CSC519
Programming Languages

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

How big is your toolbox?

If all you have is a

hammer,

every problem looks like a

nail!

PL – An Informal Example

What is a correct PL program?

Given:

- (Positive integer) variables `X1, X2, ...` (input variables),`Y1, Y2, ...`, (working variables), `Z1, Z2, ...` (output variables), and labels `l1,l2,...`
- Statements:
  - Set a variable (e.g. `X1`) to 0: `X1<=0;`
  - Increase a variable by 1: `Y1++;`
  - Loop `Y3` times: `loop Y3; <program> endloop;`
  - Statements may be labeled:
    `l: <statement>`
    Each label may only be used once.
  - General control flow: `goto <label>;` – it is illegal to jump from outside a loop into it

A PL programm is a sequence of statements, white space is ignored

For illustration, we use C-Style comments (but they are not strictly part of the language)

PL – Sematics

What does a given PL program compute?

Individual statements have the obvious meaning

Program sematics

- A program starts running at the first statement
- Unless otherwise specified, control passes from one statement to the next one
- The program terminates if control would pass to a non-existant statement
- All variables except for the `Xi` are initialized to 0
- The `Xi` are initialized to represent the input to the program
- If a program terminates, the values of the `Zi` describe the result

Examples

Compute `f(X1) = X1+1`:

```
Z1<=0;      /* Strictly redundant */
loop X1;
   Z1++;
endloop; /* Now Z1 = X1 */
Z1++;
```

Compute `f(X1,X2) = 2*(X1+X2)`:

```
Y1<=0;
loop X1;Y1++;endloop; /* Y1 = X1 */
loop X2;Y1++;endloop; /* Y1 = X1+X2 */
Z1<=0;
loop Y1;Z1++;Z1++;endloop; /* Z1 = 2*Y1 */
```

Why?

Fact: PL is as powerful as any other language (given some simple input and output recoding)

Fact: PL is quite easy to implement

Why are we using any other language?

Why?

Fact: PL is as powerful as any other language (given some simple input and output recoding)

Fact: PL is quite easy to implement

Why are we using any other language?

Answer: PL is not a very good language (except for teaching some concepts)!

(Some) Desirable Language Properties

Easy to read and write

- High-level constructs adequate for typical tasks
- Intuitive (or simple) syntax
- (Short programs)
- Supports structuring and abstraction

Efficiently executable

- Executable programs are small and fast (enough)
- Has a good mapping to the hardware

Portable

- Can be implemented easily on different hardware and under different operating systems

A Zoo of Programming Languages

AWK Ada Algol-60 Algol-68 Assembler

BASIC C C++ C# Caml CLP Clean

COBOL Common LISP Delphi Eiffel Escher

Forth Fortran Haskell Java JavaScript

Mercury ML Modula-2 Oberon OCaml PHP

Pascal Perl Prolog PL1 Python

PostScript Ruby Scheme Smalltalk

Our Aim

Learn that there are different tools

Learn that there are different categories of tools

Learn how to recognize the right tools

Learn to think about programming problems from different angles, and using different paradigms

Learn important programming language primitives

Learn some toolsmithing and language implementation

Some Criteria for Language Classification

Programming paradigm

Language organization (structure)

Application area

Language implementation

Progamming Language Paradigms

Imperative/Procedural: Statements change the global state of the computation

Object-Oriented: Objects send and receive messages that change their internal state

Functional: Function evaluation, no explicit "state"

Declarative: Describe the problem, not the computation

Mixed/Multi-Paradigm: Any or all of the above

In practice, few languages are "pure"

- LISP: Functional, with explicit state
- C++: Object-Oriented, with procedural sublanguage
- Prolog: Declarative, but allows cheating

Structured vs. Unstructured Languages

Unstructured

- Arbitrary control flow ("Spaghetti code")
- Primarily uses `goto` or similar constructs
- Examples: BASIC, COBOL, Fortran, Assember languages

Structured

- Block structure with information hiding
- Parameterized subroutines with well-defined entry and exit points
- Uses specialized loop constructs (`for`, `while`) instead of (or in preference to) `goto`
- Examples: Algol family, Pascal, Modula-2, C (but C allows arbitrary jumps)

Notes:

- Most modern languages are structured
- Most functional languages are structured
- Declarative languages (ideally) do not have explicit control flow

Application Area

General purpose languages

Special purpose languages

- "AI"-Languanges
- Real-Time Languages
- Page description languages
- ...

Scripting Languages

...

Language Implementation

Hardware-specific (''direct mapping")

– Assembler languages

Interpreted

– E.g. BASIC, LISP/Scheme (most dialects), many scripting languages

Compiled	Precompiled	JIT
Native code	Traditional languages: C, C++, Pascal (most implementations), Fortran	Java Bytecode (some I.)
Byte compiled(abstract machine code)	Java (some I.),UCSD Pascal, Prolog, ELISP	Java (some I.)
"Intermediate" Formats	BASIC (some)	Perl

Language description

Syntax

- What is a legal program?
- Normally decribed formally (with additional, informal constraints), often taught by example

Semantics

- What does it mean?
- Nearly always described informally (sometimes incomplete attempts at formal descriptions)

Environment (often implementation-dependent)

- How do programs interact with the rest of the world?
- What external functions/libraries are available?

CSC519
Programming Languages

Language Description – Syntax

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

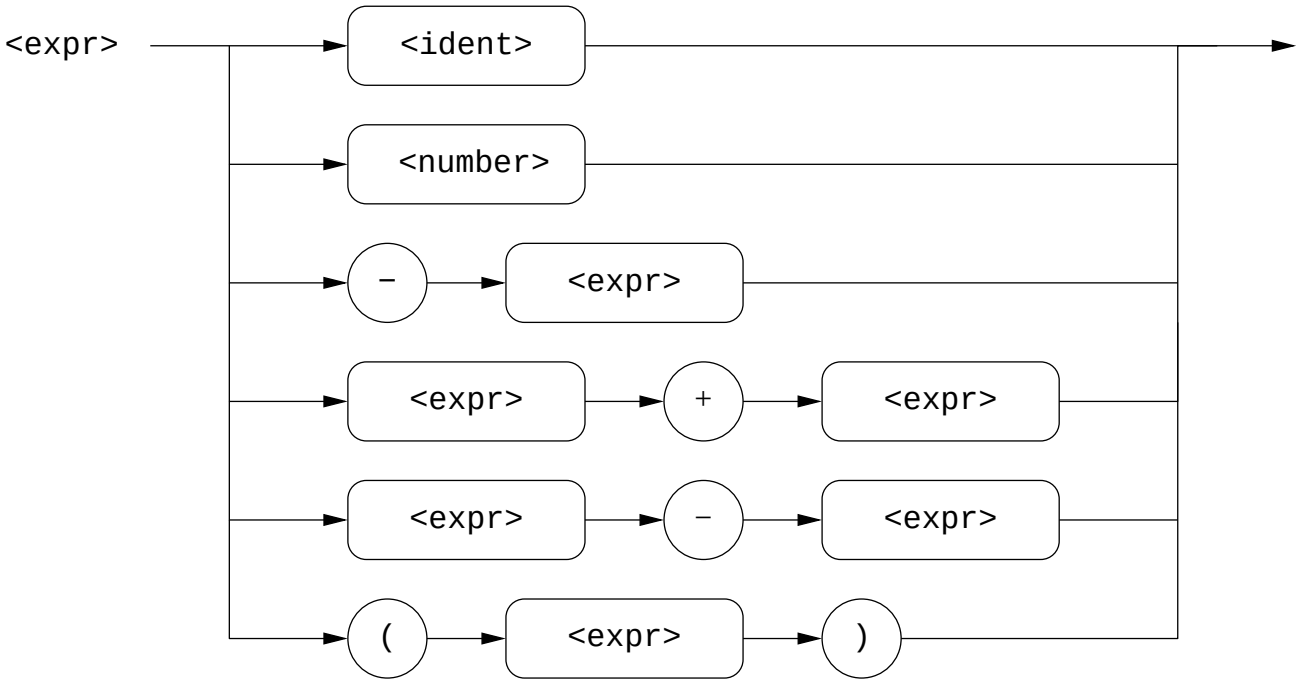
Syntax

Most obvious thing about a language

Often described in (some variant of) extended Bachus-Naur form (EBNF):

```
<expr> ::= <ident> | <number> | '-' <expr> |
           <expr> '+' <expr> | <expr> '-' <expr> | '(' expr ')';;
```

Also seen: Syntax diagrams



(Arithmetic) Expressions

We start by considering arithmetic expressions:

Mathematical notation: $\frac{1}{2}a(b+c)\sqrt{a^2+b}$

C equivalent: `1.0/2.0*a*(b+c)*sqrt(a*a+b)`

We have:

- Numbers: 1, 2 (or 1.0, 2.0)
- Variables: a, b
- Binary operators: /,*,+, (exponentiation)
- Unary operator(s): sqrt (or $\surd$)
- Parentheses for grouping

"No hassles" form: 1/2*a*(b+c)*sqrt(a*a+b) (abstracts from language idiosyncrasies)

- Numbers, variables and operators are building blocks, and are combined in certain well-defined ways: If an n-ary operator is applied to n subexpressions, the result is a new expression.

Writing Arithmetic Expressions 1

Equivalent expressions can be written in different notations:

Prefix notation: The operator precedes the arguments

- Strict version: * * * / 1 2 a + b c sqrt + * a a b
- LISP version: (* (* (* (/ 1 2) a) (+ b c)) (sqrt (+ (* a b) b)))
- Often also: *(*(*(/(1,2),a) +(b,c)),sqrt(+(*(a,b),b)))
- Advantage: Easy to parse, non-ambiguous
- LISP-Version even works with operatos with variable arity: (* (/ 1 2) a (+ b c) (sqrt(+ (* a a) b)))

Postfix notation: The operator follows the arguments

- Postfix: 1 2 / a * b c + * a a * b + sqrt *
- Advantages: Non-ambiguous, very easy to evaluate using a stack: Traverse expression from left to right.
  - If value encountered, push it onto the stack
  - If n-ary operator, pop the right number of arguments, apply the operator, and push the result.

  At the end of the expression, the result is the only value left on the stack

Writing Arithmetic Expressions 2

Infix notation: Binary operators are written between the arguments:

- Infix: 1/2*a*(b+c)*sqrt(a*a+b)
- Advantage: Familiar
- Disadvantage: Ambiguous! Without additional conventions, we have to write (((1/2)*a)*(b+c))*sqrt((a*a)+b)!
- Does not work for non-binary operators

Mathematicians use a mix of all notations

Many programming languages use a mix of all notations

Writing Arithmetic Expressions 2

Infix notation: Binary operators are written between the arguments:

- Infix: 1/2*a*(b+c)*sqrt(a*a+b)
- Advantage: Familiar
- Disadvantage: Ambiguous! Without additional conventions, we have to write (((1/2)*a)*(b+c))*sqrt((a*a)+b)!
- Does not work for non-binary operators

Mathematicians use a mix of all notations

Many programming languages use a mix of all notations

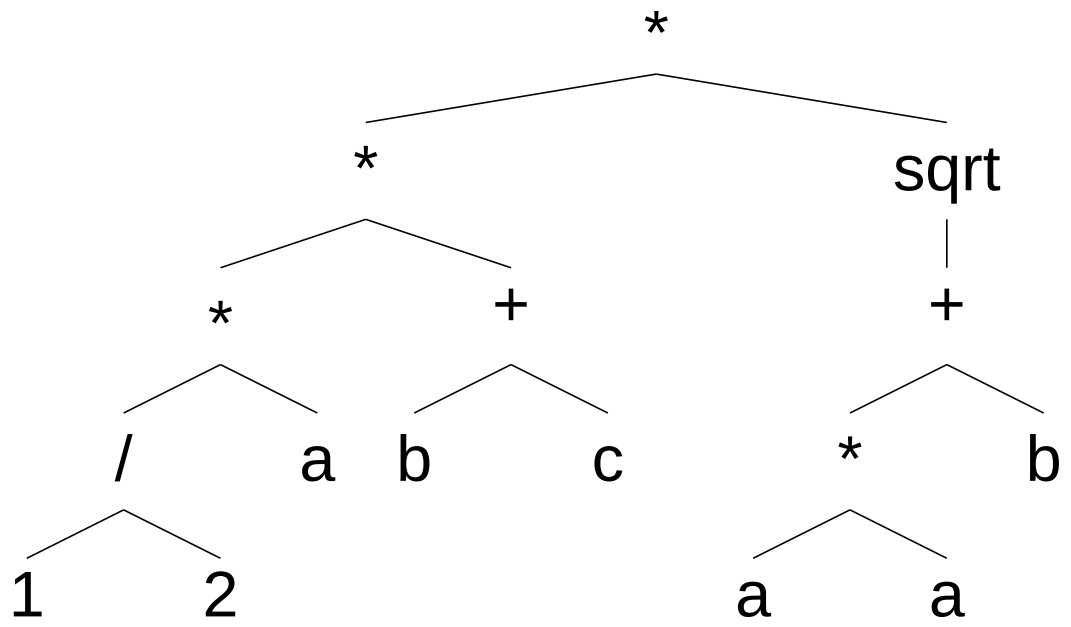
Is there a canonical form?

Abstract Syntax

Definition: An abstract syntax tree for an expression is a tree with:

- The internal nodes are labeled by operators
- The leaves are labeled by variables or numbers
- Each operator node has trees corresponding to its arguments as children.

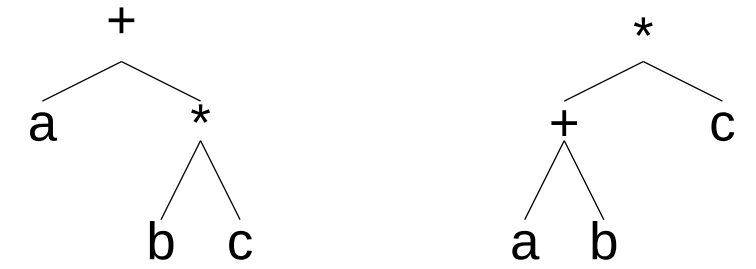
Example:



Factoid: Abstract syntax notation can be mapped 1-1 onto LISP representation (gcc even uses a LISP-like notation for its RTL language)

Operator Precedence

Problem: Two trees for a+b*c:



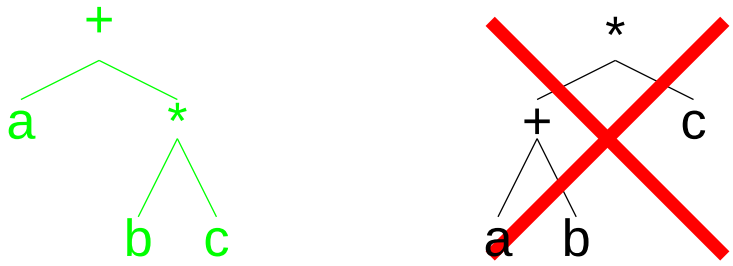
Alternative 1

Alternative 2

Which one is correct?

Operator Precedence

Problem: Two trees for a+b*c:



Alternative 1

Alternative 2

Multiplication and division have a higher precedence then addition and subtraction and thus bind tighter. a+b*c is equivalent to a+(b*c).

Different languages assign different precedences to operators!

- Common: Multiplicative operators have higher precedence than additive operators
- Assignment (:= in Pascal, = in C) has the lowest priority

Operator Associativity

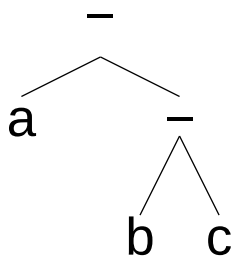
Problem: Two trees for a+b+c:

Alternative 1

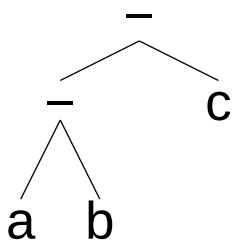
Alternative 2

Operator Associativity

Problem: Two trees for a-b-c:



Alternative 1



Alternative 2

Operator Associativity

Problem: Two trees for a-b-c:

Alternative 1

Alternative 2

If we do not want to use parentheses around every subexpression, we must decide the associativity for each operator!

- The normal arithmetic operators are all left-associative, i.e. they bind to the left. a+b-c+d is equivalent to (((a+b)-c)+d)
- In C, assignments have a value, i.e. we can write `a = b = 1;`. The assignment operator in C is right-associative (i.e. the above is equivalent to `a = (b = 1);`

Formal Languages

Definition: Let Σ be an finite, non-empty alphabet.

- A word over Σ is a sequence of elements of Σ
- The empty word is written λ
- Σ^* is the set of all words over Σ
- A formal language L over Σ is a subset of Σ^*

Examples:

- Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Then the set of all well-formed decimal numbers is a formal language over Σ, and 13 is a word from that language.
- Let $\Sigma = \{a, b, \ldots, z, A, B, \ldots, Z\}$. Then the set of all English words in the *Canterbury Tales* is a formal language over Σ.
- Let $\Sigma = \{1\}$. Then $\Sigma^* = \{\lambda, 1, 11, 111, 1111, \ldots\}$.

Formal Grammars

A grammar for a language describes how to generate words from it. Grammars allow us to finitely describe many infinite languages. Grammars also lead to a classification of formal languages.

Definition: A formal grammar is a tuple (Σ, V, S, P), where

- Σ is a finite, non-empty alphabet (of terminal symbols, letters of words in the formal language we want to describe)
- V is a finite, non-empty alphabet (of non-terminal symbols)
- $S \in V$ (called the start symbol)
- $P \subseteq ((\Sigma \cup V)^* \times (\Sigma \cup V)^*)$ is a finite set of productions (or rules). We write $l \to r$ if (l, r) in P.

Example 1: Assume $\Sigma = \{a, b, c\}$, $V = \{S\}$, $P = \{S \to aSa, S \to bSb, S \to c\}$. Then $G_1 = (\Sigma, V, S, P)$ is a formal grammar.

Example 2: Assume $\Sigma = \{0, 1\}$, $V = \{S\}$, $P = \{S \to 1, S \to 1S, S \to 0S\}$. Then $G_2 = (\Sigma, V, S, P)$ is a formal grammar.

Derivations

Definition: Let $G = (\Sigma, V, S, P)$ be a formal grammar. A G-derivation (or just derivation) is a sequence of words $w_1 \rightarrow w_2 \rightarrow \ldots w_n$ such that

- $w_1 = S$
- For each w_i $(i < n)$, w_i is of the form $\alpha l \beta$, w_{i+1} is of the form $\alpha r \beta$, and there is a production $l \rightarrow r$ in P. In other words, we apply productions as rewrite rules. If $w_n \in \Sigma^*$, we say that the derivation produces w_n.
- The language of a grammar, $L(G)$, is the set of all words produced by G-derivations.

Example 1: $S \rightarrow aSa \rightarrow abSba \rightarrow abcba$ is a G_1 derivation, and hence $abcba \in L(G_1)$

Example 2: $L(G_2)$ is just the language of binary numbers with a leading 1.

Exercises

Find prefix, postfix, and abstract grammar representations for

- 3+2*3-4*8
- 2-2*3-5*(7-3)
- 5*(1+4*(2+3*(3+2*(4+1*5))))

Assume that the precedence of * and / is greater than that of + and -.

Evaluate the above expressions in postfix notation, using the stack method.

Can you characterize $L(G_1)$?

Can you give a grammar that produces exactly all palindromes over $\{a, b, c, d\}$?

Can you give a grammar that produces correct arithmetic expressions over $\{+, -, *, /, sqrt, (,), a, b, c\}$?

CSC519
Programming Languages
Language Description – Syntax II

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Answers to (some) Exercises (Part 1)

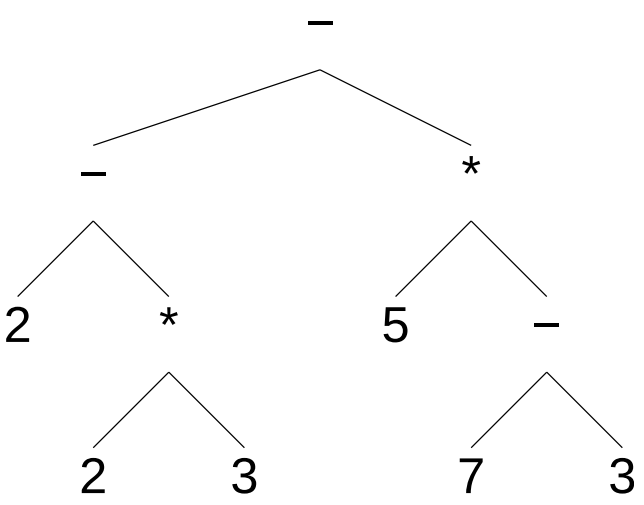
Different representations for 2-2*3-5*(7-3):

First step: Fully parenthesize all subexpressions. All operators are left-associative!
(2 - (2*3)) - (5 * (7-3))

Prefix: - - 2 * 2 3 * 5 - 7 3

Postfix: 2 2 3 * - 5 7 3 - * -

Abstract syntax:



Answers to (some) Exercises (Part 2)

Evaluating 2-2*3-5*(7-3) using the stack method:

Postfix word	Stack (grows to right)
2 2 3 * - 5 7 3 - * -	[]
2 2 3 * - 5 7 3 - * -	[2]
2 2 3 * - 5 7 3 - * -	[2 2]
2 2 3 * - 5 7 3 - * -	[2 2 3]
2 2 3 * - 5 7 3 - * -	[2 6]
2 2 3 * - 5 7 3 - * -	[-4]
2 2 3 * - 5 7 3 - * -	[-4 5]
2 2 3 * - 5 7 3 - * -	[-4 5 7]
2 2 3 * - 5 7 3 - * -	[-4 5 7 3]
2 2 3 * - 5 7 3 - * -	[-4 5 4]
2 2 3 * - 5 7 3 - * -	[-4 20]
2 2 3 * - 5 7 3 - * -	[-24]

Thus the value of the expression is -24

Answers to (some) Exercises (Part 3)

Goal: A grammar that generates all palindromes over $\{a, b, c, d\}$

- Idea: A palindrome stays a palindrome, if we add the same letter on both ends (alternatively: A palindrome stays a palindrome, if we cut off a letter at each end)
- The shortest palindromes have 0 letters or 1 letter

Thus:

- $\Sigma = \{a, b, c, d\}$
- $V = \{S\}$ (Note: The non-terminal always generates a palindrome)
- $P =$

$\{S \to \lambda$	Generate the empty word
$S \to a, S \to b, S \to c, S \to d$	Generate Palindromes of lenght 1
$S \to aSa, S \to bSb,$	Extend palindromes
$S \to cSc, S \to dSd\}$	

$G = (\Sigma, V, S, P)$ is a grammar so that $L(G)$ is the language of all palindromes over $= \{a, b, c, d\}$

Regular Languages

Let $G = (\Sigma, V, S, P)$ be a grammar with the property that all rules in P are of one of the three forms

- $A \to aB$
- $A \to a$
- $S \to \lambda$

for $A, B \in V$, $a \in \Sigma$, and with the restriction that $S \to \lambda \notin P$ or $B \neq S$ for all rules. Then G is called a right-linear or type 3 grammar, and $L(G)$ is a regular or type 3 language.

Examples:

- Any finite language is regular
- The language of valid C identifiers is regular
- The grammar describing floating point number representations in Pascal is regular

Note: Left-linear grammars can be defined similarly (with $A \to Ba$) and describe the same class of languages

Example: Identifiers

In many languages, an identifier must start with a letter, and can contain letters, digits, and underscores

Consider the following definitions:

- $\Sigma = \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9, \_\}$
- $V = \{S, R\}$
- $P = \{S \to \alpha | \alpha \in \{a \ldots, z, A, \ldots, Z\}\}$
  $\cup \{S \to \alpha R | \alpha \in \{a \ldots, z, A, \ldots, Z\}\}$
  $\cup \{R \to \alpha R | \alpha \in \Sigma\}\}$
  $\cup \{R \to \alpha | \alpha \in \Sigma\}\}$

$G = \{\Sigma, V, S, P\}$

Then $L(G)$ is just the language of all valid identifiers

Remark: Regular languages are generally used to define the lexical structure of programming languages

Context-Free Languages

Let $G = (\Sigma, V, S, P)$ be a grammar with the property that all rules in P are of the form $A \to \gamma$ with $\gamma \in (\Sigma \cup V)^*$. Then G is called a context-free or **type 2** grammar, and $L(G)$ is a context-free or type 2 language.

Examples:

- The language of palindromes (over an alphabet with at least two letters) is context-free
- The language of arithmetic expressions is context-free
- Most programming languages are (mostly) context-free

Note: Since context free grammars are a lot more convenient to specify, we often use them even to specify right-linear languages

- We can e.g. have rules of the form **operator**$\to$=> (with a right-linar grammar, we would need two rules)
- Similarly, we can write
  statement$\to$`if` **expr** `then` **statement** `else` **statement** `fi`
  Using right-linear grammars for the finite language of keywords is a lot more cumbersome

Other Types of Languages

Context-sensitive (or type 1) grammars allow rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ with $\alpha, \beta \in (\Sigma \cup V)^*$ and $\gamma \in (\Sigma \cup V)^+$ (i.e. γ is non-empty). $S \rightarrow \lambda$ is allowed, if S does not occur in any right-hand side of a rule. Context-sensitive grammars describe context-sensitive (or type 1) languages

Unrestricted or type 0 grammars allow arbitrary rules and generate type 0 or recursively enumerable languages (Note: these are in general not decidable anymore)

Finally, there are formal languages that are not generated by any formal grammar

The Chomsky-Hierarchy

All languages of type i are also of type i-1:

- Type 3 languages $\subset$ Type 2 languages $\subset$ Type 1 languages $\subset$ Type 0 languages $\subset$ Arbitrary languages
- All of these inclusions are proper

The different types of languages thus form a hierarchy, named after Noam Chomsky, who first described it in 1956

An equivalent classification can be achieved by distinguishing between different types of automata that accept a given language

The existance of a grammar G of type i proves that $L(G)$ is of type i. It does not prove that $L(G)$ is not of type i+1. This requires different proof techniques (usually based on automata theory)

Lexical Syntax

Programming languages are typically languages over an alphabet that is a subset of all ASCII characters (some languages allow other character sets, like EBDIC or Unicode)

A grammar for a programming language typically has two parts:

- A regular language describes valid tokens, using characters as terminal symbols
- A context-free grammar describes how to build programs, using tokens as terminal symbols

Typical token types are:

- Language keywords (`if`, `then`, `while`, . . .)
- Operators: `>`, `<`, `<=`, `=`, . . .
- Identifiers
- Numbers
- Delimiters like `(`,`)`,`{`, `}`, `;`,. . .

If a token type corresponds to more than one string, we distinguish between different occurances using a subscript: **identifier**$_{\texttt{a}}$, **integer**$_{12}$

Example: Arithmetic Expressions

We again consider arithmetic expressions over the operators $\{+, -, *, /\}$, identifiers that are made from lower case letters, (positive) integer numbers, and parentheses $\{(,)\}$

The lexical structure is described by the following rules (non-terminals are written in **bold face**, terminals in `typewriter font`)

openpar$\rightarrow$`(`
closepar$\rightarrow$`)`
mult$\rightarrow$`*`
div$\rightarrow$`/`
plus$\rightarrow$`+`
minus$\rightarrow$`-`
integer$\rightarrow$`d` (`d`$\in \{$`0`$, \ldots,$ `9`$\}$)
integer$\rightarrow$`d` **integer** (`d`$\in \{$`0`$, \ldots,$ `9`$\}$)
identifier$\rightarrow$`c` (`c`$\in \{$`a`$, \ldots,$ `z`$\}$)
identifier$\rightarrow$`c` **identifier** (`c`$\in \{$`a`$, \ldots,$ `z`$\}$)

To make a valid grammar, we would have to add rules like **token**$\rightarrow$**openpar** for each non-terminal symbol above and make **token** our start symbol

Example (continued)

The context-free part of our language can e.g. be described by:

expr→ **expr plus term**
expr→ **expr minus term**
expr→ **term**
term→ **term mult factor**
term→ **term div factor**
term→ **factor**
factor→ **identifier**
factor→ **integer**
factor→ **openpar expr closepar**

The start symbol is **expr**

Example derivation: **expr**→ **expr plus term**→ **term plus term**→ **factor plus term**→ **identifier plus term**→ **identifier plus term mult factor**→ **identifier plus factor mult factor**→ **identifier plus integer mult factor**→ **identifier plus integer mult identifier**→ `a`**identifier plus integer mult identifier**→ `ab`**plus integer mult identifier**→ `ab +`**integer mult identifier**→ `ab + 1`**mult identifier**→ `ab + 1 *`**identifier**→ `ab + 1 * c`

Exercises

Find a good set of tokens for PL and a right-linear grammar that describes them

Find a simpler grammar for arithmetic expressions, describing the same language as in the last example

Find a (context-free) grammar for PL. Are there any parts you cannot describe in the grammar?

CSC519
Programming Languages
Language Description – Syntax III

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Backus-Naur-Form

BNF is a more convenient way to write context free grammars, introduced for the Algol-60 report

Conventions:

- Non-Terminals are enclosed in angle brackets: <program>
- Terminals are written in plain text (I'll normaly use `typewriter font`) or enclosed in single quotes for emphasis (important if they could also be misunderstood as meta-symbols)
- Both terminals and non-terminals are only defined implicitely
- The start symbol usually is the first non-terminal occurring (or obvious from the context)
- Different productions for the same non-terminal are combined into one rule, using the vertical bar (|) to separate alternatives
- BNF uses the symbol `::=` ("defined as" or "can be") instead of $\rightarrow$ for productions
- <empty> is used to denote the empty string (λ)

Note: If you check different language reports, you will very likely find different versions of BNF!

