

Syntax and Semantics

CS395T - Domain Specific Languages
Greg Lavender
Department of Computer Sciences
The University of Texas at Austin



Designing a Language Core

- Possible objectives
 - domain specific syntax (user friendly :-)
 - general purpose sublanguages (expressions, loops, built-in types)
 - efficient in implementation
 - extensible (e.g., macros)
 - safe/secure (in the sense of static types and run-time checks)
 - orthogonal composition of different features
 - simple and consistent syntax and semantics
 - good tools (editors, debuggers, refactoring, etc)
 - facilitates portability
 - exploits underlying system features (networking, threads, etc.)
- What is a good methodology to follow?
 - Lots of different points of view!
 - Simplicity first, generalize second
 - Orthogonality, utility, extensibility
 - See the paper "Growing a Language" by Guy L. Steele

2/15/05

2



Operational semantics

- semantics of operational steps required to evaluate a program
 - operational semantics are helpful in constructing an interpreter for a language
 - for example, we could implement all of the semantic functions as procedures in an interpreter, augmented with an "environment" for looking up and updating variables (locations) with storage values in RAM
 - The goal then is to reduce a program to some value (its normal form) by a step-by-step evaluation of each syntactic phrase according to the type rules and the semantic equations

2/15/05

3



A REPL Interpreter

- REPL - Read-Eval-Print-Loop
 - First read a program and do lexical analysis and parsing into an internal form (e.g., an abstract syntax tree)
 - Have an "eval" procedure that can reduce any valid expression in the language to its normal form
 - for each expression, the eval procedure applies the semantic function for that expression using the current state of the interpreter as the store
 - Print out the result of the expression
 - Repeat for the next expression

2/15/05

4



Interpreter Computation Steps

- A computation is a sequence of steps
 - $p_0 \Rightarrow p_1 \Rightarrow \dots \Rightarrow p_n, n \geq 0$
 - we say $p_0 \Rightarrow^* p_n$ in zero or more steps
 - if p_n is a value, terminate the computation
 - for Bool, true and false are values
 - for Int, numerals are values
 - for Location, every loc_i is a value
 - for Store, every n_i in a vector is a value, so a vector is a value
 - recall that expressions and commands produce new stores
 - for the core imperative language, a program is some phrase $[[C:comm]]s_0$ starting with the initial store s_0
 - Small-step vs large-step interpretation
 - Reduce an expression piece by piece or chunk by chunk

2/15/05

5



Desired Operational Properties

- subject reduction: if p has an underlying type, τ , and $p \Rightarrow^* p'$, then p' has type τ
- soundness: if p has the underlying mathematical meaning, m , and $p \Rightarrow^* p'$, then p' has the same meaning m
- strong typing: if p is well-typed and $p \Rightarrow^* p'$, then p' has no operator-operand type errors
- computational adequacy: the meaning m of a program p is a proper meaning iff there is a value v such that $p \Rightarrow^* v$ and v has meaning m .

2/15/05

6



Key Interpreter Components

- Lexer & parser for processing the syntax
- Type checker
- Expression evaluator
- Environment data structure
 - environment-passing interpreters
- Closure construction
 - for (static) local scopes
 - for procedures/function and recursion
- Universal eval procedure
 - can evaluate any language construct, recursively
- Control
 - Read, eval, print loop with exceptions for detecting and handling/reporting errors

2/15/05

7



Language Syntax

- Programming language syntax
 - object language syntax vs meta-language syntax
 - *C* is an object language
 - a Lex specification is a meta-language for describing the regular language of words (tokens) for *C*
 - a Yacc LALR(1) grammar for *C* is a meta-language describing *C*'s syntactic structure abstractly
 - language syntax consists of two distinct linguistic structures
 - discrete lexical structure
 - legal "words" in the language
 - context-free syntactic structure
 - legal and unambiguous "phrases" in the language

