

Programming Language Basics

Organization of Programming Languages

- What features are needed in a PL?
- How do we evaluate a PL?
- How did PLs evolve?
- What design/implementation decisions go into writing a PL?
- Why/how do PLs differ?

Why study PLs?

- increase our capacity to express ideas (both programming and in general)
- increase ability to select and learn a PL
- better understanding of implementation issues in programming
- increase ability to design new PLs
- better understanding of computational theory

Application areas for PLs

- Historically, we see that PLs have had intended uses:
 - Scientific - floating point, array operations
 - Business - elaborate I/O, files, sorting
 - AI - symbolic proc/strings, recursion, lists
 - Systems - low-level (bit) facilities
 - Scripting - such as sh, awk, perl, used to tailor the OS environment to a user's specifications
 - Special purpose - statistics, reports, robotics, 4GL, graphics

Language Evaluation Criteria

- Readability
- Writability
- Reliability
- Cost
- Tradeoffs

Readability

- Simplicity of instructions
- Orthogonality of instructions
- Control statements
- Data Types and Structures
- Syntactic Issues -- identifier rules, reserved or special words, form and meaning

Writability

- Flexibility and availability of control structures, data types
- Simplicity and Orthogonality
- Support for Abstraction
- Expressivity

Reliability

- Readability and Writability
- Type Checking
- Exception Handling
- Aliasing

Cost

- Training/Learning Curve
- Time to write programs
- Compilation Speed/Efficiency
- Execution Speed/Efficiency
- Maintenance

Tradeoffs

- Readability vs. Flexibility
- Cost vs. Expressivity
- Type Checking vs. Abstraction and Flexibility

Design Influences

- Computer Architecture - usually will concentrate on Von Neumann architectures
 - Other architectures such as parallel processing (MIMD or SIMD) or pipelining might require very different languages
- Program Methodologies
 - Procedural vs. Functional vs. Logical vs. Object-oriented

Implementation Methods

1. **Compilation**
 - Translate high-level program to machine code
 - Slow translation
 - Fast execution
2. **Pure interpretation**
 - No translation
 - Slow execution
 - Becoming rare
3. **Hybrid implementation systems**
 - Small translation cost
 - Medium execution speed

Basic Computer Machinery

- The computer is an interpreter:
 - Initialize Program Counter (PC)
 - Repeat
 - Fetch instruction at PC
 - Increment PC
 - Decode Instruction into Microcode
 - Fetch Operand(s) for instruction at PC
 - Increment PC
 - Until no more programs

Layered Interface

- At the center is the bare machine (hardware)
- Macro instruction interpreter (fetch-execute cycle)
- Operating System
- Compilers, Interpreters, Assemblers, Libraries
- Software

Syntax and Semantics

- Syntax - form or structure of expressions or statements for a given language
- Semantics - meaning of the expressions
- Language - group of words that can be combined and the rules for combining those words
- Lexeme - lowest level syntactic unit in the language

Grammars

- Language Recognizer - given a sentence, is it in the given language?
- Language Generator - given a language, create legal and meaningful sentences
- We can build a language recognizer if we already have a language generator
- Grammar - a description of a language - can be used for generation

Classification of languages

- Regular
- Context-Free
- Context-Sensitive
- Recursively Enumerable
- Context-Free grammars include those which can be generated from a language generator (grammar)
- These include natural languages and programming languages

BNF (Backus Normal Form)

- Equivalent to a context-free language
- Used to specify the grammar of a language and can then be used for language generation or recognition
- Contains terminals, non-terminals and rules which map non-terminals into expressions of other non-terminals and terminals

BNF Grammar

- Mathematically, a BNF Grammar is given as $G = \{\text{alphabet, rules, } \langle \text{start} \rangle\}$
- alphabet -- those symbols used in the rules (both terminals and non-terminals)
- rules -- map from a non-terminal to other elements in the alphabet
- $\langle \text{start} \rangle$ -- a non-terminal which must be on at least 1 rule's left hand side

Examples of Grammar Rules

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

Notice the use of both terminals and non-terminals on the right side

Recursion is used as necessary

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

An Example Grammar

```
<program> -> <stmts>
<stmts> -> <stmt> |
          <stmt> ; <stmts>
<stmt> -> <var> = <expr>
<var> -> a | b | c | d
<expr> -> <term> + <term>
          | <term> - <term>
<term> -> <var> | const
```

