Programming Languages

Lecture 5: Language Syntax

Benjamin J. Keller

Department of Computer Science, Virginia Tech

Lecture Outline

- Major Elements of Language
- Defining Syntax
  - Formal Languages
  - Regular Expressions
  - Lexical Analysis
  - Context-Free Grammars
  - Parsing
  - Context-Sensitivity
- Summary

Major Elements of Language

- Syntax — determines what phrases are in language
- Semantics — determines what a phrase in language means
- Pragmatics — how language is used

Defining Syntax

- Two parts of language:
  - "words" of language, *tokens* (ex. "`if`", "`{`")
  - "phrases" of language (ex. `if (x < y) then x++;`)
- Define syntax with mechanisms from formal languages (started with Algol 60)
  - Regular expression — defines tokens
  - Context-free grammar — defines structure of language
- Language recognition
  - Lexer — recognizes tokens
  - Parser — recognizes context-free language

Formal Languages

- Language — a set of strings
- String — a finite sequence of symbols from some alphabet
- Alphabet — a (finite) set of symbols

Regular Expressions

- Way of defining regular language (simplest formal language)
  - Symbols — each symbol stands for itself

    a – denotes language $\{a\}$
  - Empty string — usually written either ϵ or λ

    Note ϵ denotes $\{\epsilon\}$ not empty set
  - Alternation — Given two regular expressions M and N the language is the union of the languages for M and N

    $a|b$ — denotes language $\{a, b\}$

Regular Expressions (cont)

- Concatenation — the language for MN is the concatenation of strings from the language of M with strings from the language of N.

  $a(a|b)$ — denotes language $\{aa, ab\}$

- Repetition — M^* is language of zero or more repetitions of strings in language of M (Kleene closure)

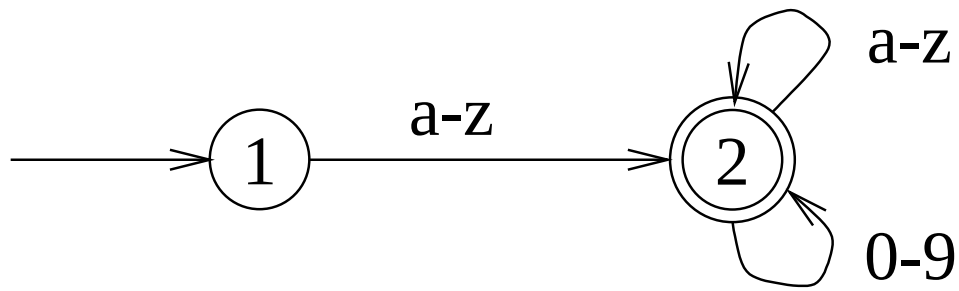
  a^* denotes $\{\epsilon, a, aa, aaa, \ldots\}$
  $(a|b)^*$ denotes $\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$

Recognizing Regular Languages

- Finite automata — recognizes regular languages
  - Set of states
  - Transitions between pairs of states, labeled by symbol
  - Distinguished start state
  - Several final states
- Deterministic Finite Automata — no two transitions from a state have same symbol
- DFA *accepts* a string w if when start in start-state a transition can be followed for each symbol in w from left-to-right until there are no more symbols in w and the automaton is in a final state. Otherwise, the DFA *rejects* w.
- Language of DFA — set of strings accepted

Finite Automata

- Example



- Accepts: a, $a1b$, $counter1$
- Rejects: $1a$, A
- Recognizes language $(a|b|\cdots|z)((a|\cdots|z)|(0|\cdots|9))^*$

Implementing Lexers

- Implement DFAs as tables:
  - next state — lookup next state based on input
  - final action — lookup action based on state (ex action: recognized Identifier)
- Lexer generators — generate code for DFA
  - Take regular expression with actions as input
  - Convert regular expression to nondeterministic FA
  - Simplify NFA to DFA
  - Generated code recognizes tokens and perform actions

Context-Free Grammars

- Set of *productions* like

$$
\begin{array}{lcl}
S & \rightarrow & S\,\texttt{;}\,S \\
S & \rightarrow & \text{id}\,\texttt{:=}E \\
E & \rightarrow & \text{num} \\
E & \rightarrow & E\texttt{+}E
\end{array}
$$

- Terminals — tokens ("id", "num", "`:=`", "`+`", "`;`")
- Nonterminals — symbols used in grammar (S, E)
- Start symbol — a nonterminal
- Derivation — generating phrases of language by replacing nonterminals with r.h.s. of productions

Backus-Naur Form

- Notation typically used for defining languages

```
<statement>  ::= <statement> ; <statement>
<statement>  ::= <identifier> := <expression>
<expression> ::= <number>
<expression> ::= <expression> + <expression>
```