2/15/05

8

Lexical structure

- lexical structure is described using a meta-language for a regular language
 - use "regular expression" notation
 - implement a deterministic finite automata (DFA) is used to recognize lexemes
 - Lex is a popular tool and regular expression language for specifying the lexical structure for a language (e.g., C)
 - many others: ML-Lex for ML, Jlex for Java, etc.
 - lex generates a "lexical scanner" which is a program that implements a DFA to recognize tokens
 - scanner ignores white space & comments
- types of lexical symbols:
 - constants, literals, variable length identifiers, predefined keywords, operator symbols, grouping symbols, punctuation symbols
 - we generally call all of these language symbols "tokens"
 - tokens can be strings and a lexical scanner will typically match the longest possible token string

2/15/05

9

Theory of Formal Languages

- The Chomsky Hierarchy
 - a language is said to be of "type i " if it is generated by a "type i " grammar, for $i = 0,1,2,3$

Grammars	Languages	Automata
Type 0: phrase-structure context-sensitive with erasing	recursively enumerable sets	non-deterministic or deterministic Turing machines
Type 1: context-sensitive monotonic	context-sensitive	non-deterministic linear bounded Turing machines
Type 2: Context-free	context-free	non-deterministic pushdown automata
Type 2 subsets: LR(k) and LL(k)	deterministic context-free	top-down or bottom-up deterministic pushdown automata (PDA)
Type 3: regular right/left linear	regular sets	1-way or 2-way NFA or DFA

Table adapted from M.A. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, 1978

2/15/05

10



Deterministic Context-Free Languages

- LR(k) grammars
 - this class of grammars is broad enough to include the syntax of almost all programming languages
 - require a left-to-right scan of the input doing a right-most derivation in reverse
 - bottom-up shift-reduce parsers require LR(k) grammars
 - k=1 is sufficient for building efficient parsers
 - SLR(1) and LALR(1) grammars are a practical subclass of LR(K) that lead to efficient parser implementations
 - SLR = Simple LR
 - LALR = lookahead LR
 - Yacc is based on LALR(1)
 - both SLR(1) and LALR(1) parsers typically have a few hundred states in the DPDA for typical programming languages, whereas for LR(1) there are often several thousand

2/15/05

11



Deterministic Context-Free Languages

- LL(k) grammars
 - this class of grammars is suitable for many programming languages, but is a subset of LR(k)
 - requires a left-to-right scan of the input doing a left-most derivation (from the starting non-terminal symbol for the grammar) with k token lookahead
 - top-down recursive descent parsers require LL(k) grammars
 - can be hand coded with each procedure in the parser corresponding to a grammar production rule
 - need to ensure that no left-recursion occurs in the grammar to avoid infinite recursion
 - for efficient parsers, K=1 or K=2

2/15/05

12

Ambiguity

- Languages & grammars may admit ambiguous syntactic structures
 - ambiguity means that we can obtain more than one parse tree for the same expression
 - how do we cope with them?
 - change the language to be unambiguous
 - define grammar rules to eliminate the ambiguities
 - build compilers that "do the right thing" anyway
 - Example:
 - $\text{Expr} ::= \text{Expr} + \text{Expr} \mid \text{Expr} - \text{Expr} \mid \text{Expr} * \text{Expr} \mid \text{Expr} \setminus \text{Expr} \mid \text{Expr} \wedge \text{Expr} \mid (\text{Expr})$
 - $a + b * c =? (a+b)*c$ or $=? a + (b*c)$
 - We know based on semantic precedence & associativity rules that the second expression is the correct one.
 - How do we fix this: Syntactically? Semantically? Pragmatically?

2/15/05

13

Ambiguity

- In the case of expressions, we can rewrite the grammar to deal with the ambiguity
 - introduce additional non-terminal symbols to represent specific elements of the expression
 - order the production rules (top-to-bottom) such that the rules force the proper precedence evaluation order
 - order the non-terminal symbols in a rule based on associativity rules
 - $\text{Expr} ::= \text{Expr} + \text{Term} \mid \text{Expr} - \text{Term} \mid \text{Term}$
 - $\text{Term} ::= \text{Term} * \text{Factor} \mid \text{Term} / \text{Factor} \mid \text{Factor}$
 - $\text{Factor} ::= \text{Expr} \wedge \text{Factor} \mid (\text{Expr})$
 - If we want to add an assignment operator rule, how do we do it?

