle \text{noun} \rangle$
| $\langle \text{determiner} \rangle \langle \text{noun} \rangle \langle \text{prepositional phrase} \rangle$

$\langle \text{verb phrase} \rangle ::= \langle \text{verb} \rangle$ | $\langle \text{verb} \rangle \langle \text{noun phrase} \rangle$
| $\langle \text{verb} \rangle \langle \text{noun phrase} \rangle \langle \text{prepositional phrase} \rangle$

$\langle \text{prepositional phrase} \rangle ::=$
 $\langle \text{preposition} \rangle \langle \text{noun phrase} \rangle$

$\langle \text{noun} \rangle ::=$ **boy** | **girl** | **cat** | **telescope**
| **song** | **feather**

$\langle \text{determiner} \rangle ::=$ **a** | **the**

$\langle \text{verb} \rangle ::=$ **saw** | **touched** | **surprised** | **sang**

$\langle \text{preposition} \rangle ::=$ **by** | **with**

Sample Wren Program

```
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)

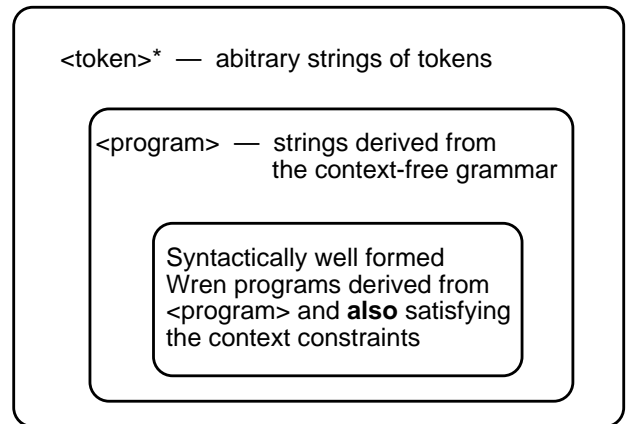
- also called “static semantics”

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



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	
ε	
a	for each $a \in \Sigma$
$(E \mid F)$	union
$(E \bullet F)$	concatenation
(E^*)	Kleene closure

May omit some parentheses following the precedence rules

Abbreviations

$E_1 E_2$	$= E_1 \bullet E_2$
E^+	$= E \bullet E^*$
E^n	$= E \bullet E \bullet \dots \bullet E$, repeated n times
$E^?$	$= \varepsilon \mid E$

Note: $E^* = \varepsilon \mid E^1 \mid E^2 \mid E^3 \mid \dots$

Nonessential Ambiguity

Command Sequences:

$\langle \text{cmd seq} \rangle ::= \langle \text{command} \rangle \mid \langle \text{command} \rangle ; \langle \text{cmd seq} \rangle$

$\langle \text{cmd seq} \rangle ::= \langle \text{command} \rangle \mid \langle \text{cmd seq} \rangle ; \langle \text{command} \rangle$

$\langle \text{cmd seq} \rangle ::= \langle \text{command} \rangle (; \langle \text{command} \rangle)^*$

Or include command sequences with commands:

$\langle \text{command} \rangle ::= \langle \text{command} \rangle ; \langle \text{command} \rangle \mid \text{skip} \mid \text{read} \langle \text{var} \rangle \mid \dots$

Language Processing:

scan : $\text{Character}^* \rightarrow \text{Token}^*$

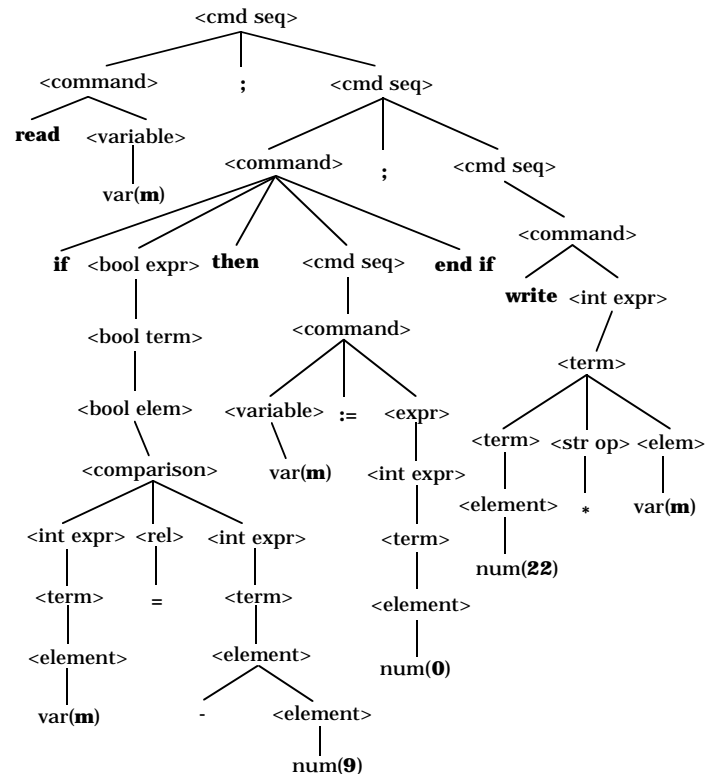
parse : $\text{Token}^* \rightarrow \text{????}$

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

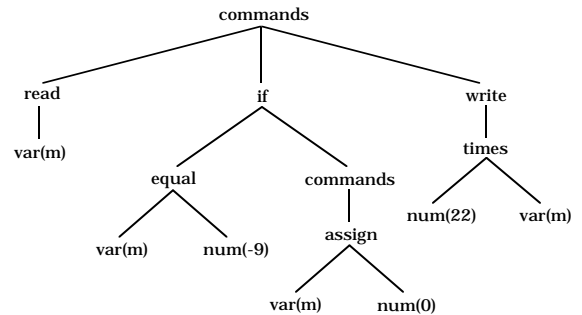


Abstract Syntax

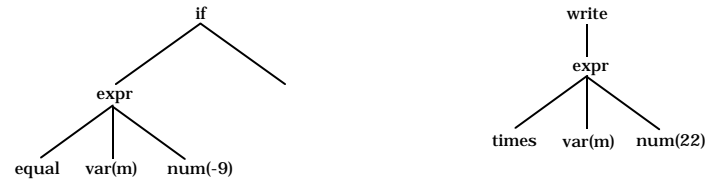
- Representation of the structure of language constructs as determined by the parser.
- The hierarchical structure of language constructs can be represented as abstract syntax trees.
- The goal of abstract syntax is to reduce language constructs to their “essential” properties.
- Specifying the abstract 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.

read m; if m=-9 then m:=0 end if; write 2*m

Abstract Syntax Tree:



OR



Designing Abstract Syntax

Consider concrete syntax for expressions:

