

**The Structure of Typed Programming Languages**  
**(Introduction & Chapter 1)**  
 CS386L- Programming Languages  
 Dr. Greg Lavender  
 Department of Computer Sciences  
 The University of Texas at Austin

CS386L: Lecture-05 - The Structure of Typed Programming Languages: Chapter 1

## Theory of Formal Languages

- 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

3/1/04
2

## 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
    - see handout of lex specification & Yacc grammar for  $C$
  - language syntax consists of different two main linguistic structures
    - discrete lexical structure
      - legal "words" in the language
    - context-free syntactic structure
      - legal and unambiguous "phrases" in the language

3/1/04

3

## 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

3/1/04

4

## 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

3/1/04

5

## Deterministic Context-Free Languages

- LL(k) grammars
  - this class of grammars is suitable for many programming languages, but is a subset of LR(k)
  - require 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 in the grammar to avoid infinite recursion
    - for efficient parsers, K=1 or K=2

3/1/04

6

## Ambiguity

- Languages & grammars may admit ambiguous 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?

3/1/04

7

## 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?

3/1/04

8

## 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

3/1/04

9

## 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  $\lambda$ -calculus)
      - `[[x+3:Expr]] = ...( $\lambda x.x+3$ )...`
      - or sometimes written as `E[[x+3]] = ...( $\lambda x.x+3$ )...`

3/1/04

10

## 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
      - Revised<sup>5</sup> 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  $x$ ?

3/1/04

11

## 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?
      - $\text{fun fact } 0 = 1 \mid \text{fact } n = n * \text{fact } (n);$
      - $\text{fun fact } n =$ 

```

              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)

3/1/04

12

## 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-Hoard logic
  - E. Dijkstra's predicate transformers
- Action Semantics
  - Peter Mosses
- Algebraic Semantics
  - J. Goguen, M. Arbib & E. Manes
- Ad hoc semantics
  - everyone else!

## 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
    - $C \in \text{Command}$
    - $E \in \text{Expression}$
    - $L \in \text{Location}$
    - $N \in \text{Numeral}$

## 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
  
```

## 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
  - $E ::= E + T \mid T$
  - $T ::= T * F \mid F$
  - $F ::= N \mid @L \mid \sim E \mid (E)$

## 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

## 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



## Natural Deduction Type Rules

Command rules:

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

## Natural Deductions Type Rules

Expression rules:

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

Location rules:

$$loc_i : \text{intloc}, \text{ if } i > 0$$

Numeral rules:

$$n : \text{int}, \text{ if } n \text{ is in the set Integer}$$