Chapter 3: Syntax and Semantics

Matt Evett
Dept. Computer Science
Eastern Michigan University

Syntax and Semantics

- Syntax - the *form or structure* of the expressions, statements, and program units
- Semantics - the *meaning* of the expressions, statements, and program units
- Who must use language definitions?
  - Other language designers
  - Implementors
  - Programmers (the users of the language)

Syntax Definitions

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (e.g., *, sum, begin)
- A *token* is a category of lexemes (e.g., identifier)

Using Formal Syntax

- Two general uses of formally defined languages:
  - Recognizers - used in compilers. Given a syntax and a string, is the string sentence of the language?
  - Generators - what we'll study. Given a syntax, generate legal sentences.

Formal Language Description

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-50s
  - Language generators, meant to describe the syntax of natural languages
  - Defined a class of languages called *context-free* languages
- Backus Normal Form (1959)
  - Invented by John Backus to describe Algol 58
  - BNF is equivalent to context-free grammars
  - A metalanguage for computer languages
    - A language used to describe other languages.

BNF

- Abstractions are used to represent classes of syntactic structures
  - Act like syntactic variables (also called **nonterminal** symbols)

<while_stmt> -> while <logic_expr> do <stmt>

- This is a rule describing the structure of a *while* statement

BNF Grammar

- A *grammar* is a finite, nonempty set of rules (*R*), plus sets of terminal (*T*) and nonterminal (*N*) symbols.
- A *rule* has a left-hand side (LHS) and a right-handside (RHS)
  - The LH is a single terminal symbol
  - The RHS consisting of terminal and nonterminal symbols
  - The sets of terminals (*T*) and nonterminals (*N*) are mutually exclusive.
- Nonterminals are indicated with "< ... >"

BNF Rule

- An abstraction (or nonterminal symbol) can have more than one RHS

<stmt> -> <single_stmt>
| begin <stmt_list> end

- BNF rules are often recursive. Ex: a list

<ident_list> -> ident
| ident, <ident_list>

Example Grammar

1. <program> -> <stmts>
2. <stmts> -> <stmt> | <stmt> ; <stmts>
3. <stmt> -> <var> = <expr>
4. <var> -> a | b | c | d
5. <expr> -> <term> + <term> | <term> - <term>
6. <term> -> <var> | const

An example derivation:

- A *derivation* is a repeated application of rules, starting with the *start symbol* (εN)and yielding a sentence (all terminal symbols).

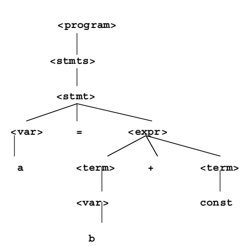
<program> => <stmts> => <stmt> *R1 and R2*
=> <var> = <expr> => a = <expr> *R3, R4*
=> a = <term> + <term> *R5a*
=> a = <var> + <term> *R6a*
=> a = b + <term> *R4*
=> a = b + const *R6b*

Derivations

- Every string of symbols in a derivation is a *sentential form.*
- A *sentence* is a sentential form that has only terminal symbols.
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be leftmost, rightmost or neither.

Parse Tree

A *parse tree* is a hierarchical rep. of a derivation.
A grammar is ambiguous iff it generates a sentential form that has two or more distinct parse trees

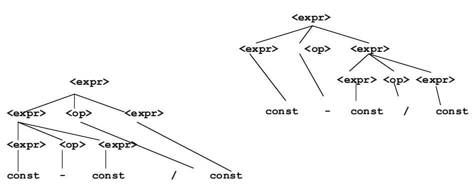


Grammars & Languages

- A language is a set (possibly empty) of strings.
- A grammar, G, *generates* or *defines* a language, L, iff *exactly* those strings comprising L can be derived with G.
  - All elements of L must be derivable with G.
  - There must be no derivations for any strings not in L.