2/15/05

14

Syntactic Ambiguity

- **Conditional statement in C is ambiguous**
 - `if E1 then if E2 then C1 else C2`
- **Does this mean:**
 - `if E1 then {if E2 then C2 else C3 }`
 - `or`
 - `if E1 then { if E2 then C2} else C3`
- **We know by convention, that the first alternative is the semantically correct one for modern languages**
 - how do we deal with this?
 - we could change the language and add a "fi" or "endif" keyword to terminate each conditional statement
 - `if E1 then if E2 then C1 else C2 endif endif`
 - we could try to rewrite the grammar rules to deal with it
 - we could "fix" the compiler to recognize this special case and implement a semantic rule to deal with it
 - Yacc does this by always favoring a "shift" action over a "reduce" action, which turns out to be the right thing in this case, but reports a shift-reduce conflict warning in case this is not what you want to happen

2/15/05

15

Language Semantics

- **Programming language semantics**
 - **assigning meaning to each legal syntactic phrase**
 - such that the meanings are consistent under composition
 - for example, what do the following phrases in C mean? Are they equivalent?
 - `if E_1 then C_1 else C_2`
 - `$(E_1) ? C_1 : C_2$`
 - **semantic functions**
 - to define semantics, we define syntactic domains D_k for each syntactic category
 - e.g., identifiers, booleans, integers, expressions, locations, commands
 - we then define semantic functions that map elements of each syntactic domain to their semantic definition in a semantic language (e.g., an extended λ -calculus)
 - `[[$x+3$:Expr]] = ...($\lambda x.x+3$)...`
 - or sometimes written as `E[[$x+3$]] = ...($\lambda x.x+3$)...`

2/15/05

16

Language Semantics

- Why define a semantics?
 - we would like a clear and unambiguous definition of what a program written in a programming language "means"
 - we usually write a manual that is a few pages to hundreds of pages that describes (in English) what programs written in a language mean
 - Algol 60 report ~16 pages
 - LISP 1.5 Programmer's manual ~100 pages
 - Revised⁵ Scheme report ~50 pages (incl. semantic definition)
 - ANSI C K&R book ~272 pages
 - ANSI C++ standard ~740 pages
 - do you really understand a language like C/C++ completely?
 - given this legal syntax:
 - `x=1;`
 - `x = ++x + x++;`
 - what is the final value of x?

2/15/05

17

Language Semantics

- Semantic driven compilation
 - compilers/interpreters effectively implement the semantics of a language
 - one goal for formal semantics is to achieve provably correct compilers
 - have you ever had a compiler incorrectly compile your program into machine code, introducing a bug? E.g., during optimization
 - We may also want to know when two programs are equivalent in the sense that they "denote" the same mathematical function
 - in what way are the following two factorial programs equivalent?
 - `fun fact 0 = 1 | fact n = n * fact (n);`
 - `fun fact n =`
 `let fun tfact (0,m) = m`
 `| tfact (n,m) = tfact (n-1,m*n)`
 `in`
 `tfact(n,1)`
 `end;`
 - they both compute the same output on the same input
 - but limits are often placed on the size of n (e.g., word size)

2/15/05

18



Different Approaches to Semantics

- Denotational semantics
 - D. Scott & C. Strachey
- Structured Operational semantics
 - G. Plotkin
- Axiomatic semantics
 - R. Floyd and C.A.R. Hoare: Floyd-Hoare logic
 - E. Dijkstra's predicate transformers
- Action Semantics
 - Peter Mosses
- Algebraic Semantics
 - J. Goguen, M. Arbib & E. Manes
- Definitional Interpreter
 - the interpreter (eval!) embodies the semantics
- Ad hoc semantics
 - write code, run program, debug until correct