Extended BNF

- Optional syntax
  - BNF

```
<conditional> ::= if <expression> then <statement>
 if <expression> then <statement> else <statement>
```

  - EBNF

```
<conditional> ::= if <expression> then <statement>
                  [else <statement>]
```

- Repeated syntax (zero or more times)
  - BNF

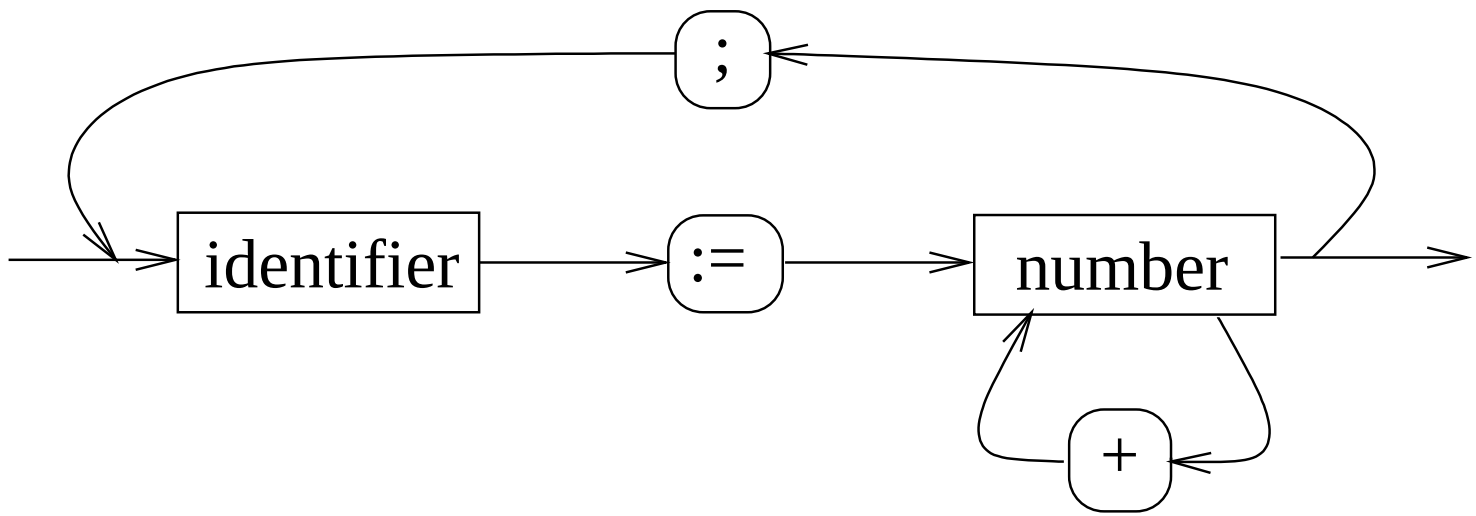
```
<literal> ::= <digit> | <digit> <literal>
```

  - EBNF

```
<literal> ::= <digit> { <digit> }
```

Syntax Diagrams

- Alternative to BNF
- Nonrecursive diagrams



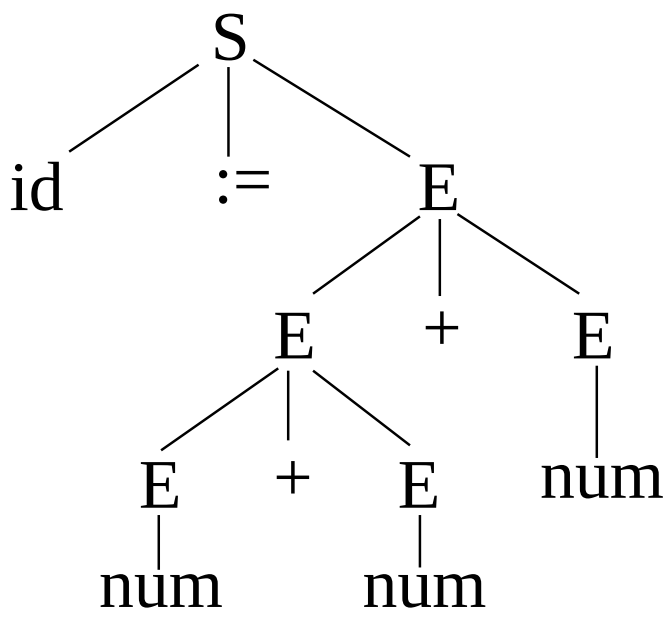
Derivations

- Begin with start symbol, replace any nonterminal with the rhs of some production for that nonterminal
- leftmost — pick leftmost nonterminal to expand
- rightmost — pick rightmost nonterminal to expand
- Example (leftmost)

$$
\begin{aligned}
S &\Rightarrow \text{id}\texttt{:=}E \\
&\Rightarrow \text{id}\texttt{:=}E\texttt{+}E \\
&\Rightarrow \text{id}\texttt{:=}E\texttt{+}E\texttt{+}E \\
&\Rightarrow \text{id}\texttt{:=}\text{num}\texttt{+}E\texttt{+}E \\
&\Rightarrow \text{id}\texttt{:=}\text{num}\texttt{+}\text{num}\texttt{+}E \\
&\Rightarrow \text{id}\texttt{:=}\text{num}\texttt{+}\text{num}\texttt{+}\text{num}
\end{aligned}
$$