Ex: an ambiguous expression grammar

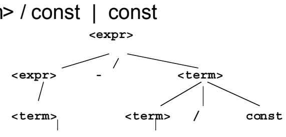
<expr> -> <expr> <op> <expr> | const
<op> -> / | -



Controlling Ambiguity

- Careful tinkering can convert ambiguous languages into equivalent unambiguous ones.
- An unambiguous expression grammar:
  <expr> -> <expr> - <term> | <term>
  <term> -> <term> / const | const

```
<expr> => <expr> - <term>
       => <term> - <term>
       => const - <term>
       => const - <term> / const
       => const - const / const
```



Encoding Precedence

- Suppose evaluation of subexpressions of an arithmatic expression depended on their location within the parse tree; bottom-up

<assgn> --> <id> = <expr>
<id> --> A | B | C
<expr> --> <expr> + <term>
| <term>
<term> --> <term> * <factor>
| <factor>
<factor> --> (<expr>)
| <id>

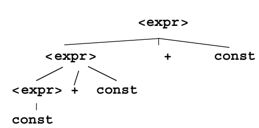
Try parsing A = B * C + A

Encoding Associativity

Operator associativity can also be indicated by a grammar

`<expr> -> <expr> + <expr>   |   const` **(ambiguous)**

`<expr> -> <expr> + const   |   const` **(unambiguous)**



Extended BNF

- Extended BNF (just abbreviations):
- Optional parts are placed in brackets ([])
  - <proc_call> -> ident [(<expr_list>)]
- Put alternative parts of RHSs in parentheses and separate them with vertical bars
  - <term> -> <term> (+ | -) const
- Put repetitions (0 or more) in braces ({})
  - <ident> -> letter {letter | digit}

Example of EBNF

- **BNF:**

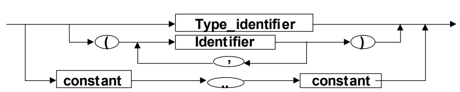
- <expr> -> <expr> + <term>
-         | <expr> - <term>
-         | <term>
- <term> -> <term> * <factor>
-         | <term> / <factor>
-         | <factor>

- **EBNF:**

- <expr> -> <term> {(+ | -) <term>}
- <term> -> <factor> {(* | /) <factor>}

Syntax Graphs

Syntax Graphs - put the terminals in circles or ellipses and put the nonterminals in rectangles;
connect with lines with arrowheads
EX: Pascal type declarations



Recursive Descent Parsing

- *Parsing* is the process of tracing or constructing a parse tree for an input string
- Parsers usually do not analyze lexemes
  – that is done by a *lexical analyzer*, which is called by the parser
- A recursive descent parser traces out a parse tree in top-down order; it is a *top-down parser*
- Each nonterminal in the grammar has a subprogram associated with it; the subprogram parses all sentential forms that the nonterminal can generate

Building Recursive Descent Parser

- Each grammar rule yields one recursive descent parsing subprogram.
- **Example: For the grammar:**

```
<term> -> <factor> {(* | /) <factor>}
```

We could use the following recursive descent parsing subprogram.

```
void term() {
    factor();  /* parse the first factor*/
    while (next_token == ast_code ||
          next_token == slash_code) {
      lexical();  /* get next token */
      factor();  /* parse the next factor */
    }
  }
```


Limitations of RDP

- Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars
- Imagine the code that would derive from:

  <A> --> <A> + <C>

```
void term() {
    A();  /* parse the lhs argument*/
    if (next_token != plus_code)
    { error(); return; }
    lexical();  /* get next token */
      C();  /* parse the rhs */
    }
  }
```


Static Semantics

Static semantics **(have nothing to do with meaning)**

Categories:

1. **Context-free (e.g. type checking), tends to be cumbersome**
2. **Noncontext-free (e.g. variables must be declared before they are used)**

Attribute Grammars

- (Knuth, 1968)
  - Cfgs cannot describe all of the syntax of programming languages
    - E.g. type info
  - Additions to cfgs to carry some semantic info along through parse trees
- Primary value of AGs:
  - Static semantics specification
  - Compiler design(static semantics checking)

Static Semantics

- Information that is difficult to encode with CFG.
- Could be encoded using CSG, but then it is more difficult to generate compilers.
- *Static* because the sentence validity can be checked at compile-time

Define Attribute Grammar

- Def: An attribute grammar is a cfg G = (S, N, T, P) with the following additions:
  - For each grammar symbol x there is a set A(x) of attribute values
  - Each rule has a set of functions that define certain attributes of the nonterminals in the rule
  - Each rule has a (possibly empty) set of *predicates* to check for attribute consistency