BNF Example

A real number is an (optionally empty) sequence of digits, followed by a decimal point, and a fractional part, i.e. another sequence of integers. In BNF:

1. <real-number> ::= <integer-part> '.' <fraction>
2. <integer-part> ::= <empty> | <digit-sequence>
3. <fraction> ::= <digit-sequence>
4. <digit-sequence> ::= <digit> | <digit> <digit-sequence>
5. <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Note that we could e.g. replace rule number 2 with two separate rules:

2a. <integer-part> ::= <empty>
2b. <integer-part> ::= <digit-sequence>

If we do this for all rules with alternatives, we are mostly back to the original context free grammars

From Derivations to Parse Trees

Derivations only tell us which words we can derive, not the structure defined by the grammar

- Consider the production <sum>::=<sum> - <sum> | a
- Derivation 1:<sum> → <sum>-<sum> →
  a - <sum> → a - <sum> - <sum> →
  a - a - <sum> →
  a - a - a
- Derivation 2:<sum> → <sum>-<sum> →
  <sum> - a → <sum> - <sum> - a →
  a - <sum> - a → a - a - a

- The two derivations yield the same word, but correspond to (a-a)-a and a-(a-a)

Parse trees make the structure of the grammar visible!

Parse Trees

Definition: A partial parse tree (with respect to a Grammar G) is an ordered tree with the following properties:

- All internal nodes are labeled with a non-terminal symbols, and the root is labeled with the starting symbol
- All leaves are labeled with either terminal or non-terminal symbols
- If a node labeled with a non-terminal has any children, then it is labeled with the left-hand side of a production from G, and the children, from left to right, form (one of the alternatives of) the right hand side

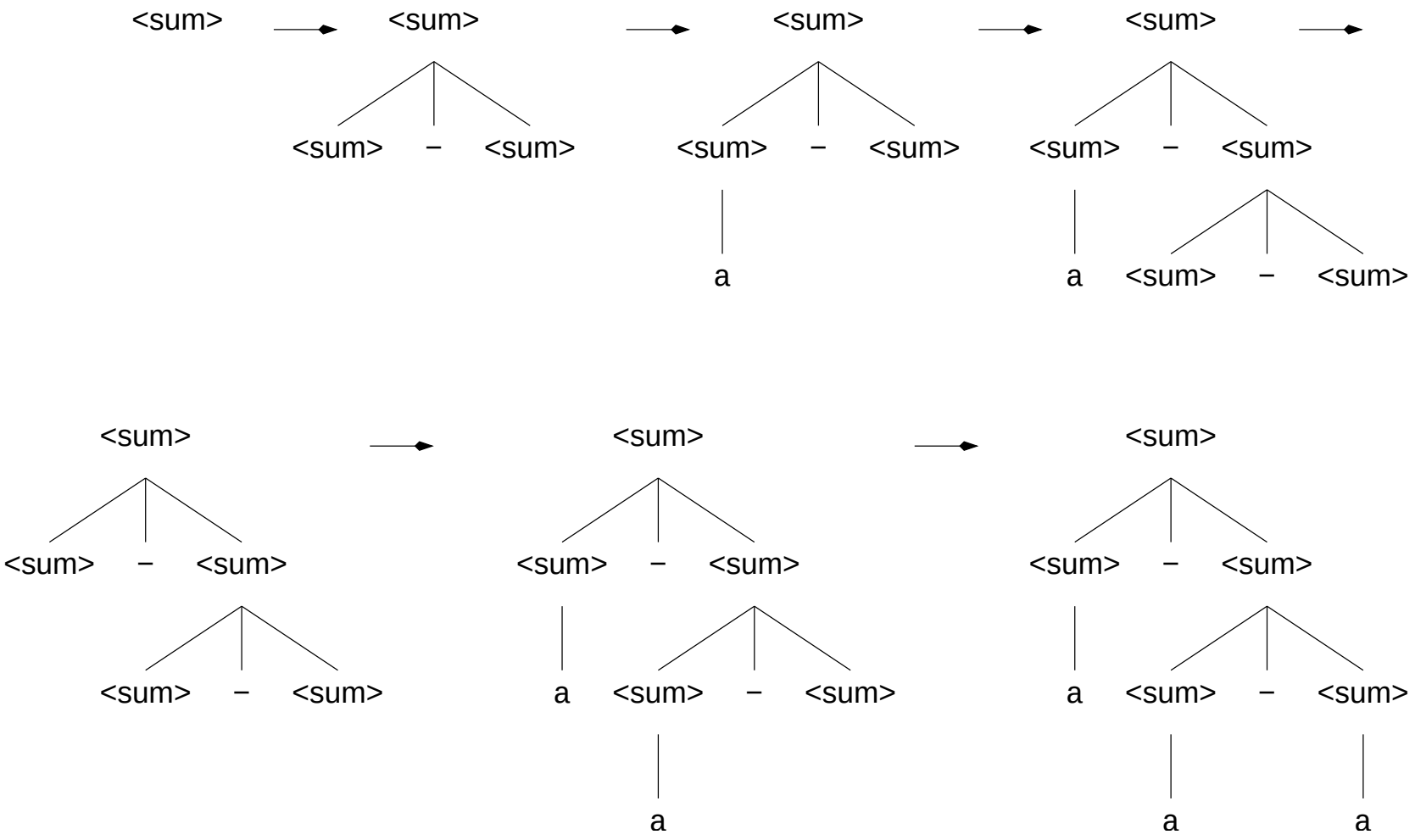
Definition: A (proper) parse tree is a partial parse tree with the property that all leaves are labeled with terminal symbols.

Definition: A parse tree generates the word that is obtained by reading the terminal symbols at its leaves from left to right

A word is in L(G) if and only if there is some parse tree that generates it!

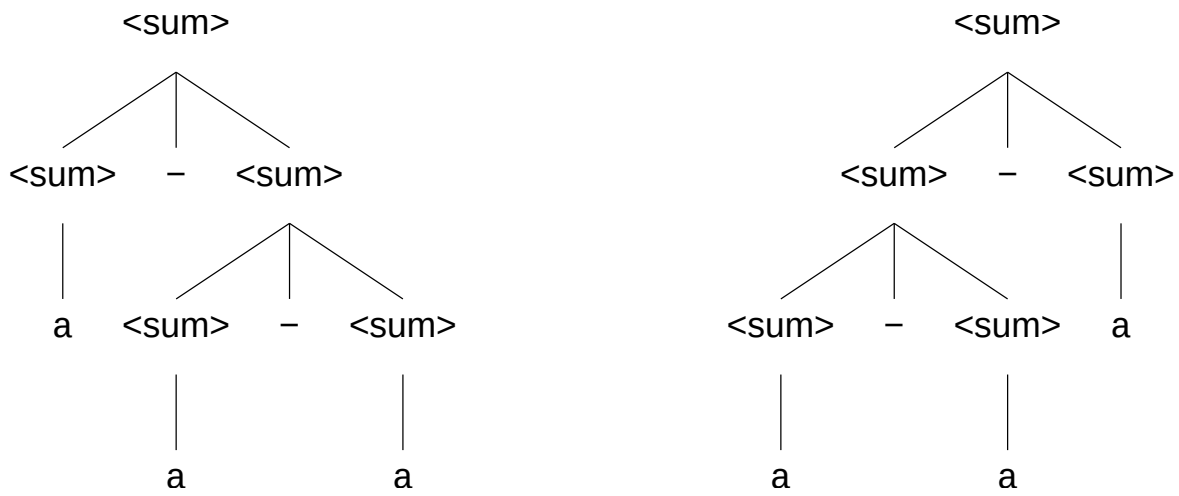
Parse Tree Example

Lets look at the derivation of a-a-a again:



Parse Tree Example (continued)

However, we have two deriviations generating different parse trees:



What now?

Ambiguity

Definition: A Grammar in which a word is generated by more than one parse tree is called ambiguous

We can avoid many ambiguities by carefully designing the grammar

– The rule
  <sum>::=<sum> - <a> | a
  generates the same language as the rule
  <sum>::=<sum> - <sum> | a
  but gives us only one unique parse tree per word

Other ambiguities must be ruled out by disallowing certain parse trees explicitely

A Famous Ambiguity

```
a:=0;
if 1=2 then
      if 1=1 then
            a:=2;
      else
            a:=3;
print a;
```

A Famous Ambiguity

```
a:=0;
if 1=2 then
      if 1=1 then
            a:=2;
else
      a:=3;
print a;
```

A Famous Ambiguity

```
a:=0;
if 1=2 then
      if 1=1 then
            a:=2;
      else
            a:=3;
print a;
```

```
a:=0;
if 1=2 then
      if 1=1 then
            a:=2;
else
      a:=3;
print a;
```

This is called the dangling else ambiguity

- It's caused by a production like

  <statement> ::= `if` <expr> `then` <statement>
  | `if` <expr> `then` <statement> `else` <statement>
- Both C and Pascal grammars have this ambiguity!

It is solved by always making an `else` bind to the nearest unmatched `if`

A Famous Ambiguity

```
a:=0;
if 1=2 then
      if 1=1 then
            a:=2;
      else
            a:=3;
print a;
```

```
a:=0;
if 1=2 then
      if 1=1 then
            a:=2;
else
      a:=3;
print a;
```

This is called the dangling else ambiguity

- It's caused by a production like

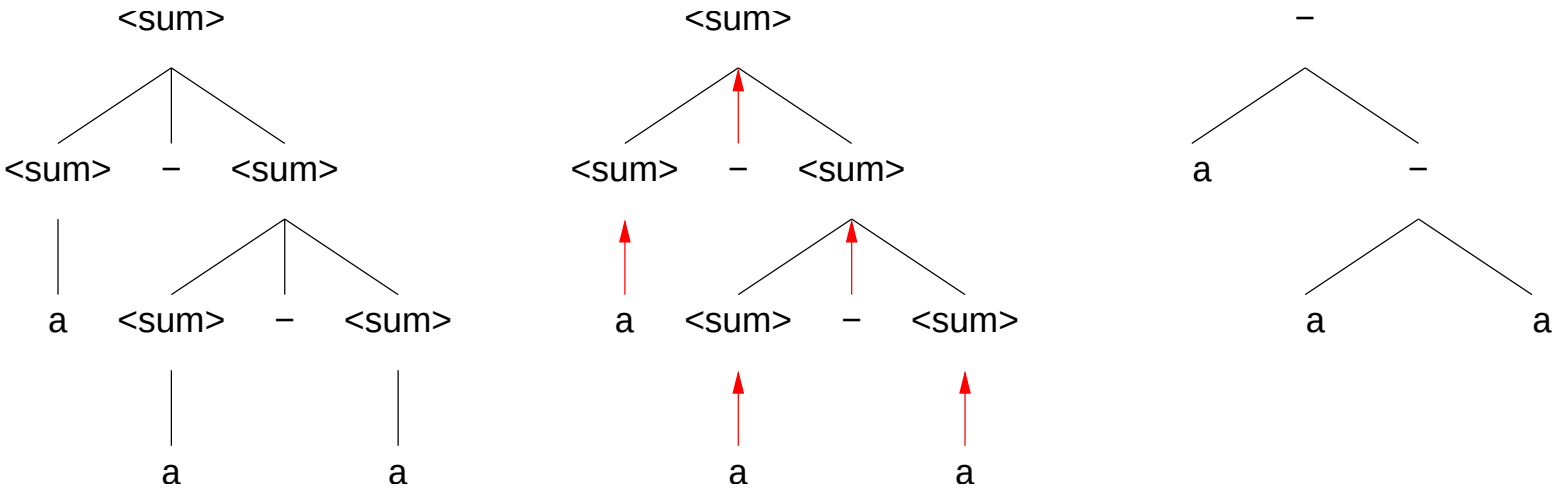
  <statement> ::= `if` <expr> `then` <statement>
  | `if` <expr> `then` <statement> `else` <statement>
- Both C and Pascal grammars have this ambiguity!

It is solved by always making an `else` bind to the nearest unmatched `if`

Parse Trees and Abstract Syntax

Parse trees reflect concrete syntax, i.e. they generate actual words from the language

Well designed grammars have parse trees that closely correspond to the abstract syntax of the language:



This does not always work quite as smoothly ;-)

– We may e.g. need to consider an operator **if-then-else** with three arguments or **if-then** with two arguments, each swallowing terminals from multiple children nodes

Extended BNF

EBNF extends BNF with operators for optional constructs, grouping, and repetitions (similar to regular expressions for right-linear languages)

- Constructs included in curly brackets ({, }) can appear zero or more times
- Constructs in square brackets are optional (i.e. they can appear zero or one times)
- Parentheses can be used to group constructs
- Since there are more meta-symbols, it is more often necessary to use quotes to mark terminals

Example:

1	<sum>	::=	<real-number> {('+' | '-') <real-number>}
2.	<real-number>	::=	[<digit-sequnce>] '.' <fraction>
3.	<fraction>	::=	<digit-sequence>
4.	<digit-sequence>	::=	<digit> {<digit>}
5.	<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Note: EBNF is (sometimes) more convenient, not more powerful!

BNF is often better for writing parsers, EBNF for learning the language

Exercises

- Write the expression grammar from last lecture in BNF and EBNF (using the EBNF features, of course)
- Find parse trees for some expressions with the BNF version
  - Are there any ambiguities?
  - Do the parse trees correspond to the abstract syntax trees? What about associativity and precedence?
  - Can you derive any general rules for good expression grammars?
- How do you generate parse trees from the EBNF grammar?

CSC519
Programming Languages

Language Implementation
Scanning and Parsing

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Compiler Architecture

Source handling

- Input: Program text in the source language
- Handles technical issues like opening and closing files, allowing for look-ahead, maintaining character positions for error messages, . . .

Lexical analysis Scanner, Lexer

- Input: Sequence of characters
- Output: Sequence of `(token,value)` pairs

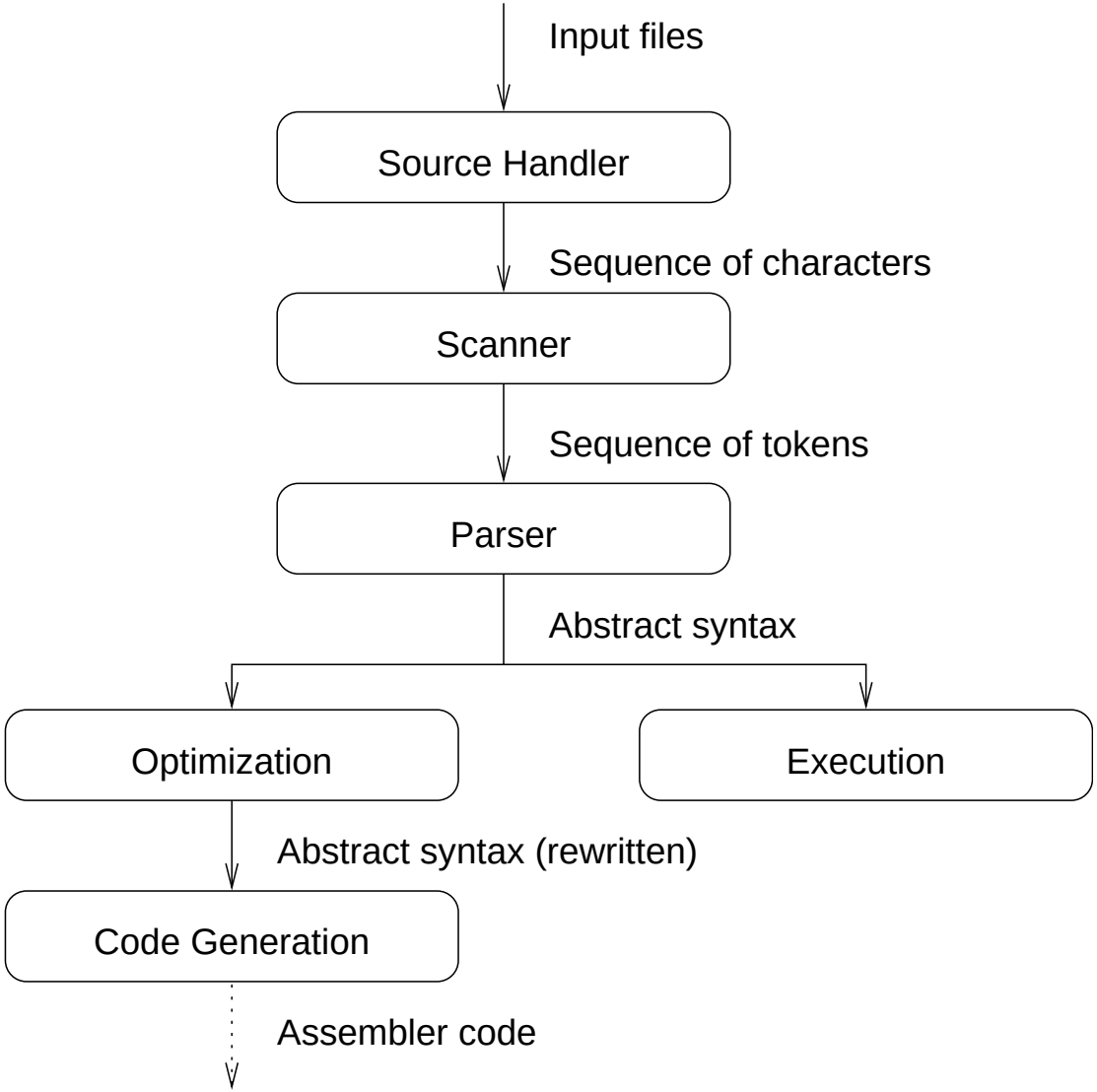
Parse tree construction and checking (Parser)

- Input: Sequence of tokens
- Output: Parse tree or abstract syntax tree

Code generation stages

- Input: Parse tree or abstract syntax tree
- Output: Program in the target language
- Normaly multiple optimization stages

Compiler/Interpreter Architecture



Compiler Architecture 2

The different phases can be interleaved with each other or be implemented in separate passes of the compiler

- Normally, source handling and scanning are integrated
- Scanning and parsing may be integrated ("give me the current/next token"), or the source code may be translated into a sequence of tokens (represented e.g. as a linked list or written to disk)
- Code generation normally is realized in multiple independent stages:
  * Machine-independent optimization (i.e. common subexpression optimizations)
  * Code-generation
  * Machine dependent optimization (e.g. peephole optimization)

Lexical Analysis

Purpose: Transform sequence of characters into sequence of tokens:
`{int var; var = 0; return var;}` $\Longrightarrow$
openbrace, reserved-int, semicolon, ident$_{var}$, assign, integer$_0$, semicolon, reserved-return, ident$_{var}$, semicolon, closebrace

Formal mechanism: Finite Automata

- A finite automaton consists of a finite number of states and a finite number of transistions between states
- Each of the states represents a set of possible tokens, the start state represents the set of all possible tokens
- A subset of states is called accepting. Accepting states correspond to a single token type
- Each transistion is labelled with a character (or character set)
- The automaton starts in the start state and processes input character by character. Each character triggers a transition to a new state
- If the automaton reaches an accepting state, the token type has been determined

FA Example

Assume C style expressions with parenthenses, `|`, `||`, `&`, `&&` and identifers as token types

(Example: `inp1 | (feedback && inp2)`)

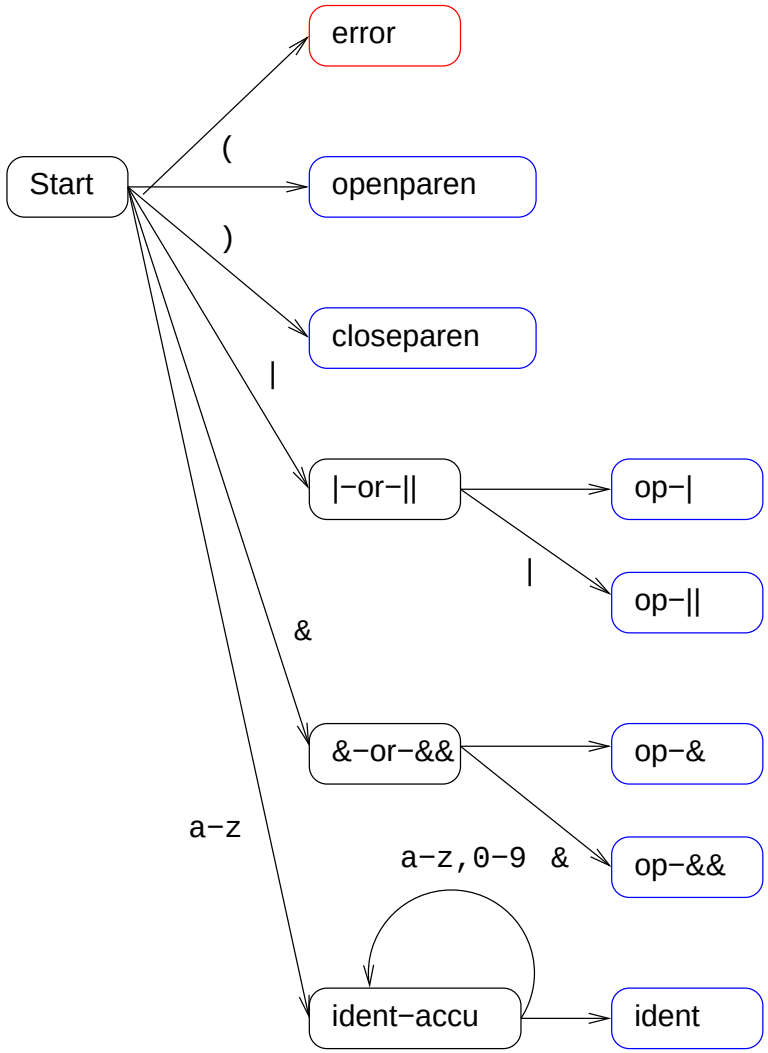
Token types are:

- **openparen** ("(")
- **closeparen** (")")
- **op-**`|` ("|")
- **op-**`||` ("||")
- **op-**`&` ("&")
- **op-**`&&` ("&&")
- **ident** (`a--z{a–z0–9}`)

Convention: Transistions without explicit label are assumed to be labelled with all characters not mentioned in another transistion from the same state

FA Example (contd.)

Automaton:



Two Problems

Sometimes it we cannot determine if a token has ended without looking at further characters

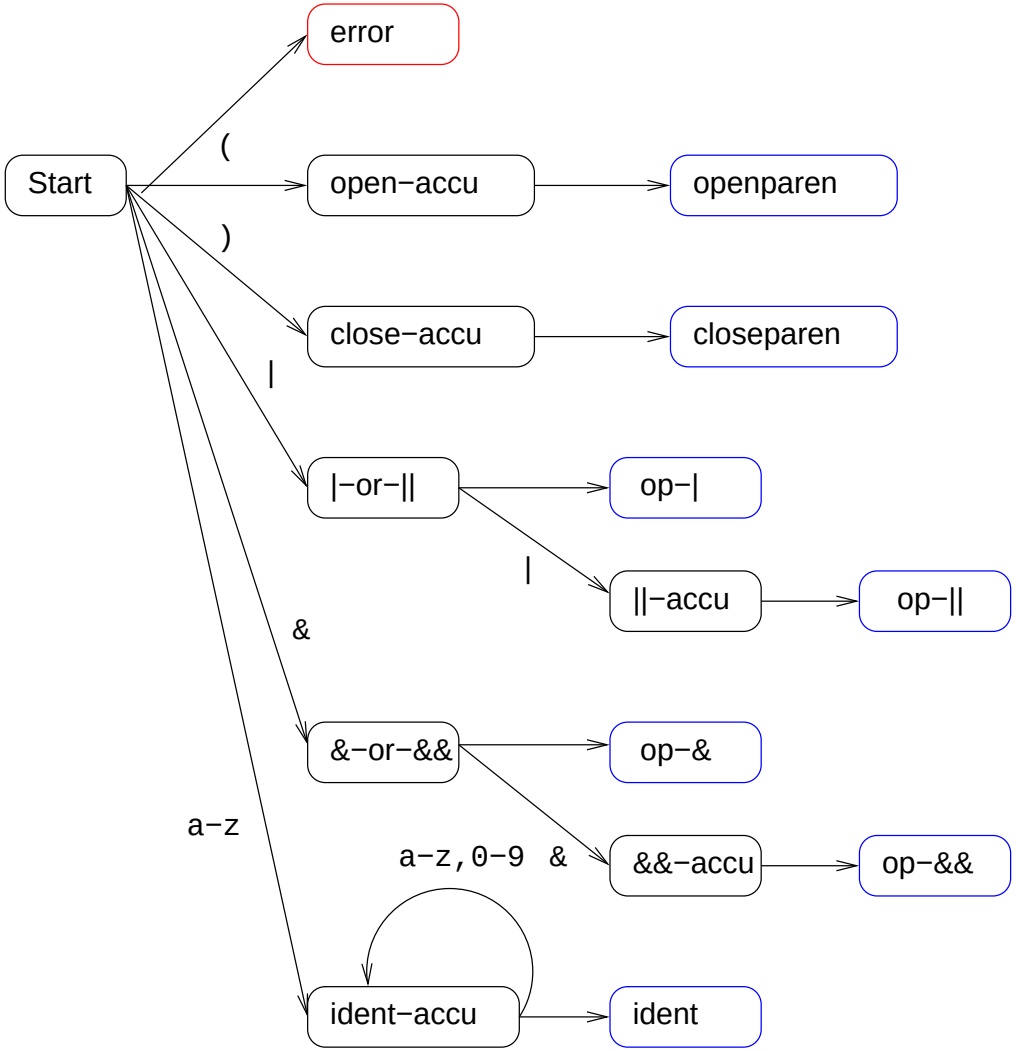
- This is called look-ahead
- Most existing languages can be tokenized with a one-character look-ahead

Some token may be a proper substring of another token

- `|` is a substring of `||` in our example
- Example from C or Pascal: Many reserved words like `while`, `if`, `for`, can also start an normal identifier
- `>` is a substring of `>=`
- Solution: These conflicts are normally resolved by going to the `longest match`, i.e. the longest string compatible with any token determines the token type

FA Example (better)

Automaton:



Coding

Finite automata can be translated into code automatically

- UNIX tools **lex/flex** available
- Normally results in quite big automata, especially if language has many reserved words
- Limited language selection (C/C++, others available, but may differ)

Manual scanner coding can take many shortcuts

FA Example (Scanner Code)

```
function scan()
   case(current) of
   '(':   nextchar(); return openpar;
   ')':   nextchar(); return closepar;
   '|':   nextchar();
          if(current == '|') nextchar(); return op-||;
          else return op-|;
   '&':   nextchar();
          if(current == '&') nextchar(); return op-&&;
          else return op-&;
   'a-z': scan_ident();
   defaut: error("Illegal character");

function scan_ident()
   spelling = "";
   while(current in 'a-z0-9')
      spelling = append(spelling, current); nextchar();
   return (ident,spelling);
```

Parsing

Note: We assume the input to be in tokenized form from now on

Aim: Given a word $p \in L(G)$, find a parse tree for p

Naive solution: Enumerate all possible deriviations until a tree is found that corresponds to p

– Conceptually simple
– Not computationally feasible

Alternative: Bottom-up or Top-down parsing, guided by the word (program, expression) in question

Recursive Descent Parsing (Top-Down)

A recursive descent parser tries to generate a parse tree starting at the start symbol

Typically, each non-terminal (i.e. non-token) is represented by a single function

- The function checks which rule is aplicable for the non-terminal and recursively calls the functions corresponding to its constituents
- If multiple rules are applicable, the parser may need to look ahead to resolve the conflict
- Alternatively, it may guess an alternative and backtrack if it fails (expensive!)
- In practice, all programming languages can be described with adequate gramars that can be parsed with one token look ahead

Recursive descent parsers are easy to write and adequate for most tasks!

Recursive Descent Example

Assume Scheme-like expressions over identifiers and `+`, `-`, `*`, `/` as given by the following grammar:

```
   <expr>   ::=   ( + <arglist> ) | ( - <arglist> ) | ( * <arglist> ) |
                  ( / <arglist> ) | <identifier>
 <arglist>  ::=   <expr> | <expr> <arglist>
```

Example expressions:

- `(+ a b c)`
- `(* (/ a b) (- a1 (* a2 a3 a4)))`

Conventions:

- `current` is the current token
- `look-ahead` is the next token
- `token.type` is the token type (`+`,`-`,`*`,`/`,`(`,`)`, `<identifier>`)
- `token.spelling` gives the token spelling (e.g. the actual identifier)
- `accept(token)` will discard the current token and move to the next if the current token matches the argument. Otherwise it will terminate with a syntax error message

Pseudo-Code

```
function parse_expr()
   if(current.type == <identifier>)
      spelling = current.spelling;
      accept(<identifier>);
      return (<identifier>, spelling);
   else
      accept('(');
      if current == '+'
         return ('+' parse_arglist());
      elseif current == '-'
         return ('-' parse_arglist());
      elseif current == '*'
         return ('*' parse_arglist());
      elseif current == '/'
         return ('/' parse_arglist());
      accept(')';
```

Pseudo-Code (contd.)

```
function parse_arglist()
   part1 = parse_expr();
   if(look-ahead.type == ')')
      return part1;
   else
      return append((part1) parse_arglist());
```

Hint: Scheme is very easy to parse ;-)

Assignment (also see web site)

Give a BNF grammar for PL, denoting which non-terminals should be lexical tokens and which ones have to be handled by the parser.

You do not need to cover the aspects that are not readily handled by a context-free grammar (in particular, the restrictions on labels and gotos).