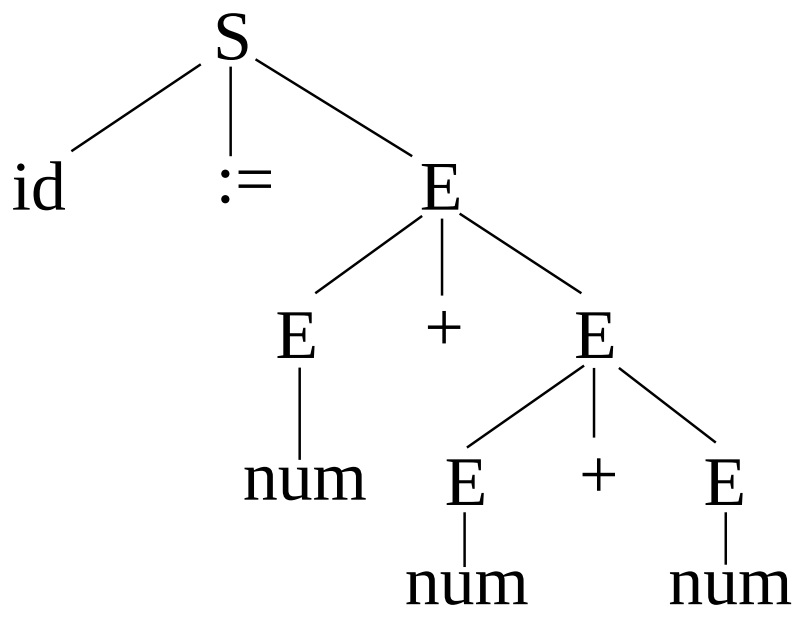
Parse Tree

- A tree representation of a derivation.
- Connect a symbol to the one from which it was derived
- Example



Ambiguity

- A grammar is *ambiguous* if a phrase has two different derivations
- Equivalently, two parse trees
- Example (rightmost derivation)



Problems with Ambiguity

- Problematic grammar

```
<statement> ::= <conditional> | <assignment>
  | begin { <statement> } end
<conditional> ::= if <expression> then <statement>
  | if <expression> then <statement> else <statement>
```

- How do you parse

```
if E1 then if E2 then S1 else S2
```

- Choices

  1. if E1 then (if E2 then S1 else S2)
  2. if E1 then (if E2 then S1) else S2

Resolving Ambiguity

- C, C++, Java, Pascal —
- To get other alternative use "begin", "end"

  `if E1 then ___ if E2 then S1 ___ else S2 ___`
- Modula-2, Algol 68 use an "end" keyword ("fi" in Algol)
  1. if E1 then if E2 then S1 else S2 end end
  2. if E1 then if E2 then S1 end else S2 end
- Ambiguity doesn't occur in SML. Why?

Parsing

- A parser recognizes programming language
- Can be seen as constructing parse tree
- Two approaches
  - Top down — begin with start symbol, use input to decide which production to replace leftmost nonterminal with
  - Bottom up — gather terminals, if recognize rhs of production replace with nonterminal
- Looking ahead in token stream helps deal with ambiguities

Kinds of Parsing algorithms

- Recursive descent — one function for each production
- $LL(k)$ — Left-to-right parse, Leftmost derivation, k symbol lookahead.
- $LR(k)$ — Left-to-right parse, Rightmost derivation, k symbol lookahead, table based
- SLR — simple LR — improvement on $LR(0)$
- $LALR(1)$ — smaller tables than $LR(1)$ — standard for compiler construction
- $LR(0) \subset SLR \subset LALR(1) \subset LR(1) \subset LR(k)$

Context-Sensitivity

- Programming languages usually not exactly context-free
- Declaration before use requires knowledge of context

```
int c = 0;
while (c < 10) c++;
```

- Similar for other constructions where definition is one place and use is another
- Solved by storing semantic information in translators (symbol tables)

Summary

- First use of formal syntax definition in Algol 60 report
- Having a formal definition of programming language syntax allows
  1. the programmer to write (generate) syntactically correct programs, and
  2. a parser to recognize syntactically correct programs.
- Can use formal syntax definitions as input to lexer and parser generators