2/15/05

19



A Core Imperative Language (CIL)

- Syntax domains
 - concrete syntax is the syntax of the language user
 - abstract syntax describes the language structure
 - BNF is a *meta* syntax used to define the abstract syntax
 - Backus Naur Form not Backus Normal Form (Knuth)
 - normally partition the syntactic forms of a language into distinct syntactic domains as an organizing principle
 - E.g., numerals, expressions, locations and commands. Maybe also Type Declarations, Labels...
 - $C \in$ Command
 - $E \in$ Expression
 - $L \in$ Location
 - $N \in$ Numeral

2/15/05

20

BNF Grammar for CIL

- Grammar symbols and production rules
 - meta syntax symbols used to form production rules
 - ::=, |
 - non-terminal symbols used to represent instances of each syntax domain
 - terminal symbols representing concrete “tokens” in the language, e.g. operators & keywords
 - Abstract syntax for CIL:

```
C ::= L := E | C1;C2 | if E then C1 else C2 fi | while E do C od | skip
E ::= N | @L | E1 + E2 | ~E | E1 = E2
L ::= loci    if i > 0
N ::= n,     if n an integer
```

2/15/05

21

Expression Domain Grammar

- We have to deal with expression ambiguity by defining a concrete syntax for expression formation in CIL, like we saw before, to cope with operator precedence and associativity rules, otherwise expressions like $1+2*3$ are ambiguous
 - $E ::= E + T \mid T$
 - $T ::= T * F \mid F$
 - $F ::= N \mid @L \mid \sim E \mid (E)$

2/15/05

22



Typing Rules

- syntax rules may still admit malformed phrases
- we can use information about types to ensure that only type correct phrases are formed
 - E.g., boolean expression vs integer expressions
 - if we have typing rules, we can add a type attribute to elements of our syntax trees to indicate that the syntax tree is well-typed
 - this type annotation prohibits the formation of non-well-typed syntax trees
 - to do this, we need to augment our syntax definition with typing rules

2/15/05

23



Static Typing

- The typing rules define a static typing
 - static typing means we can determine the type of a phrase before executing the phrase
 - a language is strongly typed if no well-typed program produces run-time type errors

2/15/05

24



Unicity of Typing

- A language has the unicity of typing property if every syntax tree has at most one assignment of typing attributes to its nodes
 - if P is a syntactically legal phrase in a language L , and $P:\tau$ holds, τ is unique
 - said another way: every program is unambiguous with regard to its type rules
 - E.g., the SML type system enforces the unicity of typing rule. If it cannot determine the type of a polymorphic function explicitly, it defaults to `int`
 - `fun square x = x * x;`
 - Think of the types as constraints on the set of legal syntax trees that can be formed, further restricting the set of trees constructible under the rules of the grammar.
 - Hence "programs" consist of only well-typed syntactic phrases that can be formed
 - a lexer/parser combination admits only well-formed syntax
 - the type checker admits only well-typed programs

2/15/05

25




Natural Deduction

- Originally due to Gerhard Gentzen
- Introduction rules & elimination rules
 - e.g., can define the rules for deduction in propositional logic in this manner
 - for programming languages, we adopt this type of rule-based logical deduction to define the rule for meaningful phrases in a well-typed language

$$\begin{array}{c}
 A \quad B \\
 \text{-----} \\
 A \ \& \ B
 \end{array}
 \quad \& \ I
 \qquad
 \begin{array}{c}
 A \ \& \ B \\
 \text{-----} \\
 A
 \end{array}
 \quad \& \ E
 \quad \text{or} \quad
 \begin{array}{c}
 A \ \& \ B \\
 \text{-----} \\
 B
 \end{array}
 \quad \& \ E$$

2/15/05

26




Natural Deduction Type Rules

Command rules:

