CPS506 - Comparative Programming Languages

Comparison

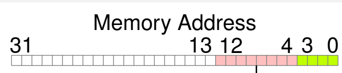
Dr. Dave Mason
Department of Computer Science
Ryerson University

Many paradigms over time

- Imperative - Fortran, C, Rust
- Functional - Lisp, Scheme, Clojure, Elixir, Haskell
- Object-Oriented - Simula, Smalltalk, C++, Java, Ruby
- Concurrent - Erlang, Elixir, Concurrent Euclid
- Parallel and Array - APL, MATLAB, R, SISAL
- Declarative - yacc, make
- Constraint - Prolog
- Dataflow - LabVIEW, PureData, Kit, Prograph, Max/MSP, spreadsheets

Evolution of Programming Languages

- Machine Language
- Assembly Language
- Low-Level Languages
- Programming Paradigms
  - Imperative
  - Functional
  - Object-Oriented
  - Concurrent
  - Parallel and Array
  - Declarative
  - Constraint
  - Dataflow
- Efficiency
  - Assembler
  - Native Code Compilers (Ahead-Of-Time)
  - Source Interpreters
  - Byte-Code Interpreters
  - Just-In-Time Compilers
- Architecture/Language/Compiler entanglement
  - Parallelism



Programming Language Basics

- Static/Dynamic Distinction
  - Declarations
  - Types
  - Bounds
  - Values
- Names, Identifiers, Variable
  - Identifiers are indentifying strings of characters
  - Variables are locations that contain values
    - usually mutation is implied
  - Aliasing - a variable can have multiple names
- Procedures, Functions, Methods
  - Functions act by returning a value
    - Pure functions have no side effects
  - Procedures act by side-effect
  - Methods are procedures/functions associated with an object (possibly via a class)

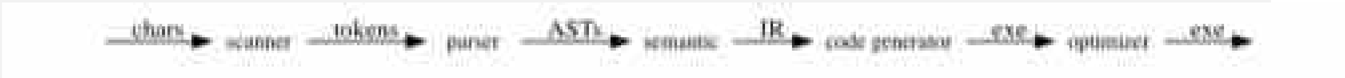
Programming Language Basics ...

- Declarations, Definitions
  - Declarations designate space/type
  - Definitions give values/implementations
- Parameter Passing Mechanisms
  - Call-by-Value
  - Call-by-Reference
  - Call-by-Name
  - Call-by-Value-Return
  - Call-by-Pattern

Syntax

- Simplicity - how much to learn
  - size of the grammar
  - complexity of navigating modules/classes
  - complexity of the type system
- Orthogonality - how hard to learn, how do features interact
  - number of special syntax forms
  - number of special datatypes
  - type system
- Extensibility - how can language align with problem
  - functional
  - syntactically
  - defining literals
  - overloading

Recognizing language components



- Scanner
  - convert characters to tokens
  - ignore comments/whitespace (unless relevant)
  - highest throughput
  - usually Regular-Expressions
  - implemented as Finite-State-Automata (FSA)
- Parser
  - order of tokens
  - typically convert to Abstract-Syntax-Tree (AST)
  - usually Context-Free-Grammar
  - many classes of CFGs
  - implemented as Pushdown-Automata
  - recursive-descent or table-driven
- Semantic Analysis
  - type checking
  - implemented as Context-Sensitive-Grammar

E.g. straight-line programming language

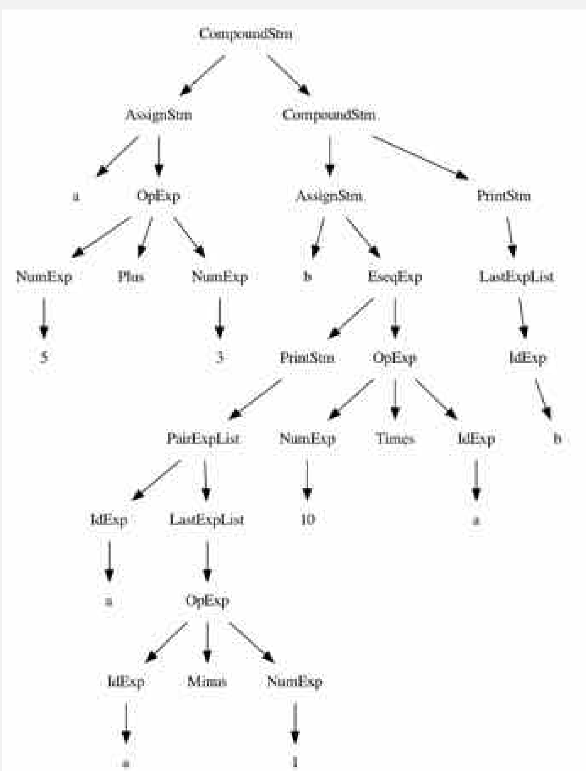
```
a := 5 + 3; b := (print ( a , a - 1 ) , 10 * a ); print(b)
```