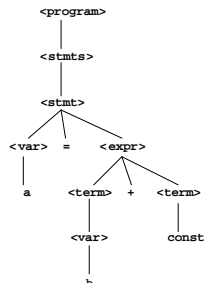
A derivation from the grammar

```
<program> => <stmts> => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

Parse Trees

- A hierarchical structure displaying the derivation of an expression in some grammar
- Leaf nodes are terminals, non-leaf nodes are non-terminals
- Parser - takes a sentence and breaks it into its component parts, deriving a parse tree. If the parser cannot generate a parse tree, then the sentence is not legal

A grammar is *ambiguous* iff it generates a sentential form that has two or more distinct parse trees



An Ambiguous Grammar

- <assign> --> <id> := <expr>
- <id> --> A | B | C
- <expr> --> <expr> + <expr>
 - | <expr> * <expr>
 - | (<expr>)
 - | <id>
- Can generate A := B + C * A and A := C * A + B -- both of which should mean the same but do not!

An Unambiguous Grammar

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

Ambiguous If-Then-Else

- `<If> --> IF <logical expr> THEN <stmt>`
| `IF <logical expr> THEN <stmt>`
| `ELSE <stmt>`
- We could generate the stmt:
 - If $X > 0$ Then If $Y > 0$ Then $X:=0$ Else $X:=Y$
 - Which condition does the else clause get attached to? $X > 0$ or $Y > 0$?
- We could use begin-end pairs or other syntactic structures to remove the ambiguity

Unambiguous If-Then-Else

- `<stmt> --> <matched> | <unmatched>`
- `<matched> --> IF <logical expr> THEN`
`<matched> ELSE <matched>` |
any non-if statement
- `<unmatched> --> IF <logical expr>`
`THEN <stmt> |`
`IF <logical expr> THEN`
`<matched> ELSE <unmatched>`

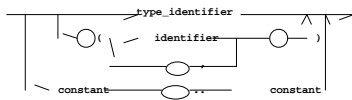
Extended BNF Grammars

- 3 common extensions to BNF:
 - [] - used to denote optional elements (saves some space so that we don't have to enumerate options as separate possibilities)
 - { } - used to indicate 0 or more instances
 - () - for a list of choices
- These extensions are added to a BNF Grammar for convenience allowing us to shorten the grammar

Syntax Graphs

- Pictorial way of illustrating a syntax

Syntax Graphs - put the terminals in circles or ellipses and put the nonterminals in rectangles; connect with lines with arrowheads



Recursive Descent Parsing

- A top-down parser, as used in program compilation or natural language understanding
- Constructs a “parse tree” from a given input and a grammar
- `<expr> --> <term> { (+|-)<term> }`
- ```
void expr () {
 /* parse the first term */ term ();
 while
 (next_token == plus_code ||
 next_token == minus_code) {
 lexical (); /* get next token from the input */
 term (); /* parse next term */
 }
```

### Attribute Grammars

- It is not possible to describe all aspects of a language solely with a BNF Grammar such as semantic information
  - In Pascal, there is a limit to the size of an identifier name
  - In an assignment statement, the left hand side type must match the value computed by the right hand side
- Attribute grammars are added to BNF grammars to handle these gaps

### Attribute Grammars

- Attributes are variables that take on values
- Rules are added to the grammar to ensure correctness by testing values of variables
  - $\langle ID \rangle \rightarrow \_ \langle ID \rangle | \langle letter \rangle \langle ID \rangle | \langle number \rangle \langle ID \rangle$
  - $\langle ID \rangle.length \leftarrow \langle ID \rangle.length + 1$
  - Correct  $\leftarrow \langle ID \rangle.length \leq 31$

### Semantics

- Describing the meaning of a program or of a statement or group of statements
- Operational Semantics - how the stmt will be executed
- Axiomatic Semantics - what results to expect from the stmt
- Denotational Semantics - a functional method of mapping the affects of a stmt

### Operational Semantics

- This can be thought of as “tracing” through a program to see what affects an instruction will have
- Implemented as an interpreter or compiler or assembler -- that is, how will the computer execute this instruction?
- This is simply a mechanistic description of the stmt and does not necessarily help us understand the stmt

### Ex. of Operational Semantics

- For (expr1; expr2; expr3) {...} becomes:
  - expr1;
  - loop: IF expr2 = 0 GOTO out
  - ...
  - Expr3;
  - GOTO loop
- out: ...

### Axiomatic Semantics

- Used mainly to prove correctness of code
- Each stmt in the language has associated pre and post conditions that specify what happens in the program when the stmt executes (what variable values change?)
- Basic form of an axiomatic semantic is  $\{P\} S \{Q\}$   
if P is true before S, then Q is true after S