$\frac{L:\text{intloc} \quad E:\text{intexp}}{L := E : \text{comm}}$	$\frac{C1:\text{comm} \quad C2:\text{comm}}{C1;C2 : \text{comm}}$	$\text{skip} : \text{comm}$
$\frac{E:\text{boolexp} \quad C1:\text{comm} \quad C2:\text{comm}}{\text{if } E \text{ then } C1 \text{ else } C2 \text{ fi} : \text{comm}}$	$\frac{E:\text{boolexp} \quad C:\text{comm}}{\text{while } E \text{ do } C \text{ od} : \text{comm}}$	

2/15/05
27



Natural Deductions Type Rules

Expression rules:

$\frac{N:\text{int}}{N:\text{intexp}}$	$\frac{L:\text{intloc}}{@L:\text{intexp}}$	$\frac{E1:\text{intexp} \quad E2:\text{intexp}}{E1 + E2 : \text{intexp}}$
$\frac{E:\text{boolexp}}{\sim E : \text{boolexp}}$	$\frac{E1:t\text{-exp} \quad E2:t\text{-exp}}{E1 = E2 : \text{boolexp}} \quad \text{if } t = \{\text{int}, \text{bool}\}$	

<p>Location rules:</p> $loc_i : \text{intloc}, \text{ if } i > 0$	<p>Numeral rules:</p> $n : \text{int}, \text{ if } n \text{ is in the set Integer}$
---	---

2/15/05
28

Semantics of the Core Language

- The purpose of a well-designed syntax is to guide the programmer's understanding of the semantics, leading to correct programs
 - but we need more than just intuitive understanding
- Core imperative programming language
 - locations and numerals represent themselves
 - expressions represent integers and booleans
 - commands represent state transformations
- Denotational semantics is one way to formalize the semantics of a language
 - what is the meaning of each legal syntactic phrase?
 - a denotational semantics maps well-typed derivation trees to their mathematical meanings
 - $[[E1 := E2:comm]] = ???$

2/15/05

29

Semantic domains & operations

```
Bool = { false, true }
not: Bool -> Bool
not(false) = true
not(true) = false
equalbool: Bool * Bool -> Bool
equalbool (m,n) = (m = n)
```

```
Int = {-∞, ..., -1, 0, 1, ..., ∞}
plus: Int * Int -> Int
plus(m,n) = m + n
equalint: Int * Int -> Bool
equalint (m,n) = (m=n)
```

```
Location { loci | i > 0 }
```

```
Store = { <n1, n2, ..., nm> | ni in Int, 1 <= i <= m, m >= 0 }
lookup: Location * Store -> Int
update: Location * Int * Store -> Store
if: Bool * Store * Store -> Store
if(true, s1, s2) = s1
if(false, s1, s2) = s2
```

2/15/05

30



Semantics of the Core Language

- A denotational semantics is a recursive definition that maps well-typed derivation trees to their mathematical (functional) meanings
 - Bool is a semantic domain with two values
 - true and false
 - two semantic functions (in a sugared λ -calculus)
 - not - logical negation
 - equalbool - boolean equality
 - Int is the set of integers
 - addition and equality functions
 - $[[\underline{2}:\text{int}]] = 2$, where $\underline{2}$ is a numeral and 2 is an integer
 - Location is a set of labeled locations
 - $[[\text{loc}_i:\text{intloc}]] = \text{loc}_i$

2/15/05

31



The Store

- An abstraction over locations and values
 - semantics for L-values and R-values
 - store is a linear storage vector
 - lookup function
 - $\text{lookup}(\text{loc}_j, \langle n_1, n_2, \dots, n_j, \dots, n_m \rangle) = n_j$
 - if $j > m$, $\text{lookup}(\text{loc}_j, \langle n_1, \dots, n_m \rangle) = 0$
 - update function
 - $\text{update}(\text{loc}_j, n, \langle n_1, n_2, \dots, n_j, \dots, n_m \rangle) = \langle n_1, n_2, \dots, n_j, \dots, n_m \rangle$
 - if $j > m$, $\text{update}(\text{loc}_j, \langle n_1, \dots, n_m \rangle) = \langle n_1, \dots, n_m \rangle$
- The if operation selects a store based on a boolean expression evaluation