AG Components

- Let X0 -> X1 ... Xn be a rule.
- Functions of the form S(X0) = f(A(X1), … A(Xn)) define *synthesized attributes*
- Functions of the form I(Xj) = f(A(X0), ... , A(Xn)), for i <= j <= n, define *inherited attributes*
- Initially, there are intrinsic attributes on the parse tree leaves

Example AG (1)

- Example: expressions of the form id + id
  - id's can be either *int_type* or *real_type*
  - types of the two id's must be the same
  - type of the expression must match it's expected type
- BNF:
  - <expr> -> <var> + <var>
  - <var> -> id
- Attributes:
  - *actual_type* - synthesized for <var> and <expr>
  - *expected_type* - inherited for <expr>

Ex: G and its Attributes

- **The CFG rules may be augmented with "[ ]"**
- **Syntax rule:** <expr> -> <var>[1] + <var>[2]
  - **Semantic rules:**
    - <var>[1].env ← <expr>.env
    - <var>[2].env ← <expr>.env
    - <expr>.actual_type ← <var>[1].actual_type
  - **Predicate:**
    - <var>[1].actual_type = <var>[2].actual_type
    - <expr>.expected_type = <expr>.actual_type
- **Syntax rule:** <var> -> id
  - **Semantic rule:**
    - <var>.actual_type <- lookup (id, <var>.env)

Computing Attributes

- How to compute attributes?
  - If all attributes were inherited, the tree could be *decorated* in top-down order.
  - If all attributes were synthesized, the tree could be decorated in bottom-up order.
  - In most cases, both kinds of attributes are used, requiring a combination of top-down and bottom-up decoration.

Computing Attributes (2)

1. <expr>.env ← inherited from parent
   <expr>.expected_type ← inherited from parent
2. <var>[1].env ← <expr>.env (inherited...)
   <var>[2].env ← <expr>.env
3. <var>[1].actual_type ← lookup (A, <var>[1].env) (synthesized...)
   <var>[2].actual_type ← lookup (B, <var>[2].env)
   <var>[1].actual_type =? <var>[2].actual_type (a predicate)
4. <expr>.actual_type ← <var>[1].actual_type
   <expr>.actual_type =? <expr>.expected_type

Annotate a parse tree

- See the board….

Dynamic Semantics

- No single widely acceptable notation or formalism for describing semantics
  - Operational semantics
  - Axiomatic semantics
  - Denotational semantics

Operational Semantics

-

- Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
- To use operational semantics for a high-level language, a VM in needed
  - A hardware pure interpreter would be too expensive
  - A software pure interpreter also has problems:
    - The detailed characteristics of the particular computer would make actions difficult to

Idealized VM

- A better alternative: A complete computer simulation
- The process:
  1. Build a translator (translates source code to the machine code of an idealized computer)
  2. Build a simulator for the idealized computer

Value of Operational Semantics

- Good if used informally
- Extremely complex if used formally (e.g., VDL)

Axiomatic Semantics

- Based on formal logic (first order predicate calculus)
- Original purpose: formal program verification
- Approach: Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions)
- The expressions are called assertions

Conditions

- An assertion before a statement (a *precondition*) states the relationships and constraints among variables that are true at that point in execution
- An assertion following a statement is a *postcondition*
- A *weakest precondition* is the least restrictive precondition guaranteeing a postcondition
- Pre-post form: {P} statement {Q}
- An example: a := b + 1 {a > 1}
  – One possible precondition: {b > 10}
  – Weakest precondition: {b > 0}

Axioms

- **An axiom for assignment statements:**

- **$\{Q_{x\text{->}E}\}$ x := E {Q}**

- **- The Rule of Consequence:**

$$\frac{\{P\}\ S\ \{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

Sequences

- **- An inference rule for sequences (the Chaining Rule)**
  - **- For a sequence S1;S2:**
  - **{P1} S1 {P2}**
  - **{P2} S2 {P3}**

the inference rule is:

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1;\ S2\ \{P3\}}$$

Axiomatic Proof Process

- Proving program correctness
- Program proof process:
  - The postcondition for the whole program is the desired result. Work back through the program to the first statement, inferring preconditions.
  - If the precondition on the first statement is the same as the program spec, the program is correct.