```

<expr> ::= <int expr> | <bool expr>
<integer expr> ::= <term>
                | <int expr> <weak op> <term>
<term> ::= <element>
          | <term> <strong op> <element>
<element> ::= <numeral> | <var>
            | ( <int expr> ) | - <element>
<bool expr> ::= <bool term>
              | <bool expr> or <bool term>
<bool term> ::= <bool element>
              | <bool term> and <bool element>
<bool element> ::= true | false | <var>
                 | <comparison> | ( <bool expr> )
                 | not( <bool expr> )
<comparison> ::=
    <int expr> <relation> <int expr>
  
```

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

What basic forms can an expression take?

```

<int expr> <weak op> <term>
<term> <strong op> <element>
<numeral>          <variable>
- <element>
<bool expr> or <bool term>
<bool term> and <bool element>
true          false
not( <bool expr> )
<int expr> <relation> <int expr>
  
```

Abstract Syntax:

```

Expression ::= Numeral | Identifier | true | false
            | Expression Operator Expression
            | - Expression | not(Expression)
  
```

```

Operator ::= + | - | * | / | or | and
           | <= | < | = | > | >= | <>
  
```

Abstract Syntax

Abstract Syntactic Categories

Program	Type	Operator
Block	Command	Numeral
Declaration	Expression	Identifier

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)))  
]
```