2/15/05

32



Compositional Semantics

- The meaning of a well-typed program is a recursive equation using $[[\cdot]]$
 - for $L := E$ we compute the meaning of this command using a semantic equation along with a typing rule
 - the typing rule indicates the compositional elements of the rhs of the equation

$$\frac{L:\text{intloc} \quad E:\text{intexp}}{L := E : \text{comm}}$$

$$[[L := E : \text{comm}]] = \dots [[L:\text{intloc}]] \dots [[E:\text{intexp}]]$$

2/15/05

33



Interpreter Semantics

- Can think of an interpreter having a universal eval function
 - any valid syntactic phrase is the input
 - the output is a string representing the reduction of that phrase to a value specified by the semantics
 - so the internals of an interpreter would have a number of functions/procedures that implement the semantic functions that correspond to each of the syntactic forms of the language
 - Also a lot of internal bookkeeping data structures and code to enforce proper typing, lexical scoping, recursion, etc.

2/15/05

34

Semantic equations

Command:

```

[[L:=E:comm]](s) = update([[L::intloc]], [[E:intexp]](s),s)
[[C1;C2:comm]](s) = [[C2:comm]] ([[C1:comm]](s))
[[if E then C1 else C2 fi:comm]](s) = if ([[E:boolexp]](s)), [[C1:comm]](s), [[C2:comm]](s)
[[while E do C od:comm]](s) = w(s)
    where w(s) = if ([[E:boolexp]](s)), w[[C:comm]](s), s)
[[skip:comm]](s) = s
    
```

Expression:

```

[[N:intexp]](s) = [[N:int]]
[[@L:intexp]](s) = lookup([[L:intloc]],s)
[[E1+E2:intexp]](s) = plus([[E1:intexp]](s),[[E2:intexp]](s))
[[~E1:boolexp]](s) = not ([[E1:boolexp]](s))
[[E1=E2:boolexp]](s) = equalbool([[E1:boolexp]](s), [[E2:boolexp]](s))
[[E1=E2:boolexp]](s) = equalint([[E1:intexp]](s), [[E2:intexp]](s))
    
```

Location: $[[loc_i:intloc]] = loc_i$

Numeral: $[[n:int]] = n$

2/15/05

35

Semantics of Expressions

- Semantics of an expression is dependent on the store
 - given the initial storage vector $\langle 3,4,5 \rangle$
 - example 1:
 - $[[@loc_1:intexp]]\langle 3,4,5 \rangle = \text{lookup}([[loc_1:intloc]],\langle 3,4,5 \rangle) = 3$
 - example 2:
 - $[[@loc_1+1:intexp]]\langle 3,4,5 \rangle = \text{plus}([[@loc_1:intloc]]\langle 3,4,5 \rangle, [[1:intexp]]\langle 3,4,5 \rangle)$
 $= \text{plus}(3, [[1:int]])$
 $= \text{plus}(3,1) = 4$

2/15/05

36



The While loop

- iteration and recursion are semantically difficult because they require explicit control
 - what is required is a way to represent a self-referential object in the semantic domain
 - one way to do this is to consider a special semantic function that incrementally approximates or converges to the final value of the iteration/recursion
 - Practically, you use side-effects to manage state (e.g., a stack) and control mechanisms for branching (e.g., goto)

2/15/05

39



Homework

- Read Chapter 3 - Environment-passing Interpreters
 - See the book's website to download code examples
 - <http://www.cs.indiana.edu/eopl/>
 - Extend the interpreter according to these exercises:
 - 3.7, 3.8, 3.10, 3.11, 3.12, 3.13, 3.15
 - I recommend doing 3.15 before 3.10 to add boolean expressions rather than using 0/1 to represent false/true

2/15/05

40