Inference Rules for Loops

- Very complicated! (Their use, that is.)
- Loop *invariants*
- Interested? See 3.6.2.6

Loops (skip!)

- An inference rule for logical pretest loops
- For the loop construct:
  {P} while B do S end {Q}
- the inference rule is:
  (I and B) S {I}
  {I} while B do S {I and (not B)}
- where *I* is the loop invariant.

Invariants

- Characteristics of the loop invariant, I:
  1. P => I (the invariant must be true initially)
  2. {I} B {I} (evaluation of the Boolean must not change the validity of I)
  3. {I and B} S {I} (I is not changed by executing the body of the loop)
  4. (I and (not B)) => Q (if I is true and B is false, Q is implied)
  5. The loop terminates (this can be difficult to prove)

Invariants (cont.)

- The loop invariant I is a weakened version of the loop postcondition, and it is also a precondition.

- I must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition

Developing Axiomatic Semantics

- Evaluation of axiomatic semantics:

- Developing axioms or inference rules for all of the statements in a language is difficult
- It is a good tool for correctness proofs, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers

Denotational Semantics

- Based on recursive function theory
- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)

Denotational Semantics (2)

- The process of building a denotational spec for a language:
  1. Define a mathematical object for each language entity
  2. Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects
- The meaning of language constructs are defined by only the values of the program's variables
- Meaning is assigned to grammar rules containing only a terminal as the RHS.

Denotational vs. Operational

- The difference between denotational and operational semantics:
  - In operational semantics, the state changes are defined by coded algorithms; in denotational semantics, they are defined by rigorous mathematical functions
- The state of a program is the values of all its current variables

  s = {<i1, v1>, <i2, v2>, …, <in, vn>}
- Let VARMAP be a function that, when given a variable name and a state, returns the current value of the variable

  VARMAP(ij, s) = vj

D.S. for Numbers

1. Decimal Numbers

<dec_num> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
| <dec_num> (0 | 1 | 2 | 3 | 4 |
5 | 6 | 7 | 8 | 9)

M_{dec}('0') = 0, M_{dec} ('1') = 1, …, M_{dec} ('9') = 9
M_{dec} (<dec_num> '0') = 10 * M_{dec} (<dec_num>)
M_{dec} (<dec_num> '1') = 10 * M_{dec} (<dec_num>) + 1
…
M_{dec} (<dec_num> '9') = 10 * M_{dec} (<dec_num>) + 9

D.S. of Numeric Expressions

M_e(<expr>, s) Δ=
case <expr> of
<dec_num> => M_{dec}(<dec_num>, s)
<var> =>
if VARMAP(<var>, s) = undef
then error
else VARMAP(<var>, s)
<binary_expr> =>
if (M_e(<binary_expr>.<left_expr>, s) = undef
OR M_e(<binary_expr>.<right_expr>, s) =
undef)
then error
else
if (<binary_expr>.<operator> = ë+í then
M_e(<binary_expr>.<left_expr>, s) +
M_e(<binary_expr>.<right_expr>, s)
else M_e(<binary_expr>.<left_expr>, s) *
M_e(<binary_expr>.<right_expr>, s)

D.S. Assignments & Loops

M_a(x := E, s) Δ=
if M_e(E, s) = error
then error
else s' = {<i_1',v_1'>,<i_2',v_2'>,...,<i_n',v_n'>},
where for j = 1, 2, ..., n,
v_j' = VARMAP(i_j, s) if i_j <> x
= M_e(E, s) if i_j = x

4 Logical Pretest Loops

M_l(while B do L, s) Δ=
if M_b(B, s) = undef
then error
else if M_b(B, s) = false
then s
else if M_{sl}(L, s) = error
then error
else M_l(while B do L, M_{sl}(L, s))

Loops

- The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors

- In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions

  - Recursion, when compared to iteration, is easier to describe with mathematical rigor

Use of D.S.

- Evaluation of denotational semantics:

- Can be used to prove the correctness of programs
- Provides a rigorous way to think about programs
- Can be an aid to language design
- Has been used in compiler generation systems