CSC519
Programming Languages

Imperative Programming
Introduction to C

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

The Imperative Programming Paradigm

The program is viewed a (sequential) list of instructions, executed in a certain order

- The order of excution is normally fully deterministic
- Search has to be programmed explicitely

Instructions change the (global) state of the program by

- Destructively changing the value of some variable (memory location)
- Changing the control flow of the program

Non-destructive computations, as e.g. the evaluation of side-effect free expression, are only temporary, and serve just to determine how to change the global state

State

The state of a running program can be split into:

- Values stored in variables accessible to the program (including dynamically allocated data structures, if the language allows for those)
- Control flow state, including the current point of control (sometimes called the program counter), and, in languages that allow for recursion, the calling sequence for functions and procedures (normally represented by some kind of stack

Assignments change the first part of the state, flow control constructs change the second part

History and Background

Imperative language naturally arose from the requirements of early hardware

- Most computers ever build have an inherently imperative paradigm: Instructions modify registers (including the PC) and memory locations
- As a consequence, all (existing) assembler languages can be seen as imperative

FORTRAN was the first higher-level language, and is perhaps the best example of a purely imperative language

- FORTRAN originally allowed no recursion, and had few control flow statements except for `goto` and (non-recursive) procedure activation

Problems with the design of large, reliable systems led to the rejection of unstructured use of `goto` (Dijkstra, "Go To Statement Considered Harmful", CACM, March 1968) and the desire for more structured languages, ultimately resulting in

- The Algol family, leading Pascal and Modula-2, of "designed" languages, introducing many important features
- The C family of "hacked" languages, still going strong and having significant influence on modern object-oriented languages like C++ and Java

Design Criteria

Efficiency

- Imperative languages map naturally to past and current hardware (and are likely to map well onto hardware build in the forseeable future, too)
- This advantage has generally made imperative language designers striving for efficiently executable language features

Understandability/Structure

- Static language features should illustrate the dynamic behaviour of the program, i.e. it should be easy to understand the dynamic control flow from the static program text
- The language should support abstraction and infomation hiding, i.e. a separation of interface and implementation for new procedures and datatypes

Design Criteria II

Scope of Application

- Language designers aim at a wide range of applications
- Since imperative languages map well to hardware, some language include low level constructs for manipulating hardware directly
- On the other hand, wide portability of code written in the language is achieved by a strong abstraction from the hardware

Imperative programming is still the predominant programming paradigm, followed by its off-shot, object-oriented programming

Statements and Expressions

An imperative program is a sequence of statements, possibly broken up into individual procedures or functions

We can distinguish between two types of statement:

- Actions directly change the value of an explicitly accessible element of the state. The prototypical action is an assignment, giving a new value to a variable
- Flow control statements only change implicitely stored elements, such as the program counter or the stack. The include `if` and `goto`

In most languages, expressions are a distinct from statements

- Expressions do not directly change the state of the computation (although they may have side effects that do)
- Rather, expressions provide values to be used in actions

C

C is a structured, imperative/procedural language, originally developed in 1972 to implement UNIX

C actions and basic data types allow low level programming

- Most C datatypes correspond to machine datatypes
- C has a (non-standardized, but in practice universal) mechanism for directly accessing specific memory regions

C's type system allows the same abstractions as the Algol family, but is more pragmatic

C supports enough important structuring elements to allow the implementation of large systems, again often in a very pragmatic way

The C language has a small core, based on just a few principles

- The core language can be learned fast, and the average programmer uses a large part of it
- Much functionality is added by (portable and standardized) libraries

C: Basic Building Blocks

A C proramm consist of

- Variables (that can store values of certain types)
- Functions (parameterized subroutines that can return values, allowing us to use abstraction at the algorithm level)
- Type definitions, introducing new datatypes and thus allowing us to introduce abstractions at the data representation level

C allows us to separate the declaration of a variable/function/type from it's definition

- Typically, declarations tell some things about the interface for interactions with the environment
- Definition describe the internal implementation of the object
- Definitions are compiled into code or cause memory to be reserved

Entities can be declared multiple times (as long as the declarations are compatible), but can be defined only once

A first C program

Consider the following C program

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  printf("Hello World!\n");
  return EXIT_SUCCESS;
}
```

It defines a single function, called `main()`, and taking no arguments

If compiled and executed, it will print `"Hello World!"` to the screen

C Comilation Environment

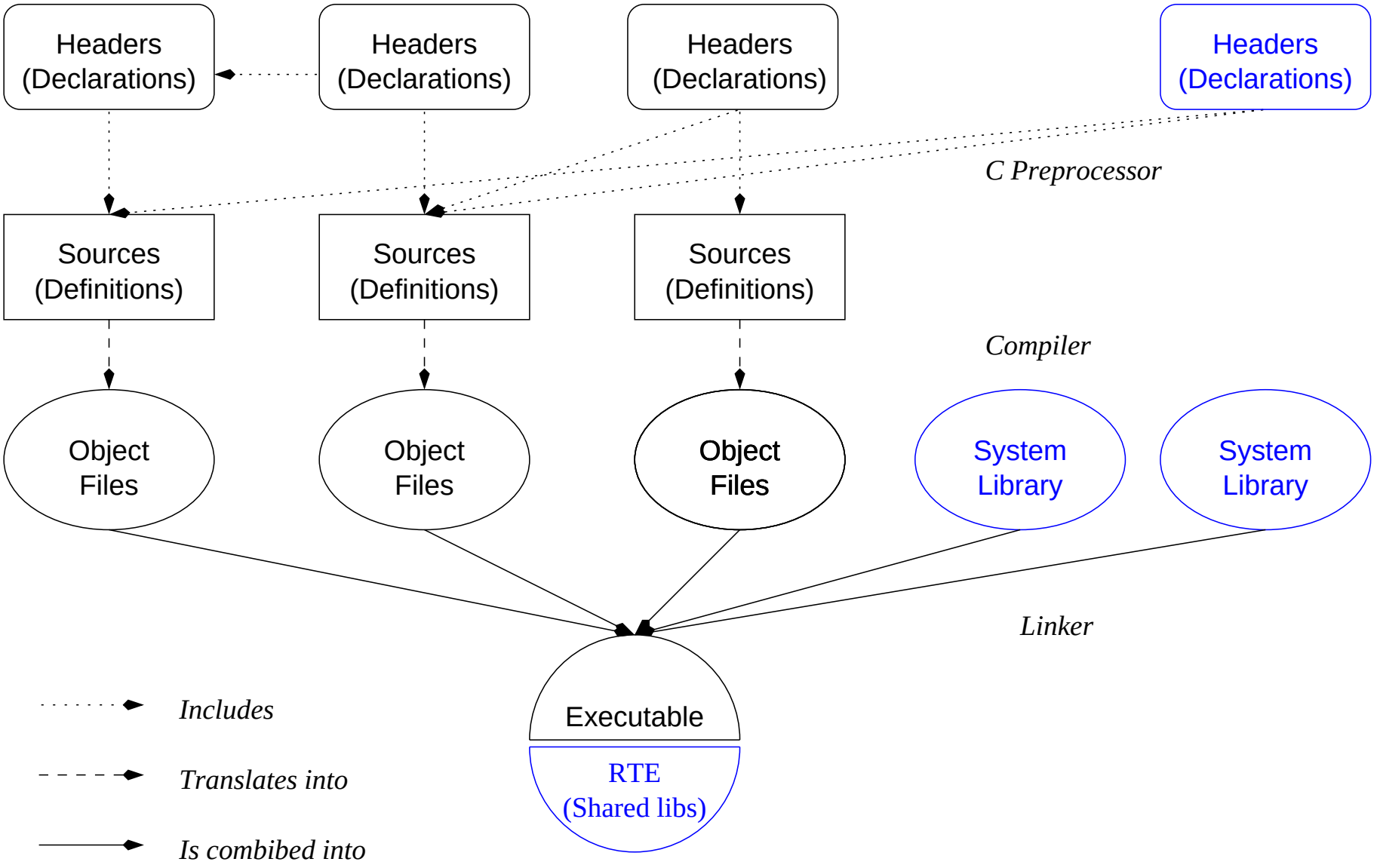
A C program typically is build from multiple source files

- Header-Files, containing mostly declarations (traditionally with the suffix `.h`)
- Implementation files (suffix `.c`)

A combination of header file (defining the interface) and implementation file can be seen as a module

- C does not have an elaborate module system
- Instead, it achieves a similar result using the combination of the C preprocessor (which physically combines header files and implementation files) and the languages scoping rules

The C Compilation Model



Compiling C in the Lab

gcc, the GNU C compiler, is installed on the lab machines

gcc takes care of all stages of compiling:

- Preprocessing
- Compiling
- Linking

It automagically recognizes what to do (by looking at the file name suffix)

Important options:

- **-o <name>**: Give the name of the output file
- **-ansi**: Compile strict ANSI-89 C only
- **-Wall**: Warn about all dubious lines
- **-c**: Don't perform linking, just generate a (linkable) object file
- **-O – -O6**: Use increasing levels of optimization to generate faster executables

The recommeneded way to compile simple programs:
gcc -o progname -ansi -Wall progname.c

Exercises

Type in, compile and and run the "Hello World" program

Vary the program (what happens if you leave out `'\n'`? What happens if you print more and different text, using multiple calls to `printf()`? Does the compiler complain if you leave out one or both of the include statements?)

CSC519
Programming Languages

Basics of C
Flow Control

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

C Fasttrack: Statements, Blocks, and Expressions

C programs are mainly composed of statements

In C, a statement is either:

- An expression, followed by a semicolon ';' (as a special case, an empty expression is also allowed, i.e. a single semicolon represents the empty statement)
- A flow-control statement (`if`, `while`, `goto`,`break`. . .)
- A block of statements (or compound statement), enclosed in curly braces '{}'. A compound statement can also include new variable declarations.

Expressions compute values and may have side effects that change the global state of the computation. An expression is either:

- Elementary (a variable, constant, or function call) or
- A parenthesized expression or
- built by applying an operator to the right number of subexpressions

C Fasttrack: Basic Data Types

C supports (among others) the following important basic data types

- `char` can store small integer numbers. It is typically 8 bit wide and used to store character in the characterset used (often ASCII)
- `int` stores integer numbers that correspond to the natural machine word of the target architecture. Typically, `int` can store 32 bit values
- `long` can store large integer numbers. However, on most common current implementations, `long` is equivalent to `int`
- `float` can store single precision floating point numbers (typically 32 bit, with one bit for the sign, 23 for the mantissa, and 8 bits for the exponent)
- `double` can store double precision floating point numbers (typically 64 bit, 11 bits exponent, 52 bits of mantissa)

All numerical types can be freely mixed in expressions

- If different types are combined in an expression, the "smaller" type is promoted to the "larger" and the result is of the larger type
- If assigned to a smaller type, values are truncated (integers) or rounded (floats)

C Fasttrack: Variables and Assignments

Variables of a certain type correspond to locations in memory that are capable of storing a value of that type

- A valid variable name starts with a letter or underscore (_), and may contain any sequence of letters, underscores, and digits. Reserved words (`if`, `while`,...) cannot be used as variable names
- Capitalization is significant – `a_variable` is different from `A_Variable`

Variable declarations:

- A (simple) variable declaration has the form `<type> <varlist>;`, where `<type>` is a type identifier (e.g. `int`), and `<varlist>` is a coma-separated list of variable names
- A variable declared in a block is (normally) visible just inside that block

The value of a variable (more general, of a lvalue) is changed with the assignment operator "=" (note, single equality sign)

- `a = <expr>` is an expression with the value of `<expr>`
- As a side effect, it changes the value of `a` to that of `<expr>`

C Fasttrack: Some Operators

Operator	Argument Types	Description
!	All basic and pointers	Logical negation, 1 if arg is 0, 0 otherwise
Binary *	Numerical	Multiplication
/	Numerical	Division (truncates on int)
%	Integer type	Division rest
+,-	Numerical, (Pointer)	Addition and subtraction
<, <=, >=, >	Numerical and pointers	Size/Adress comparison Note: All relational operators return 0 if false, 1 if true
==, !=	All basic and pointers	Equality (note two equal signs!), inequality
&&	All basic and pointers	Logical and, 1 if both arguments are not zero, 0 otherwise
||	All basic and pointers	Logical or, 0 if both arguments are zero, one otherwise
=	All basic, pointers, structs	Assignment operator

Note: Table is ordered by precedence, entries between lines share precedence

C Fasttrack: Example 1 (Temperature Conversion)

```
/* A program that prints a Fahrenheit -> Celsius conversion table */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int fahrenheit, celsius;

   printf("Fahrenheit -> Celsius\n\n");

   fahrenheit = 0;
   while(fahrenheit <= 300)
   {
      celsius = (fahrenheit-32)*5/9;

      printf("%3d   %3d\n", fahrenheit, celsius);
      fahrenheit = fahrenheit + 10;
   }
   return EXIT_SUCCESS;
}
```

C Fasttrack: Simple I/O

The C standard guarantees that each program has three I/O streams open when it starts running:

- Standard Input or `stdin` is used for normal input
- Standard Output or `stdout` is used for all normal output
- Standard Error or `stderr` is used for out of band output like warnings or error messages
- Under UNIX, by default all three are connected to your terminal, i.e. `stdin` receives your keyboard input and all output goes to your screen

`printf()` is a function for formatted output and prints to `stdin`

- `printf()` is variadic (it can receive 1 or more arguments)
- The first argument is the control string, containing plain characters (copied to output) and conversion specifiers, starting with %
- Each conversion specifier corresponds to an additional argument, and is replaced by a representation of the corresponding value
- Example: `"%3d"` prints the decimal representation of an integer value, using at least three characters
- For details read **man 3 printf** on one of the lab machines

C Fasttrack: Character-Based I/O

Reading characters from `stdin`: `int getchar(void)`

- `getchar()` returns the numerical (ASCII) value of the next character in the `stdin` input stream
- If there are no more characters available, `getchar()` returns the special value `EOF` that is guaranteed different from any normal character (that is why it returns `int` rather than `char`

Printing characters to `stdout`: `int putchar(int)`

- `putchar(c)` prints the character `c` on the `stdout` steam
- (It returns that character, or EOF on failure)

`printf()`, `getchar()`, `putchar()`, and `EOF` are declared in `<stdio.h>` (a standard library header) and provided by the standard library

C Fasttrack: Example 2 (Copying)

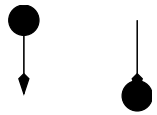
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int c;

   while((c=getchar())!=EOF)
   {
      putchar(c);
   }
   return EXIT_SUCCESS;
}
```

Copies `stdin` to `stdout` – to make a a file copy, use
ourcopy < file > newfile

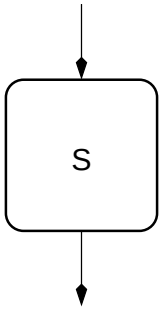
Flow Diagrams

Flow diagrams give a graphical representation of the control flow for a program or construct

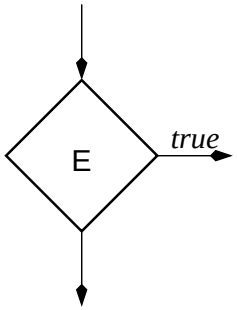
Elements:



Entry point/Exit Point: This is where the conrol flow starts or ends

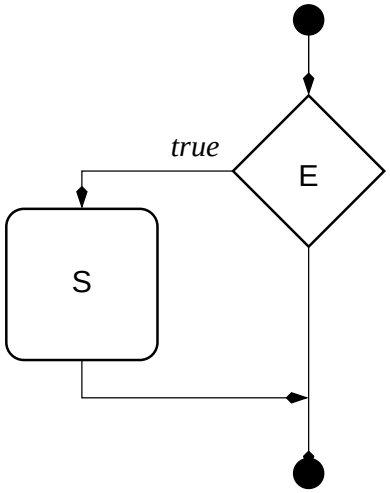


Statement (sequence): Represents an arbitrary sequence of statements, executed sequentially



Conditional branch: Evaluates the expression E, if true, control follows the arrow labeled $true$, otherwise it follows the other arrow

Conditional Execution: `if`



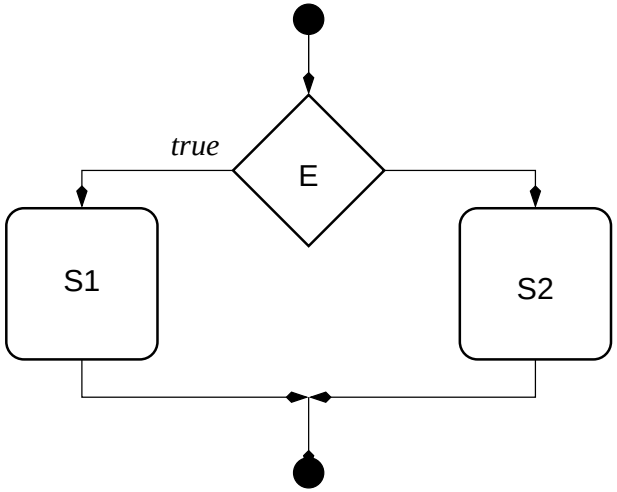
If E evaluates to true, then execute S (else skip it)

C Syntax:

```
if(E)
    S
```

where `E` is any valid expression and `S` is an arbitrary statement (including a block of statements, delimited by curly braces)

Alternatives: `if/else`



If E evaluates to true, then execute $S1$, **else** execute $S2$

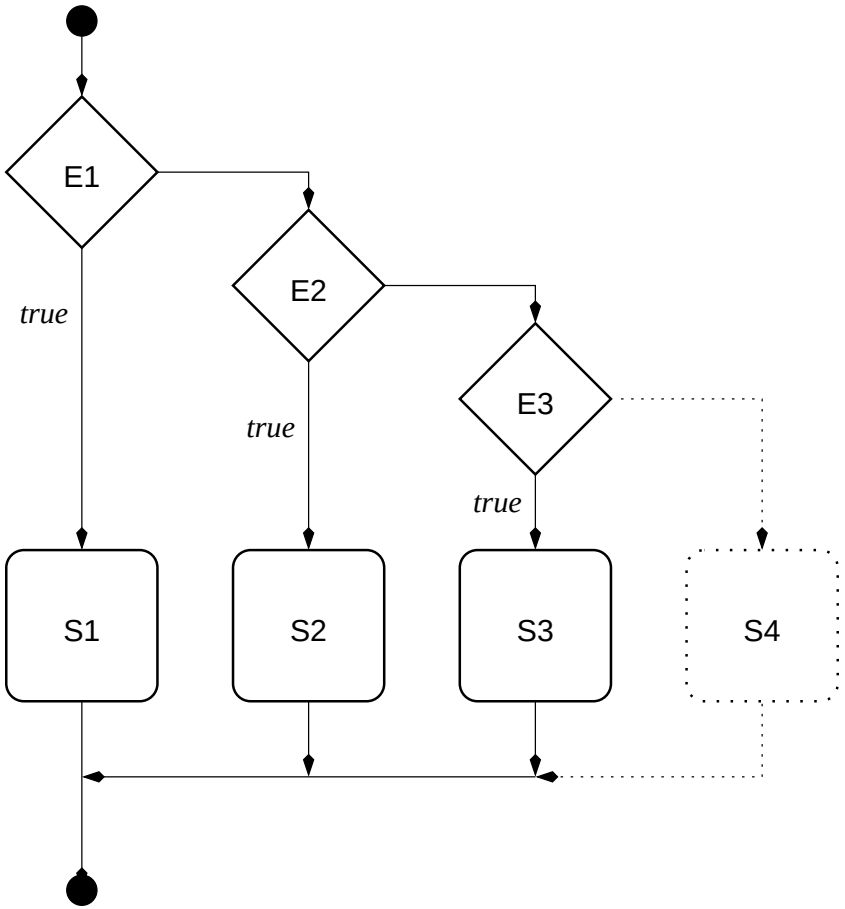
C Syntax:

```
if(E)
    S1
else
    S2
```

where again `S1` and `S2` are arbitrary statements, including statement blocks

Multiple Alternatives

Often, a program has to select between many alternatives, only one of which is applicable:



This can lead to a cascade of ifs:

```
if(E1)
   S1
else
  if(E2)
     S2
  else
     if(E3)
        S3
     [else
        S4]
```

Preferred form:

```
if(E1)
   S1
else if(E2)
   S2
else if(E3)
   S3
[else
   S4]
```

The right hand version is not only more compact, but also shows that the alternatives are at the same level of abstraction.

Exercises

Write a C program that counts lines in the input. Hint: The standard library makes sure that any line in the input ends with a newline (written as the character constant '\n')

Write a C program that prints two tables side by side (equivalent to a 4-column) table, one for the conversion from yards into meters, the other one for the conversion from km into miles. The output should use `int` values, but you can use the floating point conversion factors of `0.9144` (from yards to meters) and `1.609344` from mile to km. Try to make the program round correctly!

Note: Please **ssh** to `lee.cs.miami.edu` to use the lab machines over the net

To change your password on the lab machines, use **yppasswd**. Also check `http://www.cs.miami.edu/~irina/password.html` for the password policy

CSC519
Programming Languages

Flow Control in Imperative Languages

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

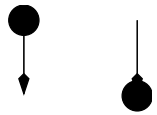
http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

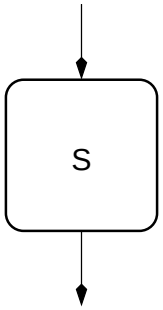
Flow Diagrams

Flow diagrams give a graphical representation of the control flow for a program or construct

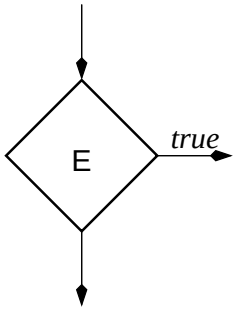
Elements:



Entry point/Exit Point: This is where the conrol flow starts or ends

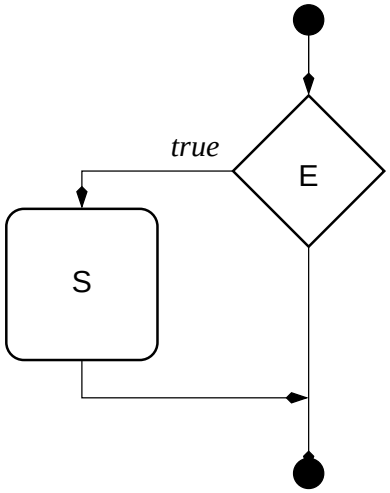


Statement (sequence): Represents an arbitrary sequence of statements, executed sequentially



Conditional branch: Evaluates the expression E, if true, control follows the arrow labeled $true$, otherwise it follows the other arrow

Conditional Execution: `if`



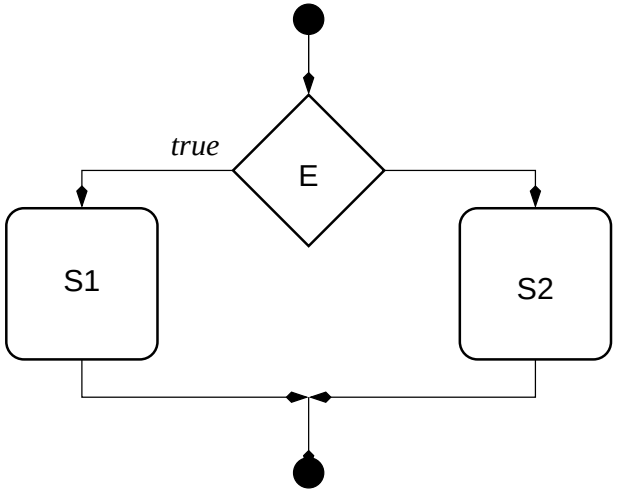
If E evaluates to true, then execute S (else skip it)

C Syntax:

```
if(E)
    S
```

where `E` is any valid expression and `S` is an arbitrary statement (including a block of statements, delimited by curly braces)

Alternatives: `if/else`



If E evaluates to true, then execute $S1$, **else** execute $S2$

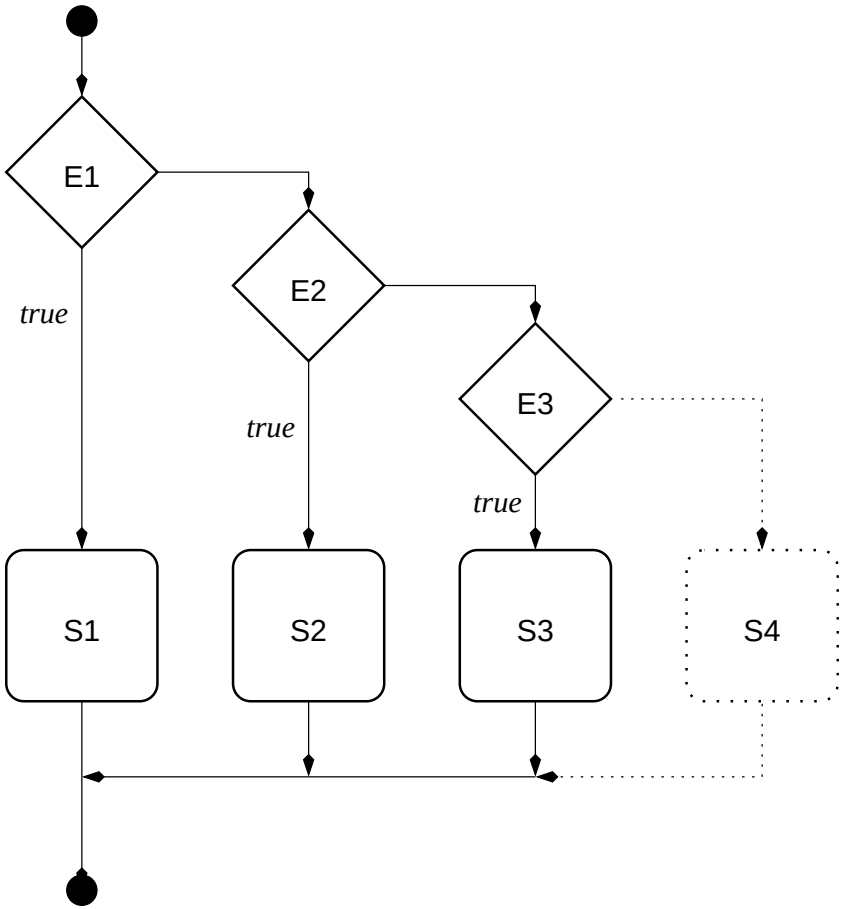
C Syntax:

```
if(E)
    S1
else
    S2
```

where again `S1` and `S2` are arbitrary statements, including statement blocks

Multiple Alternatives

Often, a program has to select between many alternatives, only one of which is applicable:



This can lead to a cascade of ifs:

```
if(E1)
   S1
else
  if(E2)
     S2
  else
     if(E3)
        S3
     [else
        S4]
```

Preferred form:

```
if(E1)
   S1
else if(E2)
   S2
else if(E3)
   S3
[else
   S4]
```

The right hand version is not only more compact, but also shows that the alternatives are at the same level of abstraction.

Loops

A loop repeatedly executes the same statements

Programming languages can support looping either by

- A combination of arbitrary branching (`goto`) and conditional execution or
- with dedicated loop constructs

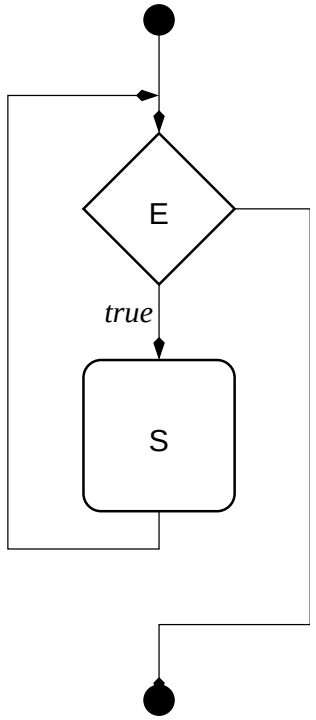
Important questions about loop constructs:

- Does the loop always terminate (after a predetermined number of repetitions)?
  - If a program (or program segment) only contains loop constructs guaranteed to terminate, overall program termination is guaranteed!
  - Potentially infinite loops are necessary for full Turing completeness
  - Potential infinite loops also are (practically) indispensibe for server-type programs or programs that read arbitrary sized input
- Is the loop body executed at least once or can it be skipped totally?
- Is the loop single entry/single exit?
  - Single entry/single exit allows us to argue about program correctness using loop invariants (assertions about the state that always true at a certain position in the loop)

"Lazy" Loops: `while`

A while-loop is controlled by a controlling expression at the top

- If this expression evaluates to true, the body of the loop is executed and control returns to the controlling expression
- If the expression evaluates to false, the body is skipped, and control is transferred to the first statement following the expression.



Syntax in C:

```
while(E)
   S
```

- S can be a single statement (even the empty statement) or a block of statements

The Initialize-and-Update Problem

Initialize and update in Pascal-like languages:

```
element := read();
while valid(element) do begin
   S;
   element := read()
end
```

– Problem: Elements are read at two disjoint positions, making debugging, understanding and maintaining harder

C-Like languages allow assignments in expressions:

```
while (valid((element=read()))){
   S;
}
```

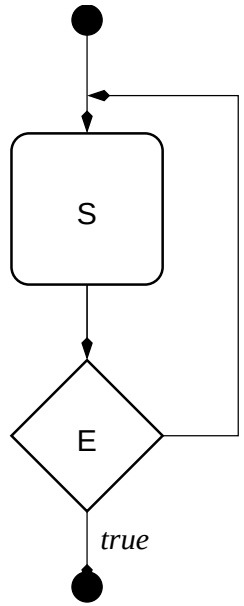
As a result, `while`-type loops (including the C-Style `for` loop, see below) are used much more often in C than in Pascal

Eager Loops (1): `repeat/until`

An alternative loop construct tests the controlling expression after the body of the loop has been executed

In many imperative languages, such a loop has the form of a `repeat-until` loop:

```
repeat
   S
until E;
```



- The body of the loop is executed at least once
- After the body has been executed, the controlling expression is evaluated
- The loop is terminated, if the controlling expression evaluates to true, otherwise control returns to the first statement of the body

`repeat/until` and the initialize-and-update-problem

Note that `repeat/until` allows loops like the following:

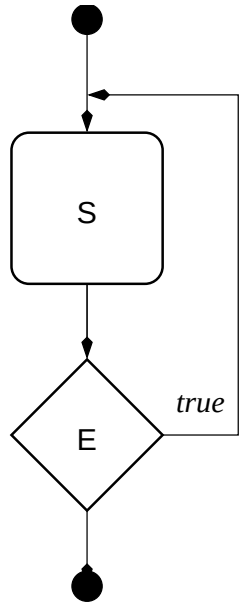
```
repeat
   element := read();
   if valid(element) then
      do something with element
   end
until not(valid(element))
```

This again localizes the `read()` in one place!