semicolon	:	`;`
assign	:	`:=`
leftParen	:	`(`
rightParen	:	`)`
plus	:	`+`
minus	:	`-`
times	:	`*`
divide	:	`/`
comma	:	`,`
id	:	`[a-zA-Z][a-zA-Z]*`
print	:	`print`
num	:	`[0-9][0-9]*`

Grammar for straight-line programming language

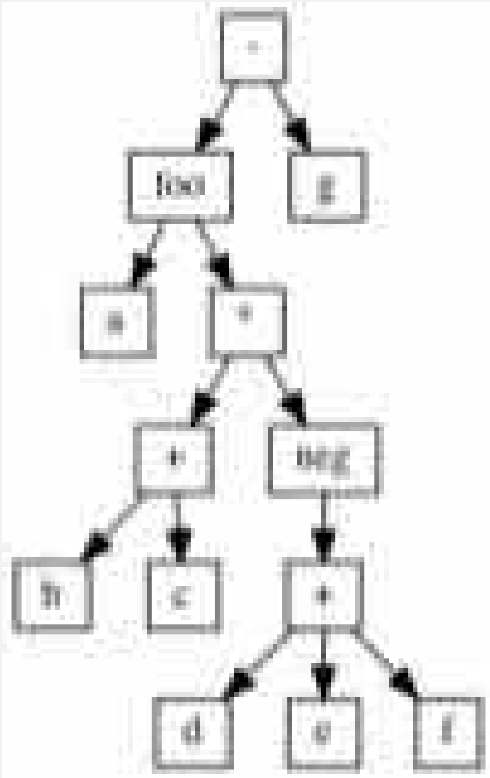
Stm	→	*Stm* ; *Stm*	(CompoundStm)
Stm	→	`id` := *Exp*	(AssignStm)
Stm	→	`print` (*ExpList*)	(PrintStm)
Exp	→	`id`	(IdExp)
Exp	→	`num`	(NumExp)
Exp	→	*Exp Binop Exp*	(OpExp)
Exp	→	(*Stm* , *Exp*)	(EseqExp)
ExpList	→	*Exp* , *ExpList*	(PairExpList)
ExpList	→	*Exp*	(LastExpList)
Binop	→	+	(Plus)
Binop	→	−	(Minus)
Binop	→	*	(Times)
Binop	→	/	(Div)

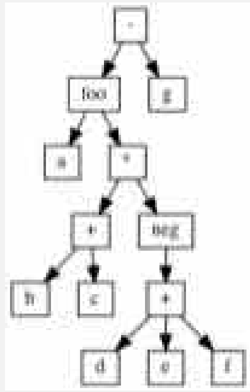
Tree representation of straight-line program



`a := 5 + 3 ; b := ( print ( a , a - 1 ) , 10 * a ) ; print ( b )`

Expression Syntax





Prefix

```
(- (foo a (* (+ b c) (- (+ d e f)))) g)
```

- Lisp (Scheme, Clojure)
- pre-order traversal of AST

Statement Syntax

- Special forms
  - Postscript
  - Smalltalk
  - Scheme
  - everything else

    - polyadic user functions
    - left-to-right, except monadic, dyadic, named

Semantics

- what does code **mean**
- addition to syntax
- more powerful syntactic models can include

Typing

- Untyped
  - similar to machine code
  - operations act on bits regardless of outcome
  - no checking of any type
- Dynamic Typing
  - Safe
  - operations know legal data
  - raise run-time errors
- Static Typing
  - compile-time determination of legality
  - weak to strong
  - OO cannot be maximally strong