Eager Loops (2): `do/while`

C does not support `repeat/until`, but a slight variation: `do/while`

- As in `repeat/until`, the controlling expression is evaluated at the end of the body
- However, the loop terminates if the expression evaluates to false



Syntax:

```
do
   S
while(E);
```

Example: Uniq

We are looking for a program that eleminates successive occurences the same element from a sequence

Eample: 1 1 1 2 3 4 4 1 2 2 $\Longrightarrow$ 1 2 3 4 1 2

Design idea:

– We consider the original sequence as a sequence of sequences of equal characters (where many of the subsequences may have length 1)
– Whenever we encounter a new subsequence, we print the first element and skip all following elements of the same type

Uniq Example: Code

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char first, next;

   first = getchar();
   while(first!=EOF)
   {
      putchar(first);
      do
      {
         next = getchar();
      }
      while(next == first);
      first = next;
   }
   return EXIT_SUCCESS;
}
```

Finite Iteration: Classical `for`

Many programming languages offer `for`-constructs for finite iteration

- Pascal: `for i:=<init> to <limit> do S` (alternative: `downto`)
  - Iterates over integer values between `<init>` and `<limit>`
- AWK: `for(i in <array>)`
  - Iterates over all keys in an associative array
- Sinclair BASIC: `for i=<init> to <limit> step <step>;S;next i`
  - As Pascal version, but allows for a non-unit stride

Note: `i` is called the loop variable or index variable (and can of course be any normal variable name)

- Common idea: The loop variable takes on a new value for each successive traversal of the loop, the loop terminates, if all relevant values have been processed

Classical `for` continued

Important questions:

- What happens if the loop variable is changed in the loop body
- What happens if variables occuring in `<limit>` are modified in the loop body?
- What is the value of the loop variable after termination?
- Is the order of values for the loop variable defined?

Pascal answers:

- It is illegal to modify the loop variable in the loop
- The limit is only evaluated once (before entering the loop), i.e. modifiying any variables does not change the loop behaviour
- The value of the loop variable after the loop is undefined
- The possible values are reached sequentially (increasing for `to`, decreasing for `downto`)

AWK answers:

- If you do stupid things, you have to live with the result (everything is implementation-defined in a cryptic way)

C-Style `for`

A `for` loop in C is a `while`-type loop which allows us to combine initialization, controlling expression and updating in one place

Syntax:
```
for(E1; E2; E3)
    S
```

- When the loop is entered for the first time, `E1` is evaluated (the value is not used, but it typically is an assignment expression initializing a loop variable)
- Then `E2` is evaluated. If it is false, the body is skipped and the loop terminates. If it is true, the body is executed
- After the body has been executed, `E3` is evaluated. Again, the value is discarded, but `E3` typically is another assignment. Control then returns to the top of the loop

The `for` statement in C is `not` restricted to finite iteration

- It generalizes `while`: `for(;E;)S` is equivalent to `while(E)S`
- It also generalizes Pascal for: `for(i=<init> ; i<=<limit>;i=i+1)S`

Example: Uniq with `for`

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char first, next;

   for(first = getchar(); first!=EOF; first = next)
   {
      putchar(first);
      do
      {
         next = getchar();
      }
      while(next == first);
   }
   return EXIT_SUCCESS;
}
```

Exercises

Rewrite the **uniq** example

- Using only `while` loops
- Using only `do/while` loops
- Using only (C style) `for` loops

Can you rewrite any `while`-loop as a `do/while` loop? What about doing it vice-versa?

Can you rewrite any C style `for` loop as a `while` loop?

CSC519
Programming Languages
Flow Control (Continued)

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Example: Uniq with `for`

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char first, next;

   for(first = getchar(); first!=EOF; first = next)
   {
      putchar(first);
      do
      {
         next = getchar();
      }
      while(next == first);
   }
   return EXIT_SUCCESS;
}
```

Search in Loops

A frequent use of loops is to search for something in a sequence (list or array) of elements

First attempt: Search for an element with property `P` in `array`

```
for(i=0; (i< array_size) && !P(array[i]); i=i+1)
{ /* Empty Body */ }
if(i!=array_size)
{
   do_something(array[i]);
}
```

- Combines property test and loop traversal test (unrelated tests!) in one expression
- Property test is negated
- We still have to check if we found something at the end (in a not very intuitive test)

Is there a better way?

Early Exit: `break`

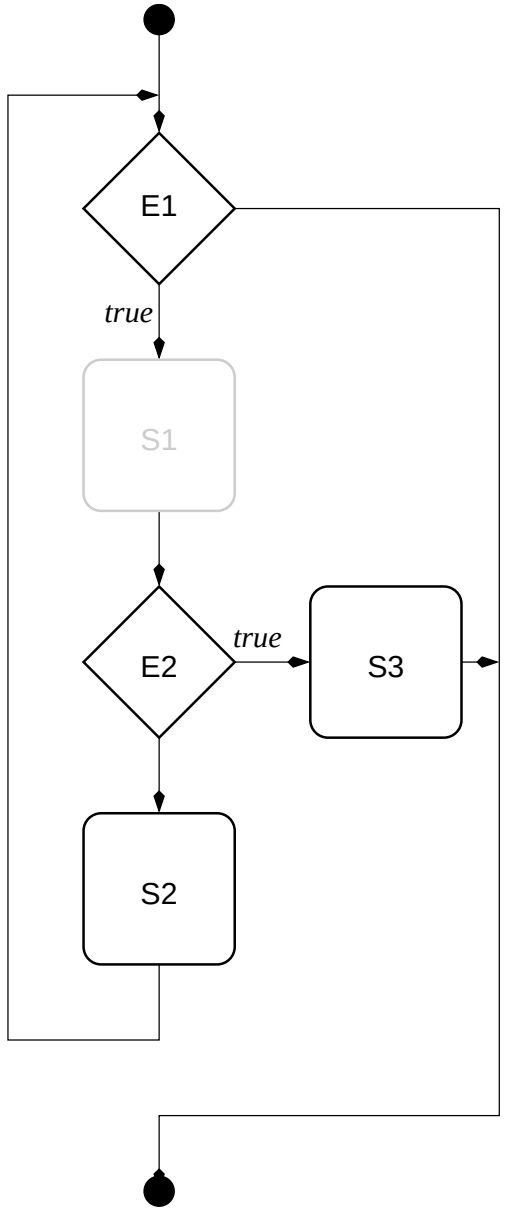
C offers a way to handle early loop exits

The `break;` statement will always exit the innermost (structured) loop (or `switch`) statement

Example revisited:

```
for(i=0; i< array_size; i=i+1)
{
   if(P(array[i])
   {
       do_something(array[i]);
       break;
   }
}
```

- I find this easier to read
- Note that the loop is still single entry/single exit, although control flow in the loop is more complex



`while`-loop with break

Selective Operations and Special Cases

Assume we have a sequence of elements, and have to handle them differently, depending on properties:

```
for(i=0; i< array_size; i=i+1)
{
   if(P1(array[i])
   {
       /* Nothing to do */
   }
   else if(P2(array[i]))
   {
       do_something(array[i]);
   }
   else
   {
      do_something_really_complex(array[i]);
   }
}
```

Because of the special cases, all the main stuff is hidden away in an else

Wouldn't it be nice to just goto the top of the loop?

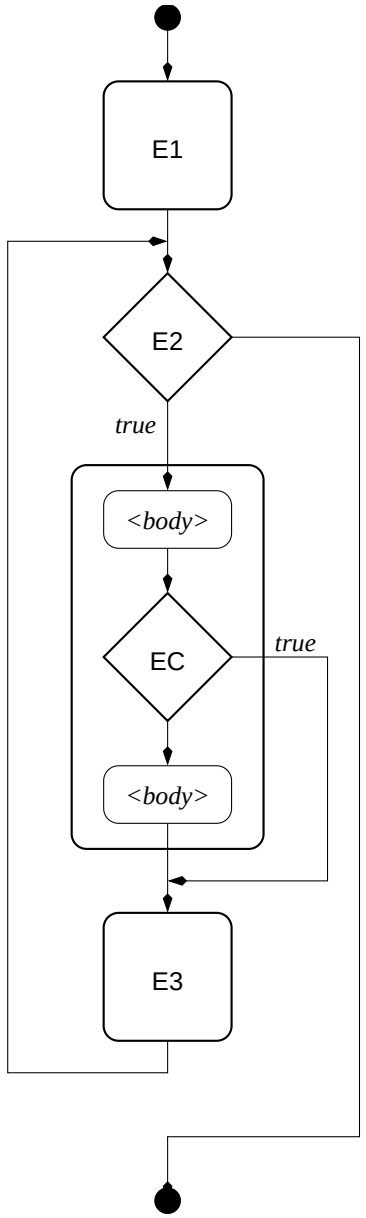
Early Continuation: `continue`

A continue; statement will immediately start a new iteration of the current loop

– For C `for` loops, the update expression will be evaluated!

Example with continue:

```
for(i=0; i< array_size; i=i+1)
{
   if(P1(array[i])
   {
       continue;
   }
   if(P2(array[i]))
   {
       do_something2(array[i]);
       continue;
   }
   do_something_really_complex(array[i]);
}
```



`for` loop with continue

Arbitrary Control Flow: `goto`

Most modern languages still support arbitrary transfer of the control (often only within a function) via `goto`

- Typically, a program can contain labels of the form `label: <statement>`
- A `goto label;` then transfers contol to the statement following the label

Note: Both C and Pascal use this syntax

`goto` is never strictly necessary

- It can always be replaced by using apropriate structured programming constructs

However `goto` can be useful in certain (rare) circumstances:

- Building state machines (e.g. finite automata): Each label corresponds to a state, `goto` is used to implement transistions (particularly important for automatically generated code)
- Common error handling for different cases

Implementation Issues: Loops

Structured loop constructs are hard and expensive to implement directly:

- Since loops are defined by textual scoping rules, it is necessary to search for the begin of a loop
- Hence, machine instruction sets for modern microprocessors do not typically direcly support any flow control constructs except for conditional and unconditional branching (limited `if` and goto)

However, high-level loop constructs are compiled down to efficient constructs

- There is normally no efficiency reason to use `goto`!

Example: Implementing `for`

`for`-loop

```
for(E1; E2; E3)
{
   if(EC)
   {
      continue;
   }
   S;
}
```

Equivalent low-level loop

```
            E1;
   loop:    if(!E2)
            {
               goto end;
            }
            if(EC)
            {
               goto update;
            }
            S;
    update: E3;
            goto loop;
end:
```

Selection of Alternatives

Often, a program has to select between multiple alternatives based on the value of a single variable:

```
if(blip ==  weather_ballon)
{
   ignore();
}
else if(blip ==  airplane)
{
   complain_to_ATC();
}
else if(blip == russian_missile)
{
   press_red_button();
}
else
{
   ufo_alarm();
}
```

Case Distinctions

Many languages offer a dedicated programming construct for case-distinctions like that

- Dedicated constructs may make code easier to read
- In many cases, it can be compiled into faster code than a more general `if/else` chain

C provides the `switch` statement:

```
switch(E)
{
   case val1: S1;
   case val2: S2;
    ...
   default: Sd;
}
```

The Switch-Statement

```
switch(E)
{
   case val1: S1;
   case val2: S2;
    ...
   default: Sd;
}
```

`E` has to be an integer-valued expression (this includes enumeration types)

`val1`, `val2`,. . . have to be constant integer expressions

`E` is evaluated and the result is compared to each of the constants after the case labels. Execution starts with the first statement after the matching case. If no case matches, execution starts with the (optional) default case.

Note: Execution does not stop at the next case label! Use `break;` to break out of the `switch`

Example: `switch`

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int main(void)
{
   int c;
   while((c=getchar())!=EOF){
      switch(c)
      {
      case '\n': printf("\\n"); break;
      case '\t': printf("\\t"); break;
      case '\r': printf("\\r"); break;
      default:   if(isprint(c))
                 { putchar(c); }
                 else
                 { printf("\\%03o",c); }
                 break;
      }
   }
   return EXIT_SUCCESS;
}
```

Potential Problems with C `switch`:

```
switch(blip)
{
case  weather_ballon
   ignore();
case airplane:
   complain_to_ATC();
case russian_missile:
   press_red_button();
default:
   ufo_alarm();
}
```

As usual, C gives you enough rope to hang yourself!

– You can use this to handle special cases and fall through to a common part
– It also is useful to provide the same code for many labels

Other languages execute exacly one statement (or block of statements) for a single label

Excercises

Transform the following loops into goto-loops

- `repeat/until`
- `do/while`
- `while` (assume one (conditional) `break` and one (conditional) `continue`)

Read `http://www.cs.berkeley.edu/~nikitab/courses/cs294-8/hw1.html` and meditate about `switch` in C

CSC519
Programming Languages
C Functions (Intro)/Scanner Datatypes

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

C Fasttrack: Functions

Functions are the primary means of structuring programs in C

A function is a named subroutine

- It accepts a number of arguments, processes them, and (optionally) returns a result
- Functions also may have side effects, like I/O or changes to global data structures
- In C, any subroutine is called a function, wether it actually returns a result or is only called for its side effect

Note: A function hides its implementation

- To use a function, we only need to know its interface, i.e. its name, parameters, and return type
- We can improve the implementation of a function without affecting the rest of the program

Function can be reused in the same program or even different programs, allowing people to build on existing code

C Fasttrack: Function Definitions

A function definition consists of the following elements:

- Return type (or `void`) if the function does not return a value
- Name of the function
- Parameter list
- Function body

The name follows the same rules as variable names

The parameter list is a list of coma-separated pairs of the form `<type> <name>`

The body is a sequence of statements included in curly braces

Example:

```
int timesX(int number, int x)
{
   return x*number;
}
```

C Fasttrack: Function Calling

A function is called from another part of the program by writing its name, followed by a parenthesized list of arguments (where each argument has to have a type matching that of the corresponding parameter of the function)

If a function is called, control passes from the call of the function to the function itself

- The parameters are treated as local variables with the values of the arguments to the call
- The function is executed normally
- If control reaches the end of the function body, or a `return` statement is executed, control returns to the caller
- A `return` statement may have a single argument of the same type as the return type of the function. If the statement is executed, the argument of `return` becomes the value returned to the caller

We can only call functions that have already been declared or defined at that point in the program!

C Fasttrack Example: Factorial

```
#include <stdio.h>
#include <stdlib.h>

int factorial(int value)
{
   if(value <= 1)
   {
      return 1;
   }
   return value*factorial(value-1);
}
int main(void)
{
   printf("8! = %d\n", factorial(8));
   return EXIT_SUCCESS;
}
```

$ gcc -o factorial -Wall -ansi factorial.c
$ ./factorial
8! = 40320
$

A Compiler for PL

Medium term aim: Implementations of PL

Steps:

1. Grammar (Assignment 2)
2. Source handler (Will be made available and discussed)
3. Scanner/Tokenizer
4. Abstract Syntax
5. Parser
6. Code Generator (compile to C)
7. . . . further plans

Recall:

- The source handler delivers the program text as a sequence of characters
- The scanner or tokenizer transforms it into a sequence of tokens
- The parser converts it into an parse tree or abstract syntax tree

Token Representation

The scanner has to break up the input file into a sequence of tokens

What kind of information do we need to store for a token?

- Token type (roughly, the name of the non-terminal corresponding to the token)
- The spelling of the token (a sequence of characters)
- Nice to have: The position where a token starts (two integer numbers, line and column)

Example: `X1<=0;` is a (small and boring) PL program

- One tokenized representation:

```
(Variable,  "X1", 1, 1)
(AssignOp,  "<=", 1, 3)
(NullDigit, "0",  1, 5)
(Semicolon, ";",  1, 6)
```

Let's look at the individual parts one at a time!

Token Type

The type of the token can be exacty one of a limited number of alternatives

The suitable programming language construct for this is an enumeration type

Enumeration Data Types

Enumeration data types can represent values from a finite domain using symbolic names

- The possible values are explictly listed in the definition of the data type
- Typically, each value can be used in only one enumeration

Questions:

- How are enumerations encoded?
- Are different enumeration types compatible with each other?
- Can enumeration types be converted to a basic data type?

In most high-level languages we can actually create enumerations as new types added to the language

Example: `enum` in C

The following code defines an enumeration data type (called `TokenType`) in C:

```
typedef enum
{
   NoToken,
   Variable,
   AssignOp,
   NullDigit,
   Semicolon
}TokenType;
```

We can now declare and assign variables of that type:

```
void testfun(void)
{
   TokenType tok;

   tok = AssignOp;

   printf("tok == Variable = %d\n", (tok == Variable));
}
```

The `enum` Datatype in C

In C, enumerations are created using the `enum` keyword

Enumeration types are integer types

- A definition of an enumeration type just assigns numerical values to the symbolic name
- Unless explicitely chosen otherwise, the symbolic names are numbered starting at 0, and increasing by one for each name
- A variable of an enumeration type can be assigned any `int` value
- Any integer datatype can be assigned an enumerated value
- Hence, all enumeration types are compatible

C enumerations have only mnemonic value, they do not enable the compiler to catch bugs resulting from mixing up different types

Pascal Enumerations

Equivalent definition in Pascal:

```
type TokenType = (NoToken, Variable, AssignOp, NullDigit,
Semicolon);
```

Different enumerations are strictly separate data types

- It is impossible to assign values of an enumeration type to a variable of any other type
- Conversion to numerical values requires an explicit function call (`ord()`)

Safer, but more inconvenient (not unusual, if we compare Pascal and C)

The Token Spelling

The token spelling gives the exact sequence of characters making up the token (sometimes called the literal value or just literal of the token)

A sequence of identical elements can be efficiently represented by an array data type (in this case an array of characters)

Arrays (in C)

A array is a data structure that holds elements of one type so that each element can be (efficiently) accessed using an index

In C, arrays are always indexed by integer values

Indices always run from 0 to some fixed, predetermined value

`<type> <var> [<elements>];` defines a variable of an array type:

- `<type>` can be any valid C type, including user-defined types
- `<var>` is the name of the variable defined
- `<elements>` is the number of elements in the array (Note: Indices run from 0 to `<elements>`-1)

Creating Array Datatypes and Variables

We can explicitly create new data types for arrays, or we can just define variables as arrays of an exiting data type

Defining a new array data type:

```
typedef char TokenSpelling[80];

TokenSpelling literal;
```

- Th first line defined a new type TokenSpelling, able to hole 80 characters (indexed 0–79)
- The second line creates a variable `literal` of that type

More frequently used (in C): Direct variable definitions

```
char literal[80];
```

Arrays and array types have some special properties in C, they cannot always be handled in the same way as other datatypes (more later)

No Safety Belts and No Air Bag!

C does not check if the index is in the valid range!

- In fact, it does not even explicitly store the size of the array anywhere in the generated code
- If you access `literal[121]` you might change some critical other data
- The operating system may catch some of these wrong accesses, but do not rely on it!)

This is source of many of the buffer-overflow errors exploited by crackers and viruses to hack into systems!

Arrays in Other Languages

Pascal Arrays:

- Arrays can be indexed by integer values, characters, and enumeration types
- However, any range of indices is possible (e.g. `('a'..'m')` or `(-10..2003)`)
- The range and hence the aray size is fixed an compile time
- The compiler inserts checks for the validity of any access, catching possible bugs (no buffer overflow errors possible (with arrays))!

Modern Scripting languages:

- Arrays are often associative: Any value can be used as an index
- Arrays grow dynamically as new elements are added
- Access of an index not in the array may produce an error message or return a special value (normally the empty string)

The Source Handler Interface

A source handler has to deliver the program file as a sequence of characters

As a bonus, it maintains line and column for error messages

Interface to our Source Handler:

void InitSourceHandler(char* filename): Initializes the source handle to read input from the named file

void ExitSourceHandler(): Cleans up after we are done with a file, releases used resources

CurrentChar(): Return the current character (initially the first character of the file)

- We follow the C convention of using `int` to represent all possible characters and the `EOF` value

int NextChar(): Read the next character from the input file and return it

int CurrentLine(): Return the line number of the current line

int CurrentColumn(): Return the column, i.e. the sequence number of the current character in the current line

Exercises

Get the source handler from the web, compile and run it, and read the code. It is available from the CSC519 home page, or directly as `http://www.cs.miami.edu/~schulz/CSC519/PL_STEP1.tgz`. Make sure that you understand at least how to use the interface (i.e. the functions declared in `source_handler.h`

CSC519
Programming Languages

Scanner Datatypes and Design

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

The Token Representation

Remember: We want to represent (PL) tokens as a tuple
`(<type>, <spelling>, <line>, <column>)`

`<type>` corresponds to the non-terminal grammar symbol for the token, i.e. it is one from a finite set of posibilities

- The proper type to use for the token type is an enumeration type
- In C, the syntax for defining an enumertation type is `typedef enum {e1,e2,e3}EType;`

`<spelling>` is the sequence of characters generated from the token non-terminal

- An adequate representation of the spelling in C is an array of characters
- C convention is to terminate any string with the character `'\0'` (the NUL character)

Line and column information are numbers, adequately represented as C integer numbers (`int`)

- Integers are one example of a basic data type in C

Basic Data Types

Basic data types have full language support in imperative languages

- They can be compared, assigned, and used in (suitable) expressions
- A wide range of predefined operators is available for those types

Typically, basic data types have a natural machine representation

- Characters are typically represented by a single byte
- Integer numbers are represented as a machine word (and supported by machine operations for arithmetic)
- Floating point numbers are represented by 1, 2, or 4 machine words, and may or may not have direct support from the processor

Depending on the language, the data types may be strictly defined by the language, or loosely defined to take advantage of different hardware features

Basic Data Types in C

C has basic data type support for integers and floating point numbers

Integer data types are

- `char` (always a single byte (but not necessarily 8 bit))
- `short` (often two bytes)
- `int` (the "natural" integer size, normally a single machine word)
- `long` ("large" integers, often the same as `int` on modern architectures)
- `long long` (Only in the latest standard, the largest integer data type, normally 64 bits on modern machines)
- All types come in signed and unsigned varieties, default is signed (except for char, where the default is machine-dependent)
- The relationship between different integer data types is only loosely specified:
  * `char` has to represent at least 256 different values
  * `short` has to be at least 16 bits
  * `int` has to be at least 16 bits
  * `long` has to have at least 32 bits
  * `short` in not bigger than `int`, which is not bigger than `long`

Basic Data Types in C (Contd.)

Floating point numbers

- `float` numbers are single precision floating point numbers, typically stored in 32 bits
- `double` number are enhanced precision numbers, used by all mathematical library functions

Enumerations and Pointers

- Enumeration data types in C are integer data types, and can be freely mixed with integer types, making them effectively into basic data types
- C allows the use of direct memory addresses (pointers) of various types, and has enough support to treat them as basic data types

Combining Different Elements: Records/Structs

Arrays have elements of one type, indexed by a key (typically an integer in compiled imperative languages)

A record or struct combines diverse elements into one type

- Different elements are referenced by name
- Memory layout typically is sequential, with each element having a fixed offset from the address of the struct in memory

C syntax:

```
struct <name> {<type1> <var1>; <type2> <var2> ;...}
```

Typically, a struct is used to define a new type:

```
typedef struct <name>
   {<type1> <var1>; <type2> <var2> ;...}<newtype>;
```

Structure elements are accessed using the dot-notation:

- If `x` is a variable of a struct type, `x.element` accesses the `element` in `x`

Example: The PL Token Type in C

A possible implementation of the tuple structure:

```
typedef struct token
{
   TokenType type;
   char      spelling[80];
   int       line;
   int       column;
}Token;

Token current; /* Define a variable of that type */
```

Note: We are reusing the `TokenType` enumeration type defined previously

We can e.g. use `current.type = Variable;` to access one of the elements

PL Grammar: Tokens

A reasonable PL grammar can be build using the following tokens:

- `<null> ::= 0`
- `<semi> ::= ';'`
- `<colon> ::= ':'`
- `<inc> ::= '++'`
- `<assign> ::= '<='`
- `<loop> ::= 'loop'`
- `<endloop> ::= 'endloop'`
- `<goto> ::= 'goto'`
- `<variable>`, starting with `X`, `Y`, `Z`, followed by a natural number in decimal representation
- `<label>`, a `l` followed by a natural number

Notice: We can almost always recognize the token type by just looking at the first character (for "loop" and labels, we'll have to look at the second one)

- If we use a good invariant, we need no lookahead and can still build a very simple scanner, using only one multi-token state

C Enumeration of PL Tokens

```
typedef enum
{
   NoToken,
   NullDigit,
   Semicolon,
   Colon,
   IncOp,
   AssignOp,
   ResWordLoop,
   ResWordEndloop,
   ResWordGoto,
   Variable,
   Label
}TokenType;
```

PL Grammar: Additional Lexical Rules

To fully specify the lexical part of the grammar, we have to specify the decimal representation of natural numbers:

- `<null> ::= 0`
- `<vdigit> ::= 1|2|3|4|5|6|7|8|9`
- `<digit> ::= <null> | <vdigit>`
- `<digit-sequence> ::= <empty> | <digit> <digit-sequence>`
- `<pos-int> ::= <vdigit> <digit-sequence>`

Tokenizer Architecture

Core of the tokenizer is a function `Token NextToken()` that returns the next token

Invariant: Whenever the function is called, the current character in the source handler is not yet part of any token

The function can be structured as follows:

- Initialize local `Token` variable as `(NoToken, "", 0,0)`
- Skip white space (using e.g. the C function `isspace()` from `<ctype.h>`)
- Record line and column (from source handler)
- Switch on the first character, determining token type (error, if no token possible)
- Read remaining characters into spelling (error, if incomplete token)
- Return complete token

Assignment

Write a function `Token NextToken()` for PL tokens (based on the supplied source handler)

Write a function `void PrintToken(Token tok)` that prints a argument of type `Token` as a 4-tupel as suggested

Write a driver program that uses these functions to print a file "`pl_test`" as a sequence of tokens

CSC519
Programming Languages

Pointers and Dynamic Data Structures

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Pointers

Pointers are derived types of a base type

- A pointer is the memory address of an object of the base type
- Given a pointer, we can manipulate the object pointed to

Typical operations:

- Create a new object in memory (note: the value of the object is undefined!) and return a pointer to it
- Destroy an object created in that way, returning the used memory
- Compare pointers for equality
- Assign pointers to variables of the correct type
- Dereference a pointer (giving us the object pointed to)

Note that none of these operations violate type safety

- Pointers in strongly typed languages have a specific type and keep it!

Why Pointers?

The are two main reasons for using pointers:

- Efficiency
- Dynamically growing data structures

Efficiency Aspects

- Pointers are typically represented by one machine word
- Storing pointers instead of copies of large objects safes memories
- Passing pointers instead of large objects is much more efficient

Dynamically growing data structures

- In most imperative languages, each data type has a fixed size and memory layout
- Pointers allow us to build data structures by adding and removing fixed size cells

Basic Pointer Operations in C

C uses the unary `*` operator for both pointer definition and pointer dereferencing, and `&` for getting the adress of an existing object

- `int var;int *p;` defines `var` to be a variable of type `int` and `p` to be a variable of type pointer to int
- `p = &var` makes `p` point to `var` (i.e. `p` now stores the address of `var`)
- `*p = 17;` assigns 17 to the `int` object that `p` points to (in our example, it would set `var` to 17)
- Note that `&(*p) == p` always is true for a pointer variable pointing to a valid object, as is `*(&var)==var` for an arbitrary variable!

Memory Allocation in C

The C language offers no dynamic creation of new objects. Instead, it offers a library function `malloc()` (declared in `<stdlib.h>`)

- It's declared as `void *malloc(size_t size);`
- `void*` is the type of a generic pointer, i.e. a pointer that can point to any object
- `size_t` is a new data type from the standard library. It's guaranteed to be an unsigned integer data type
- `malloc()` allocates a region big enough to hold the requested number of bytes on the heap (a reserved memory region) and returns the address of the first byte (a pointer to that region)
- The `sizeof` operator returns the size of an object or datatype
  - `p = malloc(sizeof(int));` allocates a memory region big enough to store an integer and makes `p` point to it
  - The `void*` pointer is silently converted to a pointer to `int`
- If no memory is available on the heap, `malloc()` will return the `NULL` pointer (also written as plain `0`)

Freeing Allocated Memory

The counterpart to `malloc()` is `free()`

- It is declared in `<stdlib.h>` as
  `void free(void* ptr);`
- `free()` takes a pointer allocated with `malloc()` and returns the memory to the heap

Note that it is a bug to call `free()` with a pointer not obtained by calling `malloc()`

It also is a bug to call `free()` with the same pointer more than once

Pointers are a Mixed Blessing!

Dangling pointers

- A dangling pointer is a pointer not pointing to a valid object
- A call to `free()` leaves the pointer dangling (the pointer variable still holds the adress of a block of memory, but we are no longer allowed to use it)
- Copying a pointer may also lead to additional dangling pointer if we call `free()` on one of the copies
- Trying to access a dangling pointer typcially causes hard to find errors, including crashes

Memory leaks

- A memory leak is a situation where we lose the reference to an allocated piece of memory:

```
p = malloc(100000 * sizeof(int));
p = NULL; /* We just lost a huge gob of memory! */
```

- Memory leaks can cause programs to eventually run out of memory
- Periodically occurring leaks are catastophic for server programs!

Example: Linear Lists of Tokens

Assume we want to store a list of PL tokens, but we cannot predict the list lenght

- One suitable data structure is a linear list
- A list is recursively defined either as the empty list (represented by the `NULL` pointer), or as an element, followed by a list
- Any element holds some local data and a pointer to the rest of the list

We modify the definition of the `Token` record as follows:

```
typedef struct token
{
   struct token *next; /* Points to rest of list */
   TokenType type;
   char      spelling[80];
   int       line;
   int       column;
}Token;
```

Example (Contd.)

```
#include <ctype.h>
#include "scanner.h"
int main(void)
{
   Token tok;
   Token *anchor = NULL, **next,*handle;

   InitSourceHandler("pl_test");
   next = &anchor;
   while((tok = NextToken()).type!=NoToken)
   {
      *next = malloc(sizeof(Token)); /* Check for NULL omitted! */
      **next = tok;
      next = &((**next).next);
   }
   for(handle = anchor; handle; (*handle).next)
   {
      PrintToken(handle); /* Redefined to accept a pointer! */
   }
   ExitSourceHandler();return EXIT_SUCCESS;
}
```

Remarks on the Example

To make the code fit onto a single slides, there are two omissions:

- No check if `malloc()` suceeded
- Allocated memory is never freed

Good programming practice always checks if `malloc()` succeeded (i.e. returns not `NULL`)

- In multi-tasking systems, even small allocations may fail, because other processes consume resources
- The OS may limit memory usage to small values
- Failing to implement that chack can lead to erratic and non-reproducable failure!

Less critical: We do not return the allocated memory

- In modern operating systems, process resources are automatically reclaimed by the OS if the process terminates
- Manual freeing does, however, allow us to check for memory leaks more easily

Pointers and Arrays in C

In C, arrays and pointers are strongly related:

- Everwhere except in a definition, arrays are equivalent to a pointer to its first element
- In particular, arrays are passed to functions by passing their address!
- More exactly: An array degenerates to a pointer if passed or used in pointer contexts

Not only can we treat arrays as pointers, we can also apply pointer operations to arrays:

- If `p` is a pointer to the first element of an array, we can use `p[3]` to asses the third element of that array
- In general, if `p` points to some memory address `a`, `p[i]` points to `a+sizeof`(base type of pointer)`*i`

Example

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char a[] = "CSC519\n";
   char *b;
   int i;

   b=a;

   printf(b);
   for(i=0;b[i];i++)
   {
      printf("Character %d: %c\n", i, b[i]);
   }
   return EXIT_SUCCESS;
}
```

Example Output

Compiling: **gcc -o csc519 csc519.c**

Running:

```
CSC519
Character 0: C
Character 1: S
Character 2: C
Character 3: 5
Character 4: 1
Character 5: 9
Character 6:
```

Additional Pointer Properties in C

Pointers in C are no strictly type safe!

- The generic pointer type `void*` can be converted to any other pointer type (and any other pointer type can be converted to `void*`)
- We can use explicit casts to coerce one pointer to a different type (although this is often a bad idea)

Pointers of the same type can be compared using `<`, `>`, `<=`, `>=`

- The result is only defined, when both pointers point at elements in the same array or struct (but note that that normally includes pointers into a chunk of memory allocated with `malloc()`)!
- Pointers to elements with a smaller index are smaller than pointers to elements with a larger index

Pointer arithmetic allows addition of integers to (non-void) pointers

- If `p` points to element `n` in an array, `p+k` points to element `n+k`
- As a special case, `p[n]` and `*(p+n)` can again be used interchangably (and often are in practice)
- Most frequent case: Use `p++` to advance a pointer to the next element in an array

Example rewritten

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char a[] = "CSC519\n";
   char *b;
   int i = 0;


   printf(a);
   for(b=a;*b; b++)
   {
      printf("Character %d: %c\n", i, *b);
      i++;
   }
   return EXIT_SUCCESS;
}
```

Exercises

Think about implementing a string library that can handle strings of changing sizes. Functionality:

- Create an empty string
- Destroy a string
- Initialize a string to a C type string
- Append a C type string (pointer to a `'\0'`-terminated array of characters) to a dynamic string
- Append two C typ strings
- Convert a dynamic string to a C type string

Hint: The user of the string only ever gets to handle a pointer to a structure!

- That structure contains a pointer to the `malloc()`ed string and the size of the allocated area
- If the string grows beyond the allocated length, a bigger region is allocated, and the old string is copied over

CSC519
Programming Languages

Pointers and Dynamic Data Structures II
Type System Revisited

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Pointers and Arrays in C

In C, arrays and pointers are strongly related:

- Everwhere except in a definition, an array is equivalent to a pointer to its first element
- In particular, arrays are passed to functions by passing their address!
- More exactly: An array degenerates to a pointer if passed or used in pointer contexts

Not only can we treat arrays as pointers, we can also apply array operations to pointers:

- If `p` is a pointer to the first element of an array, we can use `p[3]` to access the third element of that array
- In general, if `p` points to some memory address `a`, `p[i]` points to `a+sizeof`(base type of pointer)`*i`

Example

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char a[] = "CSC519\n";
   char *b;
   int i;

   b=a;

   printf(b);
   for(i=0;b[i];i++)
   {
      printf("Character %d: %c\n", i, b[i]);
   }
   return EXIT_SUCCESS;
}
```

Example Output

Compiling: **gcc -o csc519 csc519.c**

Running:

```
CSC519
Character 0: C
Character 1: S
Character 2: C
Character 3: 5
Character 4: 1
Character 5: 9
Character 6:
```

Additional Pointer Properties in C

Pointers in C are no strictly type safe!

- The generic pointer type `void*` can be converted to any other pointer type (and any other pointer type can be converted to `void*`)
- We can use explicit casts to coerce one pointer to a different type (although this is often a bad idea)

Pointers of the same type can be compared using `<`, `>`, `<=`, `>=`

- The result is only defined, when both pointers point at elements in the same array or struct (but note that that normally includes pointers into a chunk of memory allocated with `malloc()`)!
- Pointers to elements with a smaller index are smaller than pointers to elements with a larger index

Pointer arithmetic allows addition of integers to (non-void) pointers

- If `p` points to element `n` in an array, `p+k` points to element `n+k`
- As a special case, `p[n]` and `*(p+n)` can again be used interchangably (and often are in practice)
- Most frequent case: Use `p++` to advance a pointer to the next element in an array

Example rewritten

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char a[] = "CSC519\n";
   char *b;
   int i = 0;


   printf(a);
   for(b=a;*b; b++)
   {
      printf("Character %d: %c\n", i, *b);
      i++;
   }
   return EXIT_SUCCESS;
}
```

Type System Revisited

The language offers a set of basic types built into the language

We can define new, quasi-basic types as enumerations

We can construct new types using type contruction:

- Arrays over a base type
- Structures combining base types
- Unions (able to store alternative types, more later)
- Pointer to a base type

This generates a recursive type hierarchy!

Basic Types

Basic types in C:

- `char` (typically used to represent characters)
- `short`
- `int`
- `long`
- `long long`
- `float`
- `double`

All integer types come in signed (default except for `char`) and unsigned variety

- If `char` is signed or unsigned is an implementation issue!

Defining New Types

In principle, we can state all derived types explicitely in all declarations or definitions

- `int *ip; /* Pointer to int */`
- `struct {int stack[80];int sp;} *complex[20];`
  `/* Array of 20 pointers to a structure containing an array of 80 ints and a single int */`

However, this becomes tedious soon

- Types need to be retyped in exactly the same way
- Definitions become hard to understand (especially since C's syntax for types is non-trivial)

Solution: Defining names for new types

- Allows the same kind of abstraction in the type system as functions offer for algorithms
- We can build types on types, hiding implementation
- We can introduce aliases for names

typedef

The typedef keyword is used to define names for types in C

General syntax: If we add typedef to a variable definition, it turns into a type definition

Examples:

```
int size; /* Define int variable  */
typedef int size_t; /* Define a new type size, equal to int */

struct{int stack[80];int sp;} stack; /* Stack variable */
typedef struct{int stack[80];int sp;} stack_t; /* Define a stack datatype */
```

Note: typedef is often used to hide implementation-specific issues

Unions

Unions of base types allow the new type to store one value of any of its base types (but only one at a time)

The syntax is analogous to that of structures:

- `union {int i; float f; char *str;} numval`
- `numval` can store either an integer or a floating point number, or a pointer to a character (normally a string)
- Access is as for structures: `numval.i` is the integer value

Note: Unions weaken the type system:

- `numval.f=1.0; printf("%d\n",numval.i);`
- Situations like that are, in general, impossible to detect at compile time

Notes:

- In some other languages, unions are know as variants
- Variant records sometimes combine structures and unions in one data structure

Union Usage Example

Multi-Type data structures

```
typedef union int_or_p
{
   long i_val;
   void *p_val;
}IntOrP;

typedef int PStackPointer;

typedef struct pstackcell
{
   int           size;    /* ...of allocated memory */
   PStackPointer current; /* First unused address, 0 for empty stack */
   IntOrP        *stack;  /* Stack area */
}PStackCell, *PStack_p;
```

Pointers and Structures/Unions

Most interesting data strucures use pointers to structures

– Examples: Linear lists (as seen), binary trees, terms,. . .

Most frequent operation: Given a pointer, access one of the elements of the structure (or union) pointed to

– `(*stack).current = 0;`
– Note that that requires parentheses in C

More intuitive alternative:

– The `->` operator combines dereferencing and selection
– `stack->current = 0;`
– This is the preferred form (and seen nearly exclusively in many programs)

Exercises

Define datatypes for e.g. binary search trees over integers

Implement the basic algorithms on your data types:

- Inserting an element
- Finding an element (easy)
- Deleting an element (hardest)

CSC519
Programming Languages

Subroutines, Procedures and Functions

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Simple Subroutines

Let's pretend we know nothing about functions!

A basic subroutine is a sequence of statments that

- Can be called from anywhere in the program
- Executes normally
- After termination, returns control to the statement after the call

Simple subroutines only affect the computation by changing the global state of the computation

In C, this corresponds to a function of type void, and without any arguments

- BASIC offers a gosub statement
- Most modern languages offer more powerful procedures or functions

Global Variables

Variables declared outside any local scope (i.e. outside any function in C) are called global variables

– They are visible and accessible from the point of definition (or declaration) to the end of the source file

Global variables have unlimited lifetime

– They are initialized at compile time
– Their values never are lost (unless we explicitely assign new values)

Example: Use of Simple Subroutines

```
#include <stdio.h>
#include <stdlib.h>

#define MAXINPUT 64

char char_array[MAXINPUT];
int  array_pos = 0;
int  c, i, insert;
int  first, next;

void read_array(void);
void print_array(void);
void remove_doubles(void);
```

Example (Continued)

```
int main(int argc, char* argv[])
{
   read_array();
   print_array();
   remove_doubles();
   print_array();

   return EXIT_SUCCESS;
}
```

Example (Continued)

```
void read_array(void)
{
   while((c=getchar())!=EOF)
   {
      if(array_pos==MAXINPUT)
      {
         printf("Input array full!\n");
         break;
      }
      char_array[array_pos]=c;
      array_pos++;
   }
}
```

Example (Continued)

```
void print_array(void)
{
   printf("Array: ");
   for(i=0; i< array_pos; i++)
   {
      putchar(char_array[i]);
   }
   printf(" :End\n");
}
```

Example (Continued)

```
void remove_doubles(void)
{
   i = 0;
   insert = 0;

   for(first = char_array[i]; i<array_pos; first = next)
   {
      char_array[insert] = first;
      insert++;

      do
      {
         i++;
         next = char_array[i];
      }while(next==first && i< array_pos);
   }
   array_pos = insert;
}
```

Problem: Variable Conflicts

In general, the use of global variables should be kept to a minimum (Geoff Sutcliffe: "Global variables give you warts!")

– It's hard to keep track of where they are used
– Manipulations of global variables by subroutines are not obvious

Example:

```
int main(int argc, char* argv[])
{
   read_array();
   for(i=0; i<5; i++)
   {
      print_array();
   }
   return EXIT_SUCCESS;
}
```

What does this print?

Local Variables

There are two kinds of definitions (and declarations) in C

- Definitions outside any block are global definitions
- Definitions written inside a block are called local definitions

The scope of a local definition begins at the definition and ends at the end of the innermost enclosing block

- A local definition always creates a new variable
- It may hide an existing varible of the same name
- The lifetime of a variable normally is limited - the varible is destroyed at the end of the enclosing block
- Accessing a local variable does not change the value of any global, permanent variable

It's always preferable to restrict variables to the narrowest scope possible!

- May increase code size, but reduces complexity!

Example Revisited

```
#include <stdio.h>
#include <stdlib.h>

#define MAXINPUT 64

char char_array[MAXINPUT];
int  array_pos = 0;

void read_array(void);
void print_array(void);
void remove_doubles(void);
```

Example Revisited

```
int main(int argc, char* argv[])
{
   int i;

   read_array();
   for(i=0; i<5; i++)
   {
      print_array();
   }
   remove_doubles();
   print_array();

   return EXIT_SUCCESS;
}
```

Example Revisited

```
void read_array(void)
{
   int c;
   while((c=getchar())!=EOF)
   {
      if(array_pos==MAXINPUT)
      {
         printf("Input array full!\n");
         break;
      }
      char_array[array_pos]=c;
      array_pos++;
   }
}
```

Example Revisited

```
void print_array(void)
{
   int i;

   printf("Array: ");
   for(i=0; i< array_pos; i++)
   {
      putchar(char_array[i]);
   }
   printf(" :End\n");
}
```

Example Revisited

```
void remove_doubles(void)
{
   int i = 0, insert = 0;
   char first, next;

   for(first = char_array[i]; i<array_pos; first = next)
   {
      char_array[insert] = first;
      insert++;

      do
      {
         i++;
         next = char_array[i];
      }while(next==first && i< array_pos);
   }
   array_pos = insert;
}
```

Calling Conventions

Often, we want to be able to use subroutines to perform the same tasks for different variables

The simplest way of doing so is by using dedicated global variables for communication:

```
int factorial_arg1, factorial_result;
void factorial(void)
{
   int i;

    factorial_result = 1;
    for(i=2; i<= factorial_arg1; i++)
    {
       factorial_result = factorial_result *i;
    }
}
```

We call a convention of this type a calling convention

– Calling conventions allow us to more flexibly reuse code

Example: Factorial Table

```
int main(void)
{
   int i,j;

   for(i=0; i<5; i++)
   {
      factorial_arg1 = i;
      factorial();

      printf("Factorial(%d)=%d\n", i, factorial_result);

      for(j=0; j<5; j++)
      {
         factorial_arg1 = i+j;
         factorial();
         printf("Factorial(%d+%d)=%d\n", i,j, factorial_result);
      }
   }
}
```

Problem: Recursion

Consider a simple recursive implementation of factorial:

```
void factorial()
{
   if(factorial_arg <= 1)
   {
      factorial_result = 1;
   }
   else
   { /* Compute (factorial_arg - 1) */
      factorial_arg = factorial_arg - 1;
      factorial();
      factorial_result = factorial_result * factorial_arg; /* Ooops! */
   }
}
```

We need a better way!

Generalized Calling Convention

To facilitate recursion and multiple functions interacting in arbitrary way, we need a more general way of passing parameters

Idea: Use a stack

- Each caller pushes arguments on the stack
- Callee can use those stack elements as local variables
- Either callee or caller clears arguments of the stack (we let the caller clean up - it's easier)
- Different functions can share the same stack!

Return values can be passed via the stack or directly via dedicated variables

Example: Stack Based Calling Discipline

Global Definitions and Includes:

```
#include <stdio.h>
#include <stdlib.h>

#define STACKSIZE 1024

int stack[STACKSIZE];
int sp = 0; /* Points to first empty position */
int result;
```

Example: Recursive Factorial

Note that sp points to next unused stack position – our argument is stored at sp-1!

```
void factorial()
{
   if(stack[sp-1] <= 1)
   {
      result = 1;
   }
   else
   {
      stack[sp] = stack[sp-1] - 1; /* Argument for recursive call */
      sp++;
      factorial();
      sp--; /* Clean up stack */
      result = result * stack[sp-1];
   }
}
```

Example: Main Program

```
int main(void)
{
   int i,j;

   for(i=0; i<5; i++)
   {
      stack[sp] = i;
      sp++;
      factorial();
      sp--;
      printf("Factorial(%d)=%d\n", i, result);
      for(j=0; j<5; j++)
      {
         stack[sp] = i+j;
         sp++;
         factorial();
         sp--;
         printf("Factorial(%d+%d)=%d\n", i,j, result);
      }
   }
}
```

Exercises

Check the PL grammar from assignment 2 (it's on the web)

Consider how to transform PL programs into an abstract syntax notation (using either trees or prefix notation)

CSC519
Programming Languages

Procedures – Parameter Passing Methods

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Functions and Procedures

A function is a named subroutine

- It is called via its name
- It can accept (any number of) parameters
- It returns a value of a certain type and can be used in an expression
- Hence it can be seen as a unary operator extending the language

A procedure is a named subroutine

- It is called via its name
- It can accept (any number of) parameters
- It does not return a value, and can be used as a statement, extending the set of operators of a language

Modern languages (and C) do not distinguish between the two

- A procedure is simply a function returning type void (i.e. nothing)
- Since expressions can be made into statements in C, any function can be used as a procedure

More Terminology

A function definition consists of two parts:

- A header, defining the calling interface of the function
- A body, describing the actual computation going on

The header has three parts:

- The return type (what type of value does the function return)
- The name (how do we call the function)
- The formal parameters (what values go into the function, often just called formals)
  - Formal parameters have a name and a type
  - They (typically) can be used as variables in the function

Example

```
double square(double x)
{
    return x*x;
}
```

The header consists of

- The return type
- The name
- The formals

The body contains only a single statement

Binding and Activation

A use of a function is called a call of the function

- A call consists of the name of the function and a list of actual parameters (or actuals)
- A function call results in two actions:
  * First, the formals are bound to the actuals
  * Then, the function body is activated (or executed)

We distinguish different types of function calls, based on the type of binding:

- Call by value
- Call by reference
- Call by value-result
- Call by name

Call by Value

Call by value is characterized as follows:

- Formal parameters are treated as local variables
- Actual parameters are evaluated before the body is activated
- The formal parameters are initialized to the values of the actuals

Advantages

- Easy to implement
- Clear semantics/no side effects

Disadvanages

- Values have to be copied to new locations (may be expensive)
- You cannot propagate values out of the function that way

Remarks

- Call by value is used by C and it's the default passing method in Pascal
- All functional languages normally implement call by value

Example

```
int i;

void swap(int x, int y)
{
   int z;

   z=x;
   x=17;
   i=i+1;
   y=z;
}
int main(void)
{
   int arr[3] = {2, 4, 6, 8};

   i=1;
   printf("i=%d, arr={%d,%d,%d,%d}\n", i, arr[0],arr[1],arr[2],arr[3]);
   swap(arr[2],arr[i]);
   printf("i=%d, arr={%d,%d,%d,%d}\n", i, arr[0],arr[1],arr[2],arr[3]);
}
```

Swap Example under Call by Value

Under call by value, all assignments to local variables inside `swap()` have no effect outside the function

– The only change is to the global variable i

Hence, the program prints

```
i=1, arr={2,4,6,8}
i=2, arr={2,4,6,8}
```

Note: This is how the program would actually be executed in C

Call by Reference

Call by reference is characterized as follows:

- The formal parameters are treated as references or aliases for the actual parameters. The reference is computed only once, at activation time
- Any change to a formal parameter directly and immediately changes the actual parameter
- Note: For this, all actual parameters have to be lvalues (i.e. denote changable storage locations). For the other case, there are the following options
  - Non-lvalues are evaluated and stored in a temporary variable (which is discarded after the call)
  - The compiler rejects non-lvalues as actuals
  - The program just treats the locations of the values as variables (creating bugs and potentially thrashing the programs - not recommended)
- Advantages:
  - Call by reference requires no copying, just passing of a pointer
  - Changes can be propagated out of the function
- Disadvantages:
  - Functions can have non-obvious side effects

Example under Call by Reference

Under call by reference, the following bindings happen:

- `x` becomes a synonym for `arr[2]`
- `y` becomes a synonym for `arr[1]` (because `i` has the value 1 at activation time)

The assignments thus have the following effect:

- `z=x;` changes the local variable `z` to the value of `arr[2]`, which is 6
- `x=17;` sets the local variable `x` and hence `arr[2]` to 17
- `i=i+1;` only affects the global variable `i`
- `x=y;` sets `x` and hence `arr[2]` to the value of `y` (synonymous with `arr[1]`), and hence to 4
- `y=z;` finally sets `y` (aka `arr[1]`) to 6, the value of `z`

Ergo the program prints

```
i=1, arr={2,4,6,8}
i=2, arr={2,6,4,8}
```

- The two elements have been swapped. . . all is fine!

...or is it???

Consider the alternative call in main:

```
int main(void)
{
   int arr[3] = {2, 4, 6, 8};

   i=1;
   printf("i=%d, arr={%d,%d,%d,%d}\n", i, arr[0],arr[1],arr[2],arr[3]);
   swap(arr[1],arr[1]);
   printf("i=%d, arr={%d,%d,%d,%d}\n", i, arr[0],arr[1],arr[2],arr[3]);
}
```

...or is it???

Consider the alternative call in main:

```
int main(void)
{
   int arr[3] = {2, 4, 6, 8};

   i=1;
   printf("i=%d, arr={%d,%d,%d,%d}\n", i, arr[0],arr[1],arr[2],arr[3]);
   swap(arr[1],arr[1]);
   printf("i=%d, arr={%d,%d,%d,%d}\n", i, arr[0],arr[1],arr[2],arr[3]);
}
```

- Now `x` and `y` point to the same memory location (they are aliases)!
  - An assignment to one local variable also changes the other one!
- The program prints

  ```
  i=1, arr={2,4,6,8}
  i=2, arr={2,17,4,8}
  ```
- The value of `arr[1]` has been lost

Call by Value-Result

Call by value-result (or copy-in/copy-out) tries to provide call-by-reference semantics without the aliasing problem

- Formals always act like independent local variables
- They are initialized to the values of the actuals (just like call by value) (copy-in phase)
- After the activation ends, the final values of the formals are copied back to the actuals

Advantages:

- Allows for return values
- Function does not need to handle aliases

Disadvantage:

- Combines disadvantages of call by reference (side effects) and call by value (expensive)
- If global variables are passed as arguments, changes to them will be lost in the copy-out phase

Example under Call by Value-Result

Call by value-result proceeds just like call by value (Note: `green` refers to the alternative call):

- `x` is initialized to 6 (`4`)
- `y` is initialized to 4 (`4`)
- `z` is set to 6 (`4`)
- `x` (a local variable!) is set to 17
- `i` is increased
- `x` is set to 4 (`4`)
- `y` is set to 6 (`4`)
- Finally, the copy-out phase makes `arr[2]` equal to 4 and `arr[1]` equal to 6
- Finally, the copy-out phase makes `arr[1]` equal to 4 and `arr[1]` equal to 4

Ergo the program prints

```
i=1, arr={2,4,6,8}
i=2, arr={2,6,4,8}
```

or

```
i=1, arr={2,4,6,8}
i=2, arr={2,4,6,8}
```

Call by Name

Call by name conceptually passes the text of the actual

- Every access reevaluates that text (typically in the context of the function system call)
- It's now mostly of interest for historical reasons (but see below)

Original example under call by name:

- `x` is replaced by `arr[1]`
- `y` is replaced by `arr[i]`
- Since `i` changes in `swap()`, `y` refere to different locations before and after the `i=i+1;` statement!

Output:

```
i=1, arr={2,4,6,8}
i=2, arr={2,6,4,8}
```

- (We got lucky!)

Calling Methods in C

Functions are always Call by Value in C

– However, since arrays degenerate to a pointer to the first elements, teh effect is as if arrays were passed by reference!

C can emulate call by reference using pointers:

```
void swap(int *x, int *y)
{
   int z;

   z  =*x;
   *x =*y;
   *y = z;
}
...
swap(&(arr[1), &(arr[i]));
```

C++ adds real call by reference (one of the three most bogus decisions in C++ design)

Functions and Macros in C

C actually has two procedure-like constructs:

- Functions are typically compiled to simple subroutines that are actually called at compile time
- Preprocessor macros are expanded at the source code level

Simple C macros have no formal arguments

- A simple macro definition has the form
  `#define <name> <replacement>`
- `<name>` can be any valid C identifier
- `<replacement>` can be any text, up to the end of line (but lines can be extended by adding a \ as the last character of the line)

Exercises

Find an example where call by name does not do a correct swap (using our contrieved `swap()`)

Are there any more things about call by name that you have to keep in mind?

Write a working `swap()` using C macros

CSC519
Programming Languages

Scoping Issues

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

The C Preprocessor

The C preprocessor performs a textual rewriting of the program text before it is ever seen by the compiler proper

- It includes the contents of other files
- It expands macro definitions
- It conditionally processes or removes segments of the program text

After preprocessing, the program text is free of all preprocessor directives

Simple C macros have no formal arguments

- A simple macro definition has the form
  `#define <name> <replacement>`
- `<name>` can be any valid C identifier
- `<replacement>` can be any text, up to the end of line (but lines can be extended by adding a \ as the last character of the line)

Normally, **gcc** will transparently run the preprocessor. Run **gcc -E** `<file>` if you want to see the preprocessor output

C Macros

Macro definitions can also contain formal arguments

`#define <name>(arg1,...,arg1) <replacement text>`

If a macro with arguments is expanded, any occurence of a formal argument in the replacement text is replaced with the actual value of the arguments in the macro call

This allows a more efficient way of implementing small "functions"

- But: Macros cannot do recursion
- Therefore macros are usually only used for very simple tasks

By convention, preprocessor defined constants and many macros are written in `ALL_CAPS` (using underscores for structure)

Macros have call by name semantics!

- Textual replacement results in call by name
- Macros also can capture local variables (dynamic scoping)

#define Examples

macrotest.c:

```
#define XOR(x,y) ((!(x)&&(y))||((x)&&!(y))) /* Exclusive or */
#define EQUIV(x,y) (!XOR(x,y))

void test_macro(void)
{
   printf("XOR(1,1)  : %d\n", XOR(1,0));
   printf("EQUIV(1,0): %d\n", EQUIV(1,0));
}
```

$ gcc -E macro_test.c

```
# 4 "macrotest.c"
void test_macro(void)
{
   printf("XOR(1,1)  : %d\n", ((!(1)&&(0))||((1)&&!(0))));
   printf("EQUIV(1,0): %d\n", (!((!(1)&&(0))||((1)&&!(0)))));
}
```

Remarks on Macro Use

Since macros work by textual replacement, there are some unexpected effects:

- Consider `#define FUN(x,y) x*y + 2*x`
  - Looks innocent enough, but: `FUN(2+3,4)` expands into `2+3*4+2*2+3` (not `(2+3)*4+2*(2+3)`)
  - To avoid this, always enclose each formal parameter occurence in parentheses (unless you know what you are doing)
- Now consider `FUN(var++,1)`
  - It expands into `x++*1 + 2*x++`
  - Macro arguments may be evaluated more than once!
  - Thus, avoid using macros with expressions that have side effects

Other frequent problems:

- Semicolons at the end of a macro definition (wrong!)
- "Invisible" syntax errors (run **gcc -E** and check the output if you cannot locate an error)

Some Examples: Macro 1

```
#include <stdio.h>
#include <stdlib.h>

int i = 2;

#define test1(argument)\
{\
   printf("Argument: %d  i: %d\n", argument, i);\
}\

int main(void)
{
   int i = 4;

   test1(i);

   return EXIT_SUCCESS;
}
```

Some Examples: Macro 2

```
#include <stdio.h>
#include <stdlib.h>

int i = 2;

#define test1(argument)\
{\
   printf("Argument: %d  i: %d\n", argument, i);\
}\

int main(void)
{
   int j = 4;

   test1(j);

   return EXIT_SUCCESS;
}
```

Some Examples: Function 1

```
#include <stdio.h>
#include <stdlib.h>

int i = 2;

void test1(int argument)
{
   printf("Argument: %d  i: %d\n", argument, i);
}

int main(void)
{
   int i = 4;

   test1(i);

   return EXIT_SUCCESS;
}
```

Some Examples: Function 2

```
#include <stdio.h>
#include <stdlib.h>

int i = 2;

void test1(int argument)
{
   printf("Argument: %d  i: %d\n", argument, i);
}

int main(void)
{
   int j = 4;

   test1(j);

   return EXIT_SUCCESS;
}
```

Results

The macro version 1 prints

```
Argument: 4  i: 4
```

The macro version 2 prints

```
Argument: 4  i: 2
```

The function versions both print

```
Argument: 4  i: 2
```

What happened?

Dynamic vs. Lexical Scoping

Dynamic scoping has been used by many early LISP versions and results from naive macro expansion

- A called function inherits the visible names of the caller
- Hence a called function can operate on local variables of the caller (and the semantics of a program can change if local variables are renamed!)

Lexical (or static) scoping is the standard for most modern programming languages

- The visibility of names is controlled by the static program text only
- The semantics of a program is invariant under consistent renaming of local variables

Lexical scoping allows us to judge the effect of a function without knowing from where it was called!

Lexical Scoping Rules in C

Visibility of names is controlled by declarations

– Note that any definition implicitely acts as a declaration

There are two kinds of declarations in C

– Declarations written inside a block are called local declarations
– Declarations outside any block are global declarations

The scope of a local declaration begins at the declaration and ends at the end of the innermost enclosing block

The scope of a global declaration begins at the declaration and continues to the end of the source file

Note that function definitions are always at top level, simplifying scoping issues significantly

Explicit Declarations

Variables can be declared by adding the `extern` keyword to the syntax of a definition:

- `extern int counter;`
- `extern char filename[MAXPATHLEN];`

Function declarations just consist of the function header, terminated by a semicolon:

- `int isdigit(int c);`
- `int putchar(int c);`
- `bool TermComputeRWSequence(PStack_p stack,Term_p from,Term_p to);`

Scope Example

```
extern int global_count;

int abs_val (double number)
{
   double help = number;

   if(number < 0)
   {
      help = -1 * help;
      global_count++;
   }
}

int main()
{
   printf("%7f\n", abs_val(-1.0));
}

int global_count;
```

Semantics of Function Calls under Dynamic Scoping

We can model a procedure call under dynamic scoping with call by value using copying rules (Note: To keep things simpler, we assume a single exit point at the end of the procedure and no return value):

- When execution reaches the procedure call, a fresh variable (i.e. a variable with a name not used anywhere else in the program) is created for each formal parameter, and initalized to the value of that parameter
- Each occurence of the formal in the body of the procedure is replaced by the new variable
- The thusly modified body replaces the procedure call in the program text, and execution continues
- Remark: We assume local variables to be declared and created as usual

Example

```
#include <stdio.h>
#include <stdlib.h>

char symbol = '*';

void triangle(int width)
{
   int i;

   if(width !=0 )
   {
      for(i=0; i<width; i++)
      {
         putchar(symbol);
      }
      putchar('\n');
      triangle(width - 1);
   }
}
```

Example (Continued)

```
int main(void)
{
   char symbol = '#';

   triangle(2);
}
```

Running the program like that prints

```
**
*
```

- Remember that C scopes lexically!
- What happens for dynamic scoping?

Example: Recursion Level 1

```
int main(void)
{
   char symbol = '#';

   {
      int width_1 = 2; /* New variable for parameter */
      {
         int i;

         if(width_1 !=0 )
         {
            for(i=0; i<width_1; i++)
            {
               putchar(symbol);
            }
            putchar('\n');
            triangle(width_1 - 1); /* Width_1 is 2 here! */
         }
      }
   }
}
```

Example: Recursion Level 2

```
int main(void)
{
   char symbol = '#';

   {
      int width_1 = 2; /* New variable for parameter */
      {
         int i;

         if(width_1 !=0 )
         {
            for(i=0; i<width_1; i++)
            {
               putchar(symbol);
            }
            putchar('\n');
            {
               width_2 = width_1 - 1; /* Width_2 is 1 now! */
               {
                  int i;

                  if(width_2 !=0 )
                  {
                     for(i=0; i<width_2; i++)
                     {
                        putchar(symbol);
                     }
                     putchar('\n');
                     triangle(width_2 - 1); /* Width_2 is 1 now! */
                  }
} } } } } }
```

Example: Recursion Level 3

. . . finally makes the `if` expression false and terminates

Example: Recursion Level 3

. . . finally makes the `if` expression false and terminates

– . . . and I'm safing bits (and patience) by not printing the file)

As can be seen, the local `symbol` of main is in scope

Hence the program would print

```
##
#
```

Semantics of Function Calls under Lexical Scoping

Modelling execution under lexical scoping differs in minor, but significant ways:

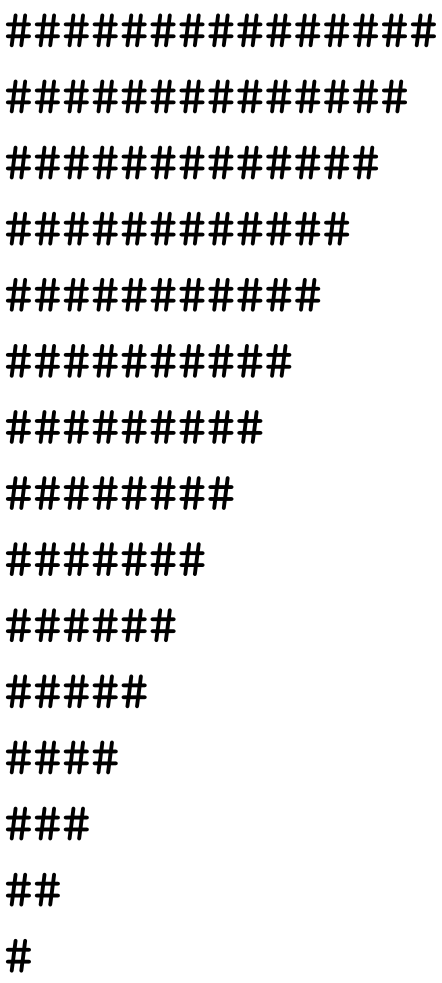
- Rename all local variables (including formal arguments) so that no variable declaration occurs more than once in the program!
- When execution reaches the procedure call, a fresh variable (i.e. a variable with a name not used anywhere else in the program) is created for each formal parameter, and initalized to the value of that parameter
- Each occurence of the formal in the body of the procedure is replaced by the new variable
- The thusly modified body replaces the procedure call in the program text, and execution continues
- Remark: We again assume local variables to be declared and created as usual (otherwise we would need to rename them to)

Exercises

Try the rules for lexical scope calling on the example

Do the dynamic expansion example for an inital call of `triangle(15)` `;-)`

The End



CSC519
Programming Languages

PL-Implementation: Abstract Syntax

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Aim: A PL $\rightarrow$ C Compiler

We already have:

- Source Handler
- Scanner

We need:

- Parser
- Code Generator

Today: Let's talk about the parser!

- Transforms PL programs into abstract syntax trees

Practical Matters: `argc` and `argv`

The C standard defines a standardized way for a program to access its (command line) arguments: `main()` can be defined with two additional arguments

- `int argc` gives the number of arguments (including the program name)
- `char *argv[]` is an array of pointers to character strings each corresponding to a command line argument

Since the name under which the program was called is included among its arguments, `argc` is always at least one

- `argv[0]` is the program name
- `argv[argc-1]` is the last argument
- `argv[argc]` is guranteed to be `NULL`

Example 1: Echoing Arguments

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char* argv[])
{
   int i;

   for(i=1; i<argc; i++)
   {
      printf("%s ", argv[i]);
   }
   putchar('\n');

   return EXIT_SUCCESS;
}
```

Example 2: Using `argc/argv` with the Source Handler

```
#include "source_handler.h"

int main(int argc, char* argv[])
{
   InitSourceHandler(argv[1]);
   while(CurrentChar()!=EOF)
   {
      putchar(CurrentChar());
      NextChar();
   }
   ExitSourceHandler();
   return EXIT_SUCCESS;
}
```

Syntax Rules for PL

```
<program> ::= <empty> |
              <lstatement> <program>

<lstatement> ::= <label> <colon> <statement> |
                 <statement>

<statement> ::= <variable> <inc> <semi>|
                <variable> <assign> <null> <semi> |
                <goto> <label> <semi> |
                <loop> <variable> <semi> <program> <endloop> <semi>
```

Practical Matters: Parser Organization

Typically, the current token is stored in a global variable of type `Token`

- Initially, we call `NextToken()` to read the first token into the variable
- Later, we may use `accept()` or `NextToken()` to get the next token

The function `accept()` is used to skip over expected tokens

- `accept(TokenType type)` takes a token type as argument
- If the type is the same as the one of the current token, `accept()` simply reads the next token into the current token variable
- Otherwise, it prints an error message and aborts the program by calling `exit()`

Example: Parsing a label:

```
/* Store label value somewhere */
accept(Label); /* We now its a label */
accept(colon); /* And it's always followed by a colon! */
/* Now we can parse the rest of the statement */
```

Parsing a Program

A correct program has to start with either

- a label or
- a variable or
- a `goto` keyword or
- a `loop` keyword

If the current token is one of the above:

- Parse a statement
- Parse the rest of the program

Suggested Data Structure for Program

A program is either empty, or a (labeled) statement, followed by a program

We can map that onto C as follows:

- An empty program is represented by the `NULL` pointer
- A non-empty program contains a pointer to a single statement and a pointer to the rest of the program

```
typedef struct prog_cell
{
   Statement_p statement;
   struct prog_cell *prog_rest;
}ProgCell, *Prog_p;
```

Parsing a Statement

Allocate a `StatementCell` (using `malloc()`)

- Parse the optional label (putting a copy of the string into `statement->label`)
- If label is present, `accept()` the colon token

Decide on statement type as soon as possible, and fill in values as you go along

Interesting case: `loop` statement

- `accept()` the `loop` token
- Put a copy of the next tokens spelling into `statement->argument` (assuming it is a variable)
- `accept()` the variable and the semicolon
- Recursively parse a program, storing the result in statement-¿subprog
- `accept()` `endloop` and semicolon

Suggested Data Structure for Statement

PL is so simple, we can use a single struct for all possible statements:

```
typedef struct statement_cell
{
   long              statement_id; /* Unique identifier */
   StatementType     type;
   char              *label; /* Optional */
   char              *argument; /* Either label for goto or */
                                /* variable for other statements */
   struct prog_cell *subprog; /* For loop */
}StatementCell, *Statement_p;
```

Assignment

Write a program that parses a PL program and prints it back in

a) Correct PL syntax
b) Prefix abstract syntax

Example Output

```
PL-Syntax:
X1<=0;
loop X2;
goto l12;
endloop;
l12: X1++;

Abstract Syntax:
(<program> (<assign0> NoLabel X1)
(<program> (<loop> NoLabel (<program> (<goto> NoLabel l12)
(<program> )
)
)
(<program> (<inc> l12 X1)
(<program> )
)
)
)
```

CSC519
Programming Languages

Introduction to Python

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Midterm Examn

Wednesday, Oct. 16th, 15:00 – 15:50

Topics: Everything we did so far

- General programming paradigms
- Aspects of syntax
  - Grammars and Languages
  - Parse Trees and Abstract Syntax
  - Grammars in BNF
- Imperative Programming and C
  - Flow control constructs
  - Expressions in C
  - Functions and Types
- Subroutines
  - Parameter passing
  - Scoping

Monday we will refresh some of that stuff (but do reread the lecture notes yourself, and check the example solutions on the web)

Python: An Object Oriented Scripting Language

Object oriented languages:

- (Most) data in the state is represented by objects
- Objects are instances of classes and contain both data and methods for operating on the data
- Classes can form a hierarchy, with some classes being derived from other classes by modifying or extending them
- Objects communicate by message passing (but note that this is just newspeak - sending a message is just another term for calling a function associated with the object)

Scripting languages:

- Typically interpreted, speed is not a prime concern
- Variables are dynamically typed (i.e. the type is a property of the value, not the variable) or even untyped
- Declarations are typically not used (since no strict typing occurs)
- Scripting languages often have extensive support for text processing and rewriting (regular expressions, matching,. . .)
- Very often, there are large, existing libraries, and languages offer interfaces to other programs on the system

Python

Python is a modern scripting language

It borrows concepts from

- C/C++ (expression syntax, control flow, classes)
- LISP (dynamic typic, interactive interpreter, lists, lambda-expressions and maps)
- BASIC (string processing, simple syntax)
- Modula-3 (modules, classes and inheritance)
- Perl (regexp support and text processing)
- ...

Python supports both object-orientend programming and a module concept

- The object model allows for multiple inheritance and is implemented using name spaces
- The module concept is a very simple one module – one file model, but allows for separate importing of names

Python: Advanced Ranting

Python is interpreted

- Interpreter can be used interactively for development (shorter debugging cycles)
- Modules can be byte compiled for faster loading
- Speed is so-so

Python is embeddable and extensible

- Can be used as extension or scripting language for other programs
- It is easy to add C functions for speed-critical operations

Python is actually used a lot (and usage is increasing):

- It's behind much of Google
- CML2 (new Linux Kernel configuration tool) is written in Python
- ...

Python has an excellent web site with lots of free documentation at `http://www.python.org/`

Interactive Python Use

To get to the python prompt, just type `python` on the lab machines:

```
[schulz@lee ~] python
Python 1.5.2 (#1, Apr  3 2002, 18:16:26)  [GCC 2.96 20000731
(Red Hat Linux 7.2 2 on linux-i386
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

You can now start typing Python code at the prompt:

```
>>> i = 0
>>> while i < 3:
...     print "Hello World"
...     i = i+1
...
Hello World
Hello World
Hello World
>>>
```

Running Python Programs

Complex Python programs are usually not developed interactively, but using a text editor

– Emacs has a very good Python mode (use extension `.py`)

We can execute Python programs by giving the name of the file as a command line argument:

```
[schulz@lee ~] python hello.py
Hello World
```

Alternatively, we can make Python programs self-contained (under our Lab environment):

– Make the very first line of your program text the following:
  `#! /usr/bin/env python`
– Use `chmod` to make your program executable: `chmod ugo+x hello.py`
– Then you can run the program from the command line:

```
[schulz@lee ~] ./hello.py
Hello World
```

Example: Hello World

```
#! /usr/bin/env python

print "Hello World"
```

Python Program Structure

A Python program is a sequence of statements

- Statements may return values
- Statements may introduce new names and bindings
- In particular, statements can create functions (and name them)

Name spaces are a very central concept in Python!

- Name spaces are used for scoping of names
- Each file has its own global namespace
- Each function creates its own local namespace
- Bindings for names are searched first in local name spaces, then in the global one
- At the moment, every name we create (e.g. by using it at the left hand side of an assignment) is in the global namespace

Comments in Python start with a # and continue to the end of line

```
>>> i = 10 # This is a useless comment
```

Python Data Type

Python is dynamically typed:

- The type is not a property of the variable, but of the value!
- The same variable can hold values of different type
- We can query the type of a value using `type()`:

```
>>> i=10
>>> print i
10
>>> i="Hello"
>>> print i
Hello
>>> type(i)
<type 'string'>
```

Buildin types include:

- Numbers (Integer, floating point, complex)
- Strings (yeah! (and characters are just strings of lenght 1 – double yeah!))
- (Polymorphic) Lists (doubling as arrays)
- Tuples
- Dictionaries (associative arrays or hashes)

Python As a Calculator

Pythons expression syntax is mostly stolen from C

We can type expressions at the Python prompt and get them evaluated:

```
>>> 10 / 3
3
>>> 10 / 3.0
3.33333333333
>>> 10 + 3 *4
22
>>>
>>> (1+5j) + 3j
(1+8j)
```

Assignment works just like in C:

```
>>> a = 20
>>> b = 44
>>> a * b
880
```

Exercises

Go to the Python web site and start reading material (the tutorial is highly recommended)

Start the Python interpreter and use it for a while

– Notice that our installation supports command line editing like the shell!

CSC519
Programming Languages
Introduction to Python – Continued

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Some Python Statements

Printing

- `print` is a build-in command that can be used to print all Python data types
- Multiple arguments are separated by commata, and are printed separated by a single space
- By default, `print` will print a newline. An empty last argument (i.e. a comma as the last character of the statement) will supress this

Assignments

- Assignments use the single equality sign =
- Unlike C, we can assign many values in parallel: `(a,b)=(b,a)`
- If a variable named as an `lvalue` does not exist in the current namespace, it will be created

Expressions

- Expressions can be used as statement
- In interactive use, the value of an expression will be printed back

Example

```
>>> print "Hello ";print "World"
Hello
World
>>> print "Hello ",;print "World"
Hello  World
>>> a = 10
>>> b = 20
>>> print a,b
10 20
>>> (a,b)=(b,a)
>>> print a,b
20 10
>>> 32+b
42
>>>
```

Python Program Structure and Statement Blocks

A single file Python program is a sequence of statements

- Multiple statements can be written on a line, separated by ;
- Hower, normally a single statement is written on a line, terminated by the end of line

Compound statements or block of statements are denoted by indention

- Compound statements are only allowed as part of certain statements (loops, conditionals, statement declarations)
- The controlling part of that statement ends in a colon :
- The first line of the compound statement must be indented by an arbitrary amount of white space relative to the context
- All further statements share this indention (but sub-statements may be indented further, giving a recursive block structure as in C)
- The end of the compound statement is denoted by a statement indented to a the same level as any containing context

Statements in Python do not generally return a value!

Example

```
a = 10
if a==10:
   print "The world seems consistent"
   if 17==3:
      print "I'm surprised!"
      a = 13
   print a
   a = 11

print a
```

```
The world seems consistent
10
11
```

Flow Control: `if`

`if` is used for conditional execution

Example:

```
if a==10:
   print "A = 10"
elif a==11:
   print "A = 11"
else:
   print "A has unrecognized value"
```

An expression is considered `true`, unless:

- It has the numerical value 0 (as in C)
- It's value is a sequence object of lenght 0 (i.e. an empty list)
- It's the special value `None`

Flow Control: `while`

The syntax of `while` is analogous to that of `if`:

```
while <expr>:
    <statment1>
    <statment2>
    ...
```

As in C, `break` and `continue` are supported

- `break` will leave the loop early
- `continue` will immedeatly return to the top

However, the `while` loop also supports a `else` clause:

- The else clause is executed, if the loop terminates because the condition is false
- It's not executed if the loop is left because of a break!

Example

```
i=1
res=1
while i<=10:
   res = res*i
   i=i+1

print "Factorial of 10:", res
```

Function Definitions

Functions are defined using the `def` statement

`def` is followed by a name, a list of formal parameters, and a colon (making up the header), followed by the body (a compound statement)

```
def <name>(<arg1>,<arg2>,...):
    <statement1>
    <statement2>
    ...
```

The function activation ends if a `return` statement is executed

- In that case, the value of the argument of the return is returned as the value of the function call
- Otherwise, execution of the function ends at the end of the body
- In that case, and if `return` is used without argument, the function returns the special value `None`

Example

```
def yes(string,number):
   i=0
   res = ""
   while i<number:
      res = res + string
      i = i+1
   return res

yes("Yes",10)
'YesYesYesYesYesYesYesYesYesYes'
```

Exercises

Write and test functions for Fibonacci number and factorials in Python

Use both recursion and iteration

CSC519
Programming Languages
Midterm Review

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Programming Language Paradigms

Imperative/Procedural: Statements change the global state of the computation

Object-Oriented: Objects send and receive messages that change their internal state

Functional: Function evaluation, no explicit "state"

Declarative: Describe the problem, not the computation

Mixed/Multi-Paradigm: Any or all of the above

In practice, few languages are "pure"

- LISP: Functional, with explicit state
- Python: Object oriented, with many procedural and functional elements
- C++: Object-Oriented, with procedural sublanguage
- Prolog: Declarative, but allows cheating

Language Description

Syntax

- What is a legal program?
- Normally decribed formally (with additional, informal constraints), using grammars

Semantics

- What does it mean?
- Nearly always described informally (sometimes incomplete attempts at formal descriptions)

Environment (often implementation-dependent)

- How do programs interact with the rest of the world?
- What external functions/libraries are available?

Arithmetic Expressions

Different Forms:

- Prefix (i.e. LISP)
- Postfix (i.e. Forth, Postscript)
- Prevalent form: Mixfix (infix for binary operators, most other operators prefix)

Problem with infix/mixfix: Ambiguous

- Written representation allows different interpretations

Solutions:

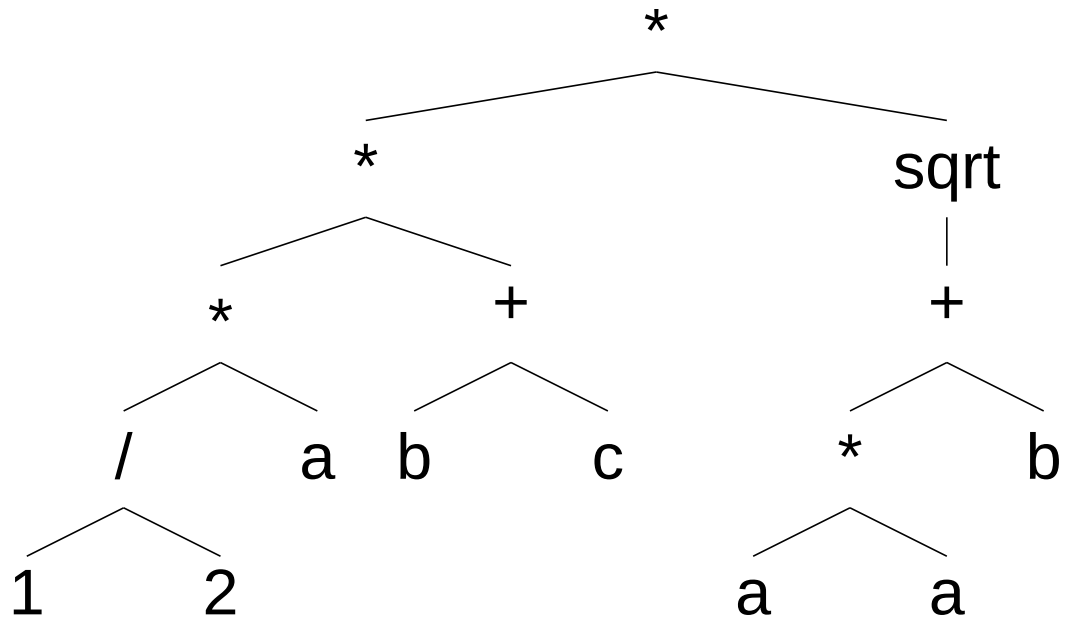
- Operator Precedence
- Associativity

Abstract Syntax Trees

Definition: An abstract syntax tree for an expression is a tree with:

- The internal nodes are labeled by operators
- The leaves are labeled by variables or numbers
- Each operator node has trees corresponding to its arguments as children.

Example:



For suitable grammars, a parse tree can normally be translated into an abstract syntax tree trivially

Formal Languages

Definition: Let Σ be an finite, non-empty alphabet.

- A word over Σ is a sequence of elements of Σ
- The empty word is written λ
- Σ^* is the set of all words over Σ
- A formal language L over Σ is a subset of Σ^*

Programming languages and formal languages:

- The set of all syntactically correct programs in a language is a formal language
- The set of all identifiers in a language is a formal language
- The set of all keywords is a (typically finite) formal language

Formal Grammars

A grammar for a language describes how to generate words from it. Grammars allow us to finitely describe many infinite languages. Grammars also lead to a classification of formal languages.

Definition: A formal grammar is a tuple (Σ, V, S, P), where

- Σ is a finite, non-empty alphabet (of terminal symbols, letters of words in the formal language we want to describe)
- V is a finite, non-empty alphabet (of non-terminal symbols)
- $S \in V$ (called the start symbol)
- $P \subseteq ((\Sigma \cup V)^* \times (\Sigma \cup V)^*)$ is a finite set of productions (or rules). We write $l \to r$ if (l, r) in P.

Regular Grammars

Context-Free Grammars

The Chomsky-Hierarchy

We distinguish between 4 types of languages:

- Regular languages (type 3)
- Context-free languages (type 2)
- Context-sensitive languages (type 1)
- Recursively enumerable languages (type 0)

All languages of type i are also of type i-1:

- Type 3 languages $\subset$ Type 2 languages $\subset$ Type 1 languages $\subset$ Type 0 languages $\subset$ Arbitrary languages
- All of these inclusions are proper

An equivalent classification can be achieved by distinguishing between different types of automata that accept a given language

The existance of a grammar G of type i proves that $L(G)$ is of type i. It does not prove that $L(G)$ is not of type i+1. This requires different proof techniques (usually based on automata theory)

Backus-Naur Form

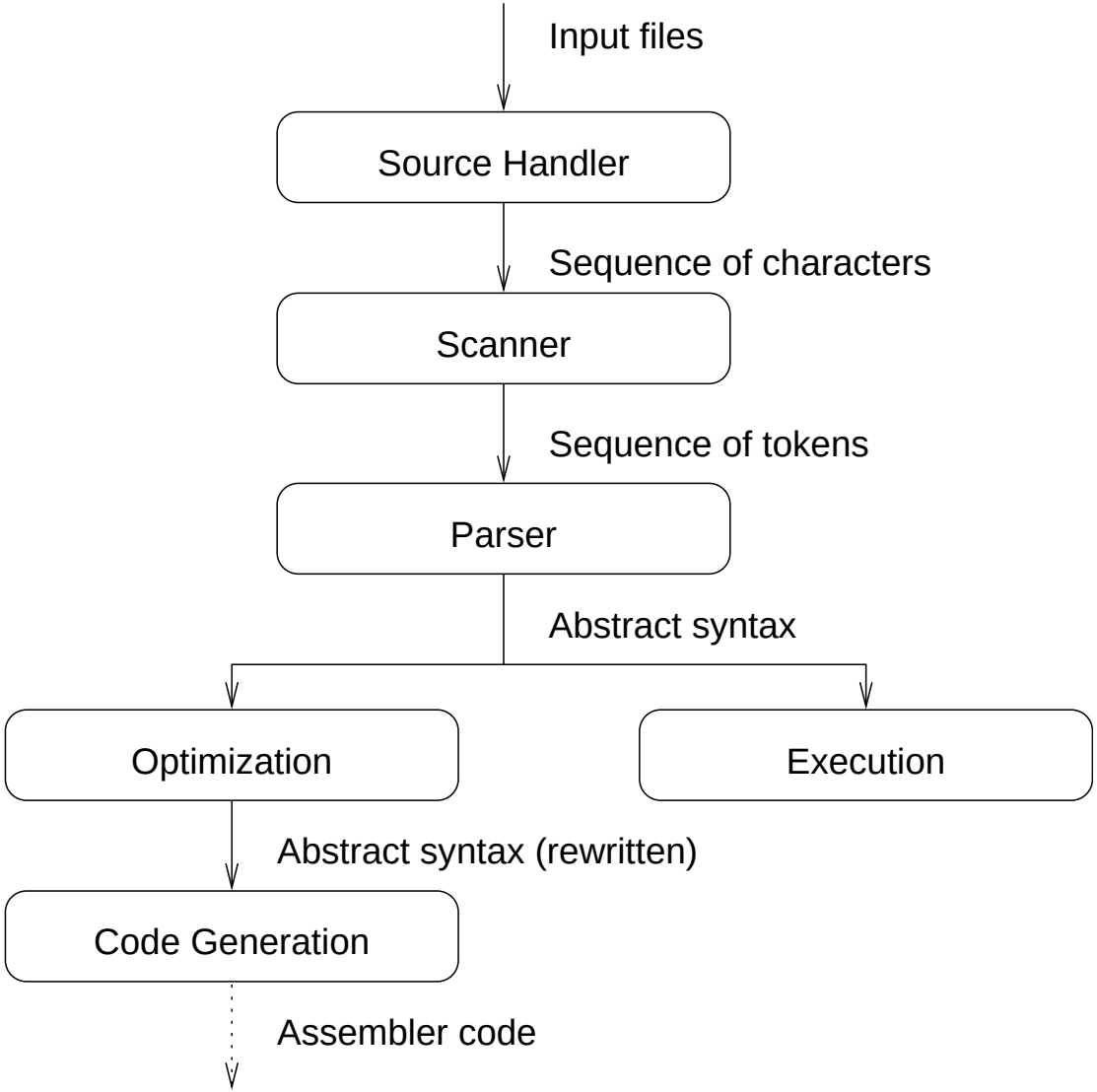
BNF is a more convenient way to write context free grammars, introduced for the Algol-60 report

Conventions:

- Non-Terminals are enclosed in angle brackets: <program>
- Terminals are written in plain text or enclosed in single quotes for emphasis (important if they could also be misunderstood as meta-symbols)
- Both terminals and non-terminals are only defined implicitely
- The start symbol usually is the first non-terminal occurring (or obvious from the context)
- Different productions for the same non-terminal are combined into one rule, using the vertical bar (|) to separate alternatives
- BNF uses the symbol ::= ("defined as" or "can be") instead of $\rightarrow$ for productions
- <empty> is used to denote the empty string (λ)

Example rule: <integer-part> ::= <empty> | <digit-sequence>

Compiler/Interpreter Architecture



Compiler

Interpreter

Flow Control Concepts

Programming patterns:

- Conditional Execution (if/then/else, switch/case)
- Loops

Unstructured flow control:

- Labels and goto

Structured flow control:

- Dedicated loop constructs
- Single entry/Single exit

Examples:

- `while`-loops
- `for`-loops (classical and C versions)
- `repeat/until` vs. `while/do`

Subroutines/Functions/Procedures

Subroutines are used to structure programs

A function is a named subroutine

- It is called via its name
- It can accept (any number of) parameters
- It returns a value of a certain type and can be used in an expression

A procedure is a named subroutine

- It is called via its name
- It can accept (any number of) parameters
- It does not return a value, and can be used as a statement, extending the set of operators of a language

Modern languages (and C) do not distinguish between the two

- A procedure is simply a function returning type void (i.e. nothing)

Parameter Passing

Different ways of handling the relationship between formals and actuals:

Call by value

- Formal parameters are purely local variables initialized to the value of the actual parameter
- Actual parameters are never changed by the function

Call by reference

- The formal parameter is just an alias for the actual parameters
- Changes to formals immediately affect actual parameters

Call by value-result

- As call by value, but on function exit, the current values are copied back to the actuals

Call by name

- Actual parameter is passed as a name and evaluated at each use

Scoping

Dynamic scoping results from naive macro expansion:

- A called function inherits the visible names of the caller

Lexical (or static) scoping is the standard for most modern programming languages

- The visibility of names is controlled by the static program text only
- The semantics of a program is invariant under consistent renaming of local variables

Scoping in C:

- Visibility of names is controlled by declarations
- There are two kinds of declarations in C
  - Declarations written inside a block are called local declarations
  - Declarations outside any block are global declarations
- The scope of a local declaration begins at the declaration and ends at the end of the innermost enclosing block
- The scope of a global declaration begins at the declaration and continues to the end of the source file

Programming in C

The C preprocessor

Functions and declarations

Statements and expressions

Flow control statements

– `while`, `for`, `do`, `if/else`, `switch`, `goto`

Type system

– Basic types/Enums
– Arrays
– Structures/Unions
– Pointer and dynamic memory handling

Simple input/output

Exercises

Reread the lecture notes and check the old excercises

Downlad and understand the solutions to the assignments

CSC519
Programming Languages

Mapping PL to C

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

A First PL Implementation

To get PL running, we first write a PL to C compiler

- Frequent technique:
  - Bigloo Scheme compiles to C
  - GNU f2c implements FORTRAN that way
  - First Bell Labs C++ compilers compiled to C
- Advantages:
  - Languages runs whereever C compiler is available
  - Profits from good C backend/optimizer
  - Mapping to C is easier than mapping to assembler
- Disadvantages:
  - Compiler is slower
  - Not all optimizations possible (compilation to C may loose information)

Simplifications

PL integers are represented by C `unsigned long`

- Limited range
- Easy to implement

Conditions on labels not checked

- C compiler will complain about multiply defined labels

Conditions on `goto` not checked

- Behaviour is undefined (inherits C semantics)

Translation of Language Elements

Names:

- PL variables are valid C variable names
- PL labels are valid C labels

PL arithmetic operations are easily translated into C:

- `X1<=0;` $\Longrightarrow$ `X1=0;`
- `X1++;` $\Longrightarrow$ `X1++;`

PL goto can be implemented using C goto

-
- `goto l14;` $\Longrightarrow$ `goto l14;`

Handling of Loops

PL loop semantics is different from any C loop!

```
loop X1;
   P;
endloop;
```

– The PL loop will execute X1 times, regardless of how or if the value of X1 changes
– C loops always dynamically check conditions!

Solution: Use a hidden new C variable for each loop

```
{
   unsigned long loop_var1;
   for(loop_var1 = X1; loop_var1>0; loop_var1--)
   {
      P;
   }
}
```

Refinements

To be remotely useful, PL needs `bignums` (arbitrary sized integers)

– . . . because arrays can only be realized using prime number encoding!

To make the transition easier, we can generate pseudo-code, and use the C preprocessor to generate plain C

```
#define INC(X)   X++;
#define SETNULL(X)  X=0;
#define PLINT    unsigned long
...
SETNULL(X1)
INC(Y1)
```

To substitute a new data type, we only need to change the header file defining the macros!

Runtime Environment

Convention: An executable PL program receives arguments on the command line

- Arguments are interpreted as numbers
- The first argument is used to initialize `X1`, the second one to initialize `X2`,. . .
- If there are fewer arguments than input variables, the result is undefined (reasonable: error message, more useful: additional variables get initialized to 0)

If the PL program terminates, the values of the output variables are printed, in the form `Z1=...`, one variable per line

To achieve this, we need additional C code:

- Startup code to initialize the variables
- Final code to print results

Assignment

Write a code generator for PL, generating valid C code

Write runtime support code for PL

Put things together to write a PL to C compiler that generates a compilable C program and test it

CSC519
Programming Languages

The Python Type System

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Python Type System

The Python type system is dynamic

– Type is a property of the value, not the variable

Every python value is an object (though not necessarily an instance of a class - build-in types are not typically classes)

Typical object properties:

– Identity: Each object has an identity (e.g. it's address in memory). `id(t)` returns the a representation of the identity of `t`.
– Type: Each object has a type (which tells us how to interprete the bit pattern in memoty). `type(t)` returns the type of `t`
– The value of an object is what we are normaly concerned about. Values may be immutable (e.g. the value of the number 1) or mutable (e.g. the value of a list, to which we can add elements), depending on the type of the object
– Objects may have methods and attributes associated with it

Basic Build-In Types

Python implements a number of predefined types. Some important ones are:

- **None:** A type with the single value `None`
- **Numbers**
  - Integers
  - Long integers
  - Floating point numbers
  - Complex numbers
- **Sequences**
  - Strings (over ASCII characters)
  - Tuples (over arbitrary values)
  - Lists (of arbitrary values)
- **Dictionaries** (associative arrays)

Numbers

Integer numbers are (at least) 32 bit fixed width integers in two's complement representation, e.g. they can at least store values from -2147483648 to 2147483647

– Integer literals are written as normal integer numbers

Long integers can be any size (memory permitting)

– Literals are written as sequences of digits, followd by the letter `l` or `L`
– Long integers will never overflow!

Floating point numbers are normal, machine-level floating point numbers (typically implemented `double` in C)

– Floating point literals can be written with decimal point or exponential (`1e3`)

Complex numbers are tuples of two floating point numbers

– The are written as `3+4j`, where `3` is the real part of the number, and `4` is the imaginary part

Machine Integers vs. Long Integers vs. Floats (1)

```
a=1
while 1:
   print a
   a=a*256

1
256
65536
16777216
Traceback (most recent call last):
  File "<stdin>", line 3, in ?
OverflowError: integer multiplication
```

Machine Integers vs. Long Integers vs. Floats (2)

```
a=1L
while 1:
   print a
   a=a*256
```

```
1
256
65536
16777216
4294967296
1099511627776
281474976710656
72057594037927936
18446744073709551616
4722366482869645213696
1208925819614629174706176
309485009821345068724781056
79228162514264337593543950336
20282409603651670423947251286016
5192296858534827628530496329220096
...
```

Machine Integers vs. Long Integers vs. Floats (3)

```
a=1.0
while 1:
   print a
   a=a*256
```

```
1.0
256.0
65536.0
16777216.0
4294967296.0
1.09951162778e+12
2.81474976711e+14
7.20575940379e+16
1.84467440737e+19
4.72236648287e+21
1.20892581961e+24
3.09485009821e+26
...
7.02223880806e+305
inf
...
```

Some Build-In Functions on Numbers

Conversion of numbers (Note: If a conversion is impossible because the new type cannot represent the value, an `OverflowError` exception will be generated)

- `float(number)` will convert any non-complex number into a floating point number
- `int(number)` will convert a number into a normal integer number, truncating the fractional part (if any)
- `long(number)` will convert a number into a long integer number. `long(1)=1L`
- `complex(n1,n2)` will create a complex number (with value `n1+n2j`)
- `i.real` will return the real part of a complex number, `i.imag` will return the imaginary part

Other operations:

- `abs(number)` will return the absolute value of any number (i.e. its distance from 0 in the number space):

```
>>> abs(1+1j)
1.4142135623730951
>>> abs(-4.3)
4.299999999999998
```

A large set of mathematical functions (`sqrt()`, `sin()`, `cos()`, `...` is defined in the `math` module

Some Operations on Numbers

Python supports the normal arithmetic operations for all numerical types:

- Addition, Subtraction, Multiplication, Division, Modulus (%)
- Division is truncating on (long or plain) integers, floating point otherwise
- Modulus always also handle floating point numbers!

```
>>> 17.5 %2
1.5
>>> 3 % 1.5
0.0
```

Numbers can be compared using

- <, >, <=, >=
- == compares for equality
- != compares for inequality

Sequences

Sequences are composite objects with components in a fixed and meaningful order

Sequence types include strings, tuples and lists

- Strings are immutable sequences of ASCII characters
- Tuples are immutable sequences of arbitrary objects
- Lists are mutable sequences of arbitrary objects

All sequences support the extraction of elements and subsequences (slicing), using a C-indexing like notation:

```
>>> "Hallo"[1]
'a'
>>> "Hallo"[2:4]
'll'
```

All sequences support the build-in function `len()` returning the number of elements

```
>>> len("ahsa")
4
```

Strings

Strings are immutable sequences of ASCII characters (note: Python supports UNICODE strings as a separate but essentially similar data type)

String literals can be written in many different ways:

- Plain strings, enclosed in double or single quotes (`"Hi"`, `'You'`). Escape sequences like `'\n'` will be expanded in a manner similar to C
- Raw strings: `r"Hallo\n"`: No escape seqences are processed
- Tripple quoted strings can span multiple lines, literal newlines are part of the string

While strings are immutable, we can create new strings easily and efficiently

- Slicing allows us to extract substrings
- `'+'` can be used to concatenate strings

String Examples

```
>>> a ="""
... This is a multi-line string
... Here is line 3, lines 1 and 4 are empty!
... """
...
>>> a
'\012This is a multi-line string\012Here is line 3, lines 1 and 4 are empty!\012'
>>> print a

This is a multi-line string
Here is line 3, lines 1 and 4 are empty!

>>> b = a[1:28]+" "+a[29:69]
>>> b
'This is a multi-line string Here is line 3, lines 1 and 4 are empty!'
>>> c="\\\\\\\\"
>>> d=r"\\\\\\\\"
>>> print c
\\\\
>>> print d
\\\\\\\\
```

Exercises

Find out what `type()` returns for `1`, `[]`, `"abc"` . . .

CSC519
Programming Languages

The Python Type System: Lists

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Lists

Lists are mutable ordered collections of arbitrary objects

- List literals are written by enclosing a comma-separated list of elements in square braces (`[1,2,3]`, an empty list is written `[]`)
- We can also create lists from any other sequence type using the `list()` function:

```
>>> list("Hallo")
['H', 'a', 'l', 'l', 'o']
```

Frequent list operations:

- Indexing and slicing for the extraction of elements. Note that you can also assign values to slices and elements to change the list!
- Deleting elements with the build-in function `del()`:

```
>>> a = ["a", "b", "c"]
>>> del(a[1])
>>> a
['a', 'c']
```

List Operations

The list data type has a number of useful methods (functions associated with an object). Note that these functions destructively change the list object (i.e. they do not compute a new list and return the value, but change the old list)

- `l.append(x)` will append the element `x` at the end of `l`
- `l.extend(l1)` will append all the elements in `l1` to the end of `l`
- `l.insert(i.x)`: Insert at position `i` in the list (shifting all elements with indices <=`i` by one)
- `l.remove(x)`: Remove first occurrence of `x` from the list
- `l.pop()`: Remove the last element from the list and return it
- `l.pop(i)`: Return the element with index `i` from the list and return it
- `l.sort()`: Sort the list (assuming there is a > relation
- `l.reverse()`: Reverse the list

List Examples

```
>>> a = []
>>> b = ["a", "b", "c"]
>>> a.append(1)
>>> a.append(2)
>>> a.append(3)
>>> a
[1, 2, 3]
>>> a.extend(b)
>>> b
['a', 'b', 'c']
>>> a
[1, 2, 3, 'a', 'b', 'c']
>>> a.reverse()
>>> a
['c', 'b', 'a', 3, 2, 1]
>>> a.pop()
1
>>> a.pop()
2
>>> a
['c', 'b', 'a', 3]
```

Making Lists of Numbers

The `range()` function creates lists of integers. It can be used with one, two, or three integer arguments

`range(arg1, arg2)` generates a list of integer, starting at `arg1`, up to (but not including) `arg2`:

```
>>> range(10, 20)
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> range(10,0)
[]
```

`range(arg)` (single argument) is equivalent to `range(0,arg)`:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The third argument can be used to specify a `step size`:

```
>>> range(0,20,3)
[0, 3, 6, 9, 12, 15, 18]
>>> range(20,10,-1)
[20, 19, 18, 17, 16, 15, 14, 13, 12, 11]
```

Iterating over Sequences: `for`

`for` iterates over arbitrary sequences (but is most often used for lists)

Syntax:

```
for element in sequence:
   <statements>
```

Remarks:

– Order is well-defined (natural order of elements)
– What happens if you change the sequence in the loop is undefine!

Example:

```
for c in "Hallo":
   print c
```

```
H
a
l
l
o
```

Idiomatic use of `for`

Pascal-like `for`: Iterates over sequence of numbers

In Python: Combine `for` and `range`:

```
for i in range(0,30): # Simulate "for i=0 to 29 do"
   print "Hallo", i
```

```
Hallo 0
Hallo 1
Hallo 2
Hallo 3
Hallo 4
Hallo 5
Hallo 6
...
Hallo 29
```

Operating on Sequences

Python has three generic ways to operate on sequences (normally applies to lists): `map`, `filter`, and `reduce`

`map` applies a certain function to all elements in a list and returns the list of results

`filter` applies a predicate to each element of a list, returning the list of all elements that pass

`reduce` applies a binary function to successively reduce the list to a single element

Note: For all of these operations, we have to actually pass a function as an argument!

- This is a feature the Python inherited from functional programming languages
- In fact, `map` ,`filter`, and `reduce` (and build-in list data types) are common in the functional programming world!

The `map` function

`map` has one functional argument, and any number of list arguments

- The first one is a function (normally a function name), to be applied to the others
- The number of arguments of that function determines, how many additional arguments there have to be
- Each of the additional arguments hast to be a list

`map` applies the function to the tuple made up by taking one element from each of the lists, and generates the list of results

- Of course that means that all lists have to have the same number of elements

Simple case: The functional argument is a function with a single argument

- Then `map` applies that argument to each element of the list

Mapping Example

```
def add(x,y):
   return x+y

def square(x):
   return x*x

def null(x):
   return 0;

>>> a = range(0,10)
>>> b = range(10,0,-1)
>>> c=map(null,a)
>>> print c
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> map(square, a)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> map(add,a,c)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> map(add,a,b)
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
```

Example: Filtering Lists

```
def is_even(x):
   return x%2

def false(x):
   return 0

>>> a = range(0,10)
>>> print a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> filter(is_even, a)
[1, 3, 5, 7, 9]
>>> filter(false, a)
[]
```

Reducing Lists

`reduce` differs from `map` and `filter` in that it reduces a list to a single value

Reduce format: `reduce(fun, list)`

- `fun` has to be a function which accepts two elements of the type making up the list, and returns an element of the same type
- `list` is a sequence of (reasonably) homogenous elements

Semantics:

- If the list is empty, an error exception is raised
- If the list contains a single element, it is returned
- Otherwise, reduce takes the first two elements, applies the `fun` to it, and recursively calls itself with a list where the first two elements have been replaced with the result

Alternative: `reduce` can accept a third argument (to use as a start value)

Reduce Example

```
def add(x,y):
   return x+y

>>> a = range(0,10)
>>> b = range(10,0,-1)
>>> c = ["Hello", " ", "World"]
>>> reduce(add, a)
45
>>> reduce(add, b)
55
>>> reduce(add, c)
'Hello World'

def sum_list(l):
   return reduce(add, l, 0)

>>> sum_list([])
0
>>> sum_list(a)
45
```

Exercises

You can represent vectors as Python lists, and matrices as lists of lists. Write functions that:

- Multiply a scalar with a vector or a matrix (can you make one function that handles both cases?)
- Multiply a two vectors
- Multiply two matrices
- Transpose a matrix (mirror along the diagonal)

CSC519
Programming Languages

Tuples, Dictionaries, Implementation Issues

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Tuples

Tuples are immutable, but can contain objects of any type

- Useful for returning more than one value from a function
- Parallel assignment (`a,b = b,c`) also uses tuples

Tuple syntax:

- Comma-separated list of values: `1,2,3`
- Tuple with one element: `1,`
- Tuple with no elements: `()`

Tuples support `len()` and slicing

Tuple packing and unpacking:

```
>>> a = 10,12,12
>>> a
(10, 12, 12)
>>> b,c,d = a
>>> print b
10
```

More on Slicing

The slicing operation can be applied to any sequence object

- In general, we give two limits
- Slicing returns a subsequence (of the same type, if possible)

Limits correspond to indices, the first one is inclusive, the second one is exclusive

- `[1,2,3,4,5][2:4] == [3,4]`
- Indices that are out of range are silently converted to the biggest/smallest index that makes sense: `[1,2,3,4,5][2:18] == [3,4,5]`
- The if the second index is smaller than the first one, we get an empy sequence as a result: `[1,2,3,4,4][2:1] == []`

We can use negative indices in slicing and indexing

- Negative indexes are interpreted as counting from the last element
- `"Hallo"[-1] == o`
- `a[1,-1]` returns a stripped by the first and last element
- General rule: If index is negative, add `len(seq)` to it to get the normal index

Dictionaries

Dictionaries are another mutable compound data type

- The allow the association of keys and values
- Other names: Associative arrays (AWK), Hashes (Perl)

Dictionaries act like arrays where any (recursively) immutable object can be used as an index

- Written representation: List of `key:value` pairs in curly braces:

```
a ={} # Empty dictionary
b = {1:"Neutral element", 2:"Smallest prime", 3:"Prime",
     4:"Composite, 2*2"}
```

Dictionary Operations

Assume `d` is a dictionary

- `d[key] = value` adds the pair `key:value` to the dictionary (if an old value for `key` existed, it is forgotten
- `del(d[key])` deletes the element with key `key`
- `d.keys()` returns a list of all keys in `d`
- `d.has_key(key)` returns true, if `key` is a key in `d`

Example:

```
b = {1:"Neutral element", 2:"Smallest prime", 3:"Prime",
     4:"Composite, 2*2"}
for i in range(1000):
   if b.has_key(i):
      print i, ":", b[i]
   else:
      print "No information about ",i
```

Implementation of Dynamic Type Systems

Dynamic type systems can be implemented in various ways:

- Boxed types represent each value by a structure containing a type field and additional information representing the value
- Tagged types encode type information and value in a single machine word (if possible)

Boxed types are used by Python, Haskell,. . .

Tagged types are traditionally used by LISP dialects and many abstract machines

Boxed Types

Values in the program are represented by pointers to objects in memory

– Each object contains type and value

Advantages:

– Arbitrary values can be represented in a consistent way
– Passing values is cheap, as only the pointer needs to be passed

Disadvantages:

– High memory overhead
– High cpu time overhead
– Unused objects need to be garbage collected

Example

Assume a language with just 4 data types

- Strings
- Lists (of any other data type)
- Machine integers
- Floating point numbers (double precision)

We give data types and an implementation of addition as an example

Example Datatypes (1)

```
typdef enum  /* Types */
{
   String,
   List,
   Integer,
   Float
}TypesType;

typedef struct cons_cell /* Lists */
{
   struct object_cell *car; /* First object */
   struct list_cell   *cdr; /* Rest of list */
}ConsCell, *Cons_p;

typedef union  /* Values */
{
   char   *string_val;
   Cons_p list_val;
   int    int_val;
   double float_val;
}ValueRep;
```

Example Datatypes (2)

```
typedef struct object_cell
{
   TypesType type;
   ValueRep value;
}ObjectCell, *Object_p;

Object_p NewObject(TypesType type)
{
   Object_p res = malloc(sizeof(ObjectCell))
   /* Check for NULL omitted */
   res->type = type;

   return res;
}
```

Example Code

```
Object_p add(Object_p x1, Object_p x2)
{
   Object_p result;

   if(x1->type == Integer)
   {
      if(x2->type == Integer)
      {
         result = NewObject(Integer);
         result->value.int_val = x1->value.int_val+
                                 x2->value.int_val);
         return result;
      }
      else if(x2->type == Float)
      {
         result = NewObject(Float);
         result->value.int_val = x1->value.int_val+
                                 x2->value.float_val);
         return result;
      }...
```

Example Code (Continued)

```
      ...
      else
      {
         Error("Incompatible types in addition!");
      }
   }
   else if(x1->type == Float)
   {
      ...
   }
}
```

Compare plain C code with static type system:

```
res = x1+x2;
```

Tagged Types

Motivation: We want to avoid some of the overhead of boxed types

Observation 1: Most values in typical scripting- and AI programs are either pointers (to list objects or hashes or...) or small integers

– If we can find an efficient representation for those, we gain a lot!

Observation 2:

–
– Typical machine word size on early mini-computers: 32 Bits
– Typical addressable memory: Kilobytes to a few megabytes
– Conclusion: We never need 32 Pixels to adress pointers!

If we use e.g. the top 4 bits from each machine word for tagging, we can still addess 256 Megabyte!

– More than anybody can ever afford!
– Also allows the representation of integers up to a couple of millions, covering most cases

Tagged Types (2)

For virtual address systems, tagged data requires support from the operating system (use low adresses only)

Systems actually even added hardware support for tagged type!

- SUN Sparc processor have support for 2 bit tags (out of 32)
- Symolics LISP machines had explicit tag bits

Note that we can tag only a small number of types!

- Typically types that fit into the remaining bits of the machine words: Small integers, pointers, characters,. . .
- For other types: Pointer to a structure containing the value
- If there are more types than can be represented by the tag bits, we need one "other" type with a pointer that points to a boxed type!

Example: Integer Addition with 2 Bit Tags

```
#define TAG_MASK 0xB0000000 /* Bits 30 and 31 tag, bits 0-29  value */
#define INT_TAG    0x80000000 /* 10 */
#define STRING_TAG 0x40000000 /* 01 */
#define LIST_TAG   0xB0000000 /* 11 */
#define FLOAT_TAG  0x00000000 /* 00 */

typedef int ValueType;
```

Example: Integer Addition with 2 Bit Tags

```
ValueType add(ValueType x1, ValueType x2)
{
   int val1, val2, res;

   if(x1 & TAG_MASK == INT_TAG)
   {
      val1 = x1 & ~TAG_MASK; /* Remove tag bits from value */
      if(x2 & TAG_MASK == INT_TAG)
      {
         val2 = x2 & ~TAG_MASK;
         res = val1 + val2;
         if(res & TAG_MASK)
         {
            /* Overflow handle code here */
         }
         return res | INT_TAG;
      }
      else ...
   }
}
```

Exercises

Write some Python programs using dictionaries and tuples

- Compute Fibonacci-Numbers recursively (but from the bottom up), passing tuples (f(n-1), f(n))
- Generate dictionaries of factorized numbers, storing lists of factors (either all factors or prime factores) as values

CSC519
Programming Languages
Garbage Collection/Exceptions

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Memory Handling

C and similar languages use explicit memory handling

- Memory is explicitely allocated by the user for creating new objects (`malloc()`)
- After use, this memory has to be explicitely returned by the user (`free()`)

In higher level languages, objects are created automatically:

```
result = NewObject(Integer);
result->value.int_val = x1->value.int_val+
                        x2->value.int_val);
return result;
```

What happens if they are now longer needed?

Garbage Collection

In order to be viable for general programming tasks, high-level languages have to reclaim unused memory

- This process is called garbage collection
- There is a large number of different garbage collection schemes – garbage collection is an old and rich research field
- Which algorithm is most suitable depends on the memory access patterns of the program and on other constraints

We distinguish between two kinds of garbage collection algorithms:

- Incremental
- Periodic (all-or-nothing)

Advantages of garbage collection:

- No memory leaks (if garbage collection algorithm is perfect)
- May be more efficient (!) (but may be less efficient either)
- Fewe things for the programmer to worry about

Garbage Collection (Continued)

Incremental garbage collection algorithms collect garbage as soon as possible

- Advantage: Memory is immediately available for reuse
- Advantage: No noticable slowdown
- Disadvantage: Potentially higher overhead
- Disadvantage: Memory fragmentation

Periodic garbage collection algorithms periodically collect all (or most of the) garbage

- Advantage: Lower overhead
- Advantage: Memory still in can be compacted, improving locality of reference and cache hit rates
- Disadvantage: Program seems to stop for some time at unpredictable intervals

Reference Counting Garbage Collection

Idea: Each object contains a counter, counting the number of references to it

- If that counter reaches zero, the object can be freed
- Implementation is relatively straightforward (just use macros for assignment that handle counters in old and new objects):

  Phython: `a = b`

```
C: => a->refcount--;
      if(a->refcount == 0)
      {
         free(a); /* May also need to remove references from
                     objects *a points to (e.g. if *a is a
                     list) */
      }
      a = b;
      b->refcount++;
```

Reference Counting Discussion

Reference counting garbage collectors:

- Are very easy to implement
- Have low overhead
- Are naturally incremental (but can be delayed)

Are there any problems?

A Koan from Elder Times

One day a student came to Moon and said:

"I understand how to make a better garbage collector. We must keep a reference count of the pointers to each cons."

Moon patiently told the student the following story:

"One day a student came to Moon and said: 'I understand how to make a better garbage collector. . . ' "

Problem: Cyclic References

Consider this (valid) piece of Python code:

```
a = [1,2,3]
a.append(a)
a=0
```

- Obviously, after the last assignment, we have no handle to the list anymore – the list is garbage
- However, there still is a reference to the list – from within the list
- This situation is called a cyclic reference

Pure reference counting garbage collection cannot recover objects which contain cyclic references

- One solution: Use reference counting if possible, occasionally use an additional all-or-nothing algorithm
- The current Python implementation uses reference counting garbage collection

Mark-and-Sweep Garbage Collection

Every object contains a binary flag

- During garbage collection, all reachable objects are marked by setting the flag to a certain value
- Then, a sweep of all objects recovers those that are not marked

This is a periodic algorithm

- Advantage: Can handle cyclic references easily
- Disadvantage: We need to keep track of all allocated objects (for the sweep phase)
- Disadvantage: We need to know the type of all objects in the garbage collector (to be able to follow references)
- Disadvantage: We need to access all live variables
- Disadvantage: Garbage collection can cause noticable delays (bad for real-time and interactive programs)

Dynamic Error Handling in Programming Languages

There are a number of different ways to handle errors or unexpected situations in a program:

- Aborting
- Signalling the error via return values/global variables
- Exceptions

Solution for C:

- A function that encounters an error returns an out of band value, i.e. an illegal value (often -1 or `NULL`, depending on the function)
- Additionally, the global variable `errno` is set to an error code describing the error

Disadvantages:

- Some function have no obvious out-of-band values
- In recursive calls, each function has to pass the error code back!

Exception-Based Error Handling

In exception-based error handling, an error raises an exception

– Exceptions may have different values to signal the type of error

This exception is passed back the function call hierarchy until it is caught (and can be handled)

– It's possible for a function to handle some types of exceptions, but not others

Advantages:

– Much more elegant!
– No need for special reserved values
– Error handling code is only necessary in the functions that catch the exceptions

Disadvantages:

– Expensive (in CPU time)
– Needs language support, complicates calling sequence

Exceptions in Python

In Python, run time errors cause exceptions

– Large number of predefined exception values

Exceptions can also be explicitely raised by a user program

– Programs can raise both predefined and user-defined exceptions

Some predefined exceptions:

– `ZeroDivisionError`
– `NameError` (used an undefined name)
– `KeyError` (used a non-existing dictionary key)
– `OverflowError` (Result of operation does not fit into type)

User defined exception in Python 1.5.2 are strings

– Note that exceptions are compared by object id, not by value!
– Two instances of `"a"` may generate different objects (or the may return references to the same one)

Language Support for Exceptions

Syntax for catching exceptions:

```
try:
   <program>
except <ExceptionNameList>:
   <exception handling code>
else:
   <optional normal code>
```

Syntax for raising exceptions:

```
raise <exception>
```

Example

```
stack = []
StackEmpty = "StackEmpty"

def Push(element):
   stack.append(element)

def Empty():
   return len(stack)==0

def Pop():
    if(Empty()):
        raise "StackEmpty"
    return stack.pop()

def PrintAndPopAll():
   try:
      while 1:
         print Pop()
   except StackEmpty:
      print "Stack is empty"
```

Example Continued

```
>>> Push(10)
>>> Push("Hello")
>>> Push("Halloween Surprise Examn")
>>> PrintAndPopAll()
Halloween Surprise Examn
Hello
10
Stack is empty
```

Exercises

Read about garbage collection at `http://www.iecc.com/gclist/GC-faq.html`

Write a function that computes

$$\sum_{i=1}^{n} \frac{1}{x^i}$$

for arbitrary values of n and x. Make sure you handle the case of a division by zero error caused by an underflow of x^i (if $x < 0$ and i becomes very big, using an exception handler

Write a function for accessing dictionaries that uses an exception handler to return a default value if a key cannot be found in a dictionary

CSC519
Programming Languages

I/O in Python

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Suggested Topics for Student Talks

Strange languages (`http://www.tuxedo.org/~esr/retro/`):

- intercal
- Unlambda
- *var'aq*
- Orthogonal (`http://www.muppetlabs.com/~breadbox/orth/home.html`)
- Befunge `http://www.catseye.mb.ca/esoteric/befunge/`

Normal languages:

- Any of those from your papers

Suggested Topics for Student Talks (2)

Interesting Language Concepts:

- Coroutines
- Threads
- Continuations (Hard!)
- Lazy evaluation

Garbage Collection Techniques

- Copying Garbage Collectors
- Garbage collectors for multi-processor systems

Practical Hints

Format: 10 minutes per talk, up to 5 minutes discussion

Presentation:

- Concentrate on important ideas (don't waste time on filler)
- Start with something interesting (if you don't capture your audience in the first 3 minutes, they are lost forever!)

Visual aids:

- Calculate between 90 seconds and 3 minutes per slide
- Avoid details, concentrate on ideas
- Use big fonts (not more than 6 ideas per slide)

If you need help, ask me!

- We have an overhead projector for slides (or paper)
- My laptop can project PDF or PostScript (but not Powerpoint)

Using Python Modules

Python uses modules to selectively add capabilities to the language

Modules may be compiled into the interpreter, or may correspond to external files

- System modules offer additional services
- User-defined modules can be used to structure larger applications

Using modules:

- `import Module` makes names in `Module` available as `Module.Name`
- `from Module import Name` makes `Name` available unqualified
- `from Module import *` imports all names from `Module` (not normally recommended!)

The `sys` Module

The `sys` module is the interface to assorted system information:

- `argv` is a list of strings giving the command line of the program (it's empty in interactive use)
- `stdin` is a file object corresponding to UNIX/C `stdin`
- Similarly, there are `stderr` and `stdout`
- `exit()` will exit the program
- `exit(arg)` will exit the program and return the value of `arg` to the environment (in an implementation-defined way)
- `getrefcount(obj)` returns the current number of references to `obj`

Example

```
$ python
>>> sys.argv
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'sys' is not defined

>>> import sys
>>> sys.argv
['']
>>> sys.stdin
<open file '<stdin>', mode 'r' at 0x80daea8>

>>> from sys import *
>>> stdout
<open file '<stdout>', mode 'w' at 0x80daf18>
>>> getrefcount(1)
55
>>> exit()
$
```

File-Based Input

Scripting languages often have very convenient text I/O functions – Python is no exception

I/O is performed on file objects

File objects are created with `open(Name,Mode)`

- `Name` is a string containing a valid filename
- `Mode` is a string containing the mode (e.g. reading (`"r"`) or writing (`("w")`))

If `f` is a file object, the following methods are defined:

- `f.readline()`: Read a single line and return it (including `'\n'`). Return empty string on EOF
- `f.readlines()`: Read the whole file into a list of strings, one line per element
- `f.read()` reads the whole file into a single string
- `f.read(bytes)` reads up to `bytes` into a string

Example: Sorting Files

```
#!/usr/bin/env python

import sys

if len(sys.argv) == 1:
    l = sys.stdin.readlines()
else:
    l = []
    for name in sys.argv[1:]:
        f = open(name,"r")
        l.extend(f.readlines())
        f.close()

l.sort()

for line in l:
    print line,
```

More on Non-Standard Control Flow

Note: Open files are a finite resource!

– There is a per-process limit on the number of open files

How do we make sure that an open file gets closed if an exception occurs?

More generally, how can we easily ensure that cleanup code gets called regardless of how we leave a block of code?

```
...
f=open("/etc/passwd", "r")
line = f.getline()
while line:
   if line[0:6]=="schulz:":
       return line
   line = f.getline()
f.close()
return ""
```

try...finally

The `try` statement also can be used with a `finally` clause instead of the except clause

- The program part in the finally clause will be executed regardless of how control leaves the `try` clause
- If an exception causes the exit of the `try` clause, this exception will be re-raised after the `final` clause

Example:

```
f=open("/etc/passwd", "r")
try:
   line = f.getline()
   while line:
      if line[0:6]=="schulz:":
          return line
      line = f.getline()
finally:
   f.close()
return ""
```

CSC519
Programming Languages
Regular Expressions and Strings

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Suggested Topics for Student Talks

intercal
Unlambda
$var'aq$
Orthogonal
Befunge
Any normal programming language
Coroutines
Threads
Continuations (Hard!)
Lazy evaluation
Copying Garbage Collectors
Garbage collectors for multi-processor systems

Regular Expressions

Regular expressions are a concise ways of specifying regular languages

The are most often used as search patters

- Return true if a word is in the language described by a regexp
- Return true if a string contains a word that is in the language
- Return the word found (if any) that matches
- Also: Search and replace (replace matching RE's)

Regular expression support is at the core of most advanced string processing tools

- Various levels of RE support are availabe in nearly all scripting languages

Regular Expression Format

A regular expression is described by a string

- It contains normal characters and `meta-characters`
- In Python, the meta-characters are `. ^ * + ? { } [ ] \ | ( )` and `$`, all other characters are normal

Simple matching rules:

- A normal character in a regular exression matches the same character in a string
- A sequence `[`$c_1 \ldots c_n$`]` matches any character between the two braces
- A sequence `[^`$c_1 \ldots c_n$`]` matches any character except those between the two braces
- A dot `.` will match any single character (except for newlines)
- A sequence `\c` will match `c`, even if it is a meta-character
- A sequence of regexps will match, if each component matches a subseqent part of a string: `ab..a` matches `abD1a`
- Regular expressions can be enclosed in parentheses for grouping

Regular Expression Format (2)

Pattern repetition: Let `r` be a regular expression

- `r*` matches everything that any sequence of `r`'s would match
- `r+` is equivalent to `rr*`
- `r?` matches anythin `r` matches, or the empty string

Examples:

- `a*b` matches `aaab`, `aaaaaaab`, `b`
- `(Hallo)+` matches `HalloHallo`
- `[_a-zA-Z][_a-zA-Z0-9]*` matches any C identifier

Pattern disjunction:

- If `e` matches a certain language L_e, and `f` matches L_f, then `e|f` will match $L_e \cup L_f$

Python Support for Regular Expressions

Regular expression support is included by importing the `re` module

– To use the facilities, `import re`

Regular expressions are written as strings, but need to be compiled to an internal format for efficient matching

Hint: Regular expression that need a lot of `\` characters to escape the special meaning of characters are hard to write as normal strings. . .

– . . . because normal strings already use `\` to escpae special meanings
– . . . but we can use Python raw strings: `re.compile(r"\[|\]")` will create a regulare exression that matches `[` and `]`

Basic RegExp Functions

Creating regular expressions: `re.compile(arg)`

– Compile a string into a regular expression object and return it

Some methods of regular expression objects:

– r.match(string): Return a match object if `r` matches any initial substring of `string`, `None` otherwise
– `r.search(string)` : Return a match object if `r` matches any substring of `string`, `None` otherwise

Match Objects

Match objects contain information about a successful match. Let `m` be a match object

- `m.group()` returns the string matched
- `m.start()` returns the index of the first character matched
- `m.end()` returns the ending position of the match (i.e. the index of the first character not matched)
- `m.span()` return a tuple (start, end)

Example:

```
>>> r = re.compile("H.*o")
>>> m = r.search("Is there a hello in here?")
>>> print m
None
>>> m = r.search("Is there a Hello in here?")
>>> print m
<re.MatchObject instance at 80ce5d8>
>>> m.span()
(11, 16)
>>> "Is there a Hello in here?"[m.start():m.end()]
'Hello'
>>> m.group()
'Hello'
```

Example

```
!/usr/bin/env python

import sys
import re

def grep(regexp,file):
    "Print all lines from file matching regexp"
    lp = file.readline();
    while lp:
        if regexp.search(lp):
            print lp,
        lp = file.readline()

f=open(sys.argv[2],"r")
pattern = re.compile(sys.argv[1]);
grep(pattern,f)
f.close()
```

More RegExp Operations

Regular expressions can be used to build "good enough" parsers for many purposes

- Frequent requirement: Parse components of a list or extract certain fields of the input

Let `r` be a regular expression

- `r.split(string)` will return a list of substrings of `string` which are separated by a string matching `r`
- Example:

```
>>> import re
>>> import string
>>> sep = re.compile(" *, *")
>>> l = sep.split("This,  sentence ,, has,to many,comas!")
>>> print l
['This', 'sentence', '', 'has', 'to many', 'comas!']
>>> print string.join(l,' ')
This sentence  has to many comas!
```

- `r.split(string, limit)` will split the string into at most `limit` substrings

Search and Replace

The `sub()` function replaces strings matching a patter with new strings:

- `r.sub(repl, string)` will create a new string in which each string matching `r` will be replaced by `repl` Example:

```
>>> import re
>>> s = "This sentence  has to many      spaces!"
>>> r = re.compile(" *")
>>> r.sub(" ", s)
' T h i s s e n t e n c e h a s t o m a n y s p a c e s ! ' # Ups ;-)
>>> r = re.compile(" +")
>>> r.sub(" ", s)
'This sentence has to many spaces!'
```

- Variant: `r.sub(repl, string, count)`: Limit replacing to at most count occurences:

```
>>> r.sub(" ", s, 3)
'This sentence has to many      spaces!'
```

Assignment

Write a simple Python program that acts similar to a stripped-down version of the UNIX stream editor **sed**:

- Any command line argument that does not start in a - (dash) is seen as an input file name
- Any command that starts in a dash is an editor command
- The program reads all lines from all input files (in the specified order, if no input files is present, read `stdin`), executes all editor commands on each line, and prints it to the standard output.

Editor commands:

- `-g<pattern>`: Grep for the pattern (a regular expression string), i.e. print only lines in which a string matching the pattern occurs
- `-s<pattern>/<repl>`: Substitute all strings matching `<pattern>` with `<repl>` before printing

Each option can occur multiple times, the effect is the same as if the input would be piped through a sequence of programs:

```
mysed.py -gHello -syou/me -gHolla infile
```

is the same as

```
mysed.py -gHello infile | mysed.py -syou/me | mysed.py -gHolla
```

i.e. the modifications and selections are performed sequentially on a per-line basis

CSC519
Programming Languages

Modules

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Programming in the Large

Problem: The "complexity" of an unstructured program grows roughly exponential with the number of lines of code

- Any line/construct can interact with any other line/construct

Result: Large programs become unmanageble very quickly

- Brooks: "Adding more programmers to a late project makes it later"
- Bell Labs MULTICS: Never achieved real usability status
- IBM OS/2 required complete reimplementation
- Windows NT always behind schedule
- Large German Bank spends about 30 million EUROS (projected cost 11 million)
  - then reverts to old system

What can a programming language offer to alleviate this problem?

Modules

Modules are a way of structuring large programs

- A program consists of a number of modules
- Each module has an interface and an implementation

Modules allow us to limit the interaction between different program parts

- Only the interface is visible, details about the implementation are (or should be) hidden
- A program part (e.g. another module) can selecticely import certain functionality offered by another module
- A module should not have any outside dependencies except via documented interfaces to other modules (or OS services)

Ideally, each module implements a certain concept or datatype

- . . . but Spaghetti modules are not unknown
- Good modular design is important and must precede actual coding!

Modules in Python

Python offers a simple but normally sufficient module system:

- Each file is a module
- All names visible in the global namespace are automatically exported
- Control over names is performed by the importing side (unusual!)

Importing:

- We can import a full module, making names available with the module name qualifier (`re.compile()`)
- We can import a limited set of names as in `from re import compile`
- We can import all names (but won't!)

Code in modules is actually executed if the module is imported for the first time

- Defines all functions
- Executes code not in functions, i.e. the module can be automatically initialized!

Example: Providing a Stack

```
# Module "stack"
# Names intended for export: Push(), Empty(), Pop(), StackEmpty

stack = []
StackEmpty = "StackEmpty"

def Push(element):
   stack.append(element)

def Empty():
   return len(stack)==0

def Pop():
    if(Empty()):
        raise StackEmpty
    return stack.pop()

print "Stack Module initialized"
```

Example Continued

```
>>> import stack
Stack Module initialized
>>> stack.Push(10)
>>> stack.Push("Hello")
>>> from stack import Pop
>>> Pop()
'Hello'
>>> Pop()
10
>>> Pop()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "stack.py", line 15, in Pop
    raise StackEmpty
StackEmpty
>>>
```

Discussion

Good Things:

- Stack module implements a single concept (a stack)
- Contains all functions and variables necessary
- User does not need to know how the stack is implemented

Bad Things:

- Implementation is only hidden by convention, user can access and mess with the `stack` and `StackEmpty` variables
- Only a single stack available
- No separate formal interface definition

Example: Providing a Stack Datatype

```
# Module "stack_dt"
# Names intended for export: Push(), Empty(), Pop(), StackEmpty

StackEmpty = "StackEmpty"

def NewStack():
   return []

def Push(stack, element):
   stack.append(element)

def Empty(stack):
   return len(stack)==0

def Pop(stack):
    if(Empty(stack)):
        raise StackEmpty
    return stack.pop()

print "Stack Module initialized, create stacks using NewStack"
```

Example Continued

```
>>> import stack_dt
Stack Module initialized, create stacks using NewStack
>>> mystack = stack_dt.NewStack()
>>> stack_dt.Push(mystack, 10)
>>> stack_dt.Push(mystack, Hello)
>>> stack_dt.Push(mystack, "Hello")
>>> stack_dt.Pop(mystack)
'Hello'
>>> stack_dt.Pop(mystack)
10
>>> stack_dt.Pop(mystack)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "stack_dt.py", line 17, in Pop
    raise StackEmpty
StackEmpty
```

Comparison: Modular Programming in C

C has no explicit module concept (it predates the widespread acceptance of modules as a good thing)

Nevertheless, C arguambly offers as good support for modular programming as Python!

- A module consist of a source file (the implementation) and the header file (describing the interface)
- Names not mentioned in the header file are not visible
- It is possible to control which names are accessible via the `static` keyword
- Importing a module is done by including the heaer file and linking the object file created from the source file (not elegant, but sufficient)

Weaknesses:

- Importing is all or nothing
- If static is not used, insulation between modules is weak
- If header files include header files, there is a proliferation of visible names
- No implicit intialization

Other Module Systems

Modula-2

- Interface describes exactly which names can be exported and what their signature is (variable, function with arguments of a specific type . . .)
- Importer can selectively import names from each module
- There can be multiple modules in a single file
- It's a real PITA ;-)

Bigloo Scheme

- Interface describes names and arity, but not types (Scheme is weakly typed)
- Modules have to explicitely list what they export
- You cant import both a whole module (i.e. all names exported by it) or single names

Exercises

Write a modular PL parser in Python (if possible with a partner)

- Specify an interface for the scanner (e.g. `NextToken()`, `CurrentToken()`, `Accept()`)
- Specify an interface for the Parser (`ParseProgram()`, `PrintProgram()`) and the token format (e.g. a tuple)

Implement two different scanners, both conforming to your interface definition

- One based on a source handler and NextChar() as the C version
- One using Python's regular expressions to extract the tokens (much easier, but probably slower!)

Note: If the Parser is to hard, just printing back a sequence of tokens will give you much of the same learning effect (but less satisfaction)!

CSC519
Programming Languages
Object Oriented Programming

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Object-Oriented Programming Languages

Modular programming: Keep data types and functions together

Object oriented programming: Keep data and functions together

State of the program is organized as a set of objects

- Objects have internal state
- Objects communicate by exchanging messages (that may change their internal state)

Ideal case: Objects in the program correspond to objects in the application domain

- Windowing system: Objects are scrollbars, menues, buttons. . .
- Airline ticketing system simulation: Objects are tellers, queues, passengers. . .

Important object-oriented languages: C++, Java, Simula, Smalltalk, Eiffel, Objective C

Objects

Objects contain functions and variables as elements

Common OO terminology:

- Elements of an objects are called members
- Functions are called member functions or methods
- Variables are called data members
- Calling a member function is is often called sending a message

Members may be public or private

- Public members can be accessed from anywhere in the program
- Private members can be only accessed by member functions

Note: Objects are a generalization of structure or records

- Structures are objects which contain only public data members

Classes

Objects are instances of a class

Classes are to objects as structures are to structure definitions (`typedefs`)

- A class definition describes what members the instances will have
- Class definition typically also describe how to create and initialize objects of that class

Classes can be used to add new datatypes to a programming language

- Enhances expressive power of the language
- Allows customization of language for an application domain

Classes in Python

Python implements classes as separate name spaces, using the class keyword

Syntax:

```
class NewClassName:
   <StatementBlock>
```

All statements in StatementBlock are executed in a new namespace and create local names

- Assignments create data members
- `def` functions create member functions

Note: In Python, all members are public!

Member Functions in Python

Member functions in Python are defined in the scope of a class definiton

- They have no special access to object members
- However, they are always called with a first argument that represents the object, and they can use this argument to manipulate object members

Example:

```
class SimpleString:
   str = "Default String"
   def QueryString(self):
      return self.str

   def SetString(self, arg):
      self.str = arg
```

Instances and Methods

Instances of a class are created by treating the class name as a function without arguments:

```
>>> a = SimpleString()
>>> print a
<__main__.SimpleString instance at 80b25d0>
```

Unless explicitely changed, instances share the values of class variables:

```
a.str
'Default String'
```

Methods in an instance are called as members, but without the initial argument (`self`), which is supplied by the Python system:

```
>>> a.SetString("New String")
>>> a.str
'New String'
>>> a.QueryString()
'New String'
```

Contructors and Destructors

Most often, we want to create an object in a well-defined initial state

We also may need to perform clean-up action if an object gets destroyed (e.g. close a file)

Constructors are functions that are automatically called when an object is created

- In Python, a method called `__init__(self)` acts as a constructor
- `__init__()` can also take additional arguments (which have to be added to the class creation call

Destructors are called when an object is destructed (note that this acutally only happens when the object is garbage collected!)

- The destructor is called `__del__(self)`

Note: Constructors and (especially) destructors are a lot more important in languages with explicit memory handling (e.g. C++)!

Example: Stacks as Objects

```
class Stack:
   stack = []
   StackEmpty = "StackEmpty"

   def __init__(self):
      self.stack = []

   def __del__(self):
      if not self.Empty():
         print "Warning: Non-Empty stack destroyed"

   def Push(self,element):
      self.stack.append(element)

   def Empty(self):
      return len(self.stack)==0

   def Pop(self):
      if(self.Empty()):
         raise Stack.StackEmpty
      return self.stack.pop()
```

Example Continued

```
>>> a = Stack()
>>> b = Stack()
>>> a.Push(10)
>>> b.Push(20)
>>> a.stack
[10]
>>> b.stack
[20]
>>> a.Push(11)
>>> a.Pop()
11
>>> a.Pop()
10
>>> try:
...   a.Pop()
... except Stack.StackEmpty:
...    print "Caught Exception"
...
Caught Exception
>>> a = 10 # a will be garbage-collected, but is empty
>>> b = 10 # b will be garbage-collected
Warning: Non-Empty stack destroyed
```

Functions and Optional Arguments

Functions and methods in Python can have optional arguments

- They can be called with a variable number of argumens
- If an argument is missing, a default value will be provided

Syntax:

```
def f_name(arg1, arg2, opt_arg1=default1, opt_arg2=default2):
    <Function Body>
```

This is particularly useful for constructors

- Often you want to convert some other data type to the new class
- Sometimes you just want to create a defined class instance

Example: Complex Numbers (First Try)

```
import math

class MyComplex:
    real_val = 0
    imag_val = 0

    def __init__(self, re=0, im=0):
        self.real_val = re
        self.imag_val = im

    def display(self):
        print "(",self.real_val,",",self.imag_val,")"

    def abs(self):
        return math.sqrt(self.real_val*self.real_val+self.imag_val*self.imag_val)

    def real():
        return self.real_val

    def imag():
        return self.imag_val
```

Example (Continued)

```
>>> c=MyComplex()  # Just create a default complex number
>>> c.display()
( 0 , 0 )
>>> c1 = MyComplex(10) # Convert an integer to complex
>>> c1.display()
( 10 , 0 )
>>> c2 = MyComplex(math.pi, 42) # Create a complex given real and
>>>                              # imaginary parts
>>> c2.display()
( 3.14159265359 , 42 )
>>> c2.abs()
42.1173314017
```

Exercises

Add methods to the `MyComplex` class to implement e.g.

- Complex additon and subtraction
- Rotation by an angle ϕ (if the number is seen as a vector in two-dimensional space)
- Complex multiplication

Create a text canvas class

- That class maintains a (X time Y) matrix of characters (make X and Y optional arguments to the constructor)
- It allows the user to set any character in the matrix to any value
- Upon request, it will print the matrix directly
- If you have spare energy, add functions that will draw a line for character from (x_1, y_1) to (x_2, y_2)

CSC519
Programming Languages

Object Oriented Programming – Inheritance

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Taxonomy and Classification

Normally, we do not perceive all properties of an object at onece - our world would be much to complex

Instead, we classify things, and silently assume that most things about a given class are the same

– Example: Students are human, humans have two legs – by default, students have two legs

This gives us a is-a hierarchy:

– A student is a human
– A human is a living thing
– A living thing is a thing

Hierarchical Classification

Some properties and mechanisms are shared by all things:

– Mass and gravitational attraction

Others are shared by living things:

– Eats, drinks, metabolizes, reproduces

Some are shared by all (or most) humans:

– Number of legs
– Gender

Others are shared by students:

– Are ~~partying~~ working hard
– Have student id

Why is this useful?

We can reason at an adequate level of abstraction:

- If we design a bridge, we do not need to know the religion of somebody passing it
- We may need to know his weight, though!

When specializing a concept, we do not loose all accumulate knowledge, but rather extend it:

- All warm-bloded animals breathe
- Birds are warm-bloded animals
- All birds have feathers and can fly
- Penguins are birds but cannot fly

Inheritance in Object-Oriented Languages

Inheritance allows us to model is-a hierarchies in our program design

- Data types (classes) form a hierarchy and can be derived from other data types
- Allows the use of members (including methods) of the base class in the derived class
- Derived class can extend base class by adding new members
- Derived class can override members

Important: Because of the is-a property, we can use members of a derived class whenever code expects a member of the base class

Advantages:

- Promotes code reuse (we do not need to reimplement methods of the base class)
- Allows general routines to handle specialized objects (an array of humans can store students...)

Single Inheritance in Python

A class can inherit another class by listing it in the class definiton:

```
class newclass(baseclass):
   <proceed as usual>
```

Members of the base class are accessible in the derived class (if not overwritten)

Example

```
#! /usr/bin/env python
import string

class Shape:
    NoDrawMethod = "Don't know how to draw generic shape"
    object_count = 0;
    colour = "Y"

    def __init__(self,colour="X"):
        self.colour = colour
        Shape.object_count = Shape.object_count+1

    def draw(self):
        raise NoDrawMethod

    def set_colour(self, colour):
        self.colour = colour

    def get_count(self):
        return Shape.object_count

    def __del__(self):
        print "Destroying:",self
```

Example (2)

```
class Rectangle(Shape):
    sizex = 1;
    sizey = 1;

    def __init__(self,sizex=1,sizey=1, colour="X"):
        Shape.__init__(self, colour)
        self.sizex = sizex
        self.sizey = sizey

    def draw(self):
        for i in range(0,self.sizey):
            print string.join(map(lambda x,y=self.colour:y, range(0,self.sizex)),"")

class Square(Rectangle):
       def __init__(self,size=1, colour="X"):
           Rectangle.__init__(self,size,size,colour)
```

Example (3)

```
class Triangle(Shape):
    size = 1;

    def __init__(self,size=1, colour="X"):
        Shape.__init__(self, colour)
        self.size = size

    def draw(self):
        for i in range(1,self.size+1):
            print string.join(map(lambda x,y=self.colour:y, range(0,i)),"")

a = Rectangle(8,2)
a.draw()
b=Triangle(3)
b.draw()
c= Square(2,"A")
c.draw()
print a.get_count()
```

Example Output

```
XXXXXXXX
XXXXXXXX
X
XX
XXX
AA
AA
3
Destroying: <__main__.Triangle instance at 80e44c8>
Destroying: <__main__.Square instance at 80e4500>
Destroying: <__main__.Rectangle instance at 80e39b8>
```

Another Example

```
#! /usr/bin/env python

class Base:
    def method1(self):
        print "Base Class 1"
        self.method2()

    def method2(self):
        print "Base Class 2"


class Derived(Base):
    def method1(self):
        print "Base Class 2"
        Base.method1(self)

    def method2(self):
        print "Derived Class 2"

a = Derived()
a.method1()
```

Example Output

```
Base Class 2
Base Class 1
Derived Class 2
```

Name Resolution and Virtual Functions

Member references in a class are handled as follows:

- First, the name is searched for in the most specific class of an object
- Then it is searched for in the class it is directly derived from
- Then we search one level deeper. . .

Important: In Python, this search starts over for each member reference

- In OO language, all Python member functions are virtual
- A base class member function can call a function in a derived class

To make certain that we call a function in the base class, we explicitely need to specify it:

```
Base.method1(self)
```

Exercises

Design data types for a traffic simulation, with

- Passengers (that want to go someplace)
- Cars (that go to where the majority of passengers want to go)
- Ferries, that take a limited amount of weight someplace fixed

Think about the weight of passengers and cars (base weight + passenger weight)

Think about getting the destination of any object - for passengers and ferries it is fixed, but for cars it depends on the passengers

CSC519
Programming Languages

Multiple Inheritance
Operator Overloading

Stephan Schulz

Department of Computer Science

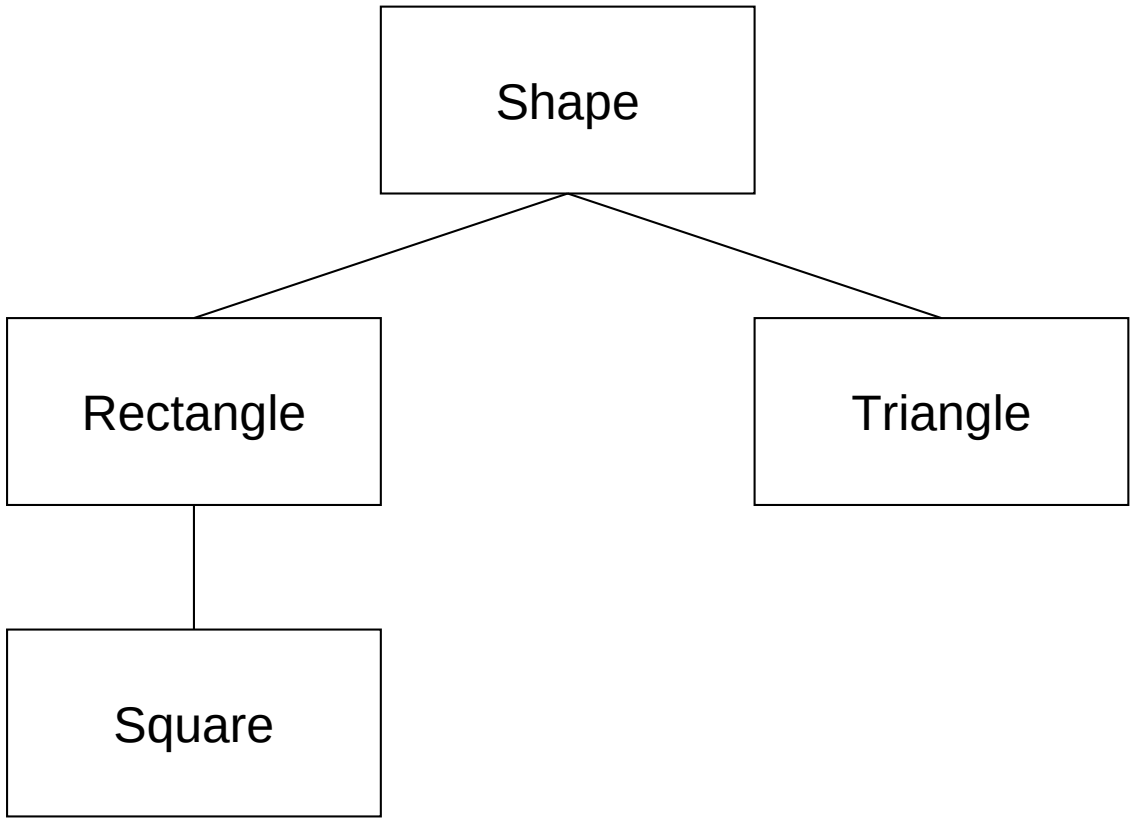
University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Single Inheritance Generates a Tree



Each class is derived from a single superclass

Each class may have multiple subclasses

What happens if we allow an arbitrary inheritance graph?

Multiple Inheritance

Java and Smalltalk only allow single inheritance

- Reflects the taxonomic thinking and the is-a hierarchy
- Derived classes always represent a subset of objects of a base class

Both C++ and Python allow a class to have multiple superclasses!

- Class inherits members from all superclasses

Advantages:

- Allows combination of features from different classe

Disadvantages:

- May lead to unexpected interaction between classes
- Spaghetti inheritance makes it very hard to understand the code
- May create a lot of interdependency, thus breaking the modularization concept

Multiple Inheritance in Python

To inherit a class from multiple super classes, just name them in the class declaration:

```
class Newclass(baseclass1,baseclass2, baseclass3):
   <Business as usual>
```

Name (member) resolution is done depth first, left to right:

- First, names defined in the derived class are accessed
- If a name is not defined in the derived class, it is searched for in the first base class (and it's superclasses)
- If it is not found there, search proceeds in the second base class. . .

Note: Python does not automatically call constructors and destructors of superclasses!

- If necessary, add explicit calls in `__init__()`

Example (1)

```
class BinTreeCell:
    DuplicateIdentity = "Cannot have multiple entries with the same key!"
    NotFound = "Key not found!"

    def __init__(self, newkey):
        self.key = newkey
        self.lson = None
        self.rson = None

    def Insert(self, new):
        if new.key < self.key:
            if self.lson:
                self.lson.Insert(new)
            else:
                self.lson = new
        elif new.key > self.key:
            if self.rson:
                self.rson.Insert(new)
            else:
                self.rson = new
        else:
            raise BinTreeCell.DuplicateIdentity
```

Example (2)

```
def Find(self, key):
    if key == self.key:
        return self
    if key < self.key:
        if self.lson:
            return self.lson.Find(key)
    elif key > self.key:
        if self.rson:
            return self.rson.Find(key)
    raise BinTreeCell.NotFound

def Print(self):
    if self.lson != None:
        self.lson.Print()
    print "TreeNode:", self.key
    if self.rson != None:
        self.rson.Print()
```

Example (3)

```
class BinTree:
    def __init__(self):
        self.anchor = None
        print "BinTree Initialized"

    def Insert(self, new):
        if self.anchor:
            self.anchor.Insert(new)
        else:
            self.anchor = new

    def InsertKey(self, newkey):
        new = BinTreeCell(newkey)
        self.Insert(new)

    def Find(self, key):
        if self.anchor:
            return self.anchor.Find(key)
        else:
            raise BinTreeCell.NotFound
```

Example (4)

```
def Print(self):
    if self.anchor:
        self.anchor.Print()
    else:
        print "<empty>"
```

Example (5)

```
 class Person:
    name = ""
    job = ""

    def __init__(self,name, job):
        self.name = name
        self.job = job

    def Print(self):
        print "Person:", self.name,":",self.job


class PersonInTree(BinTreeCell,Person):
    def __init__(self, name, job):
        BinTreeCell.__init__(self,name)
        Person.__init__(self,name, job)
```

Example (6)

```
a = BinTree()

me = PersonInTree("Stephan", "Scientist")
you = PersonInTree("George W.", "President")
a.Insert(me)
a.Insert(you)

print "Tree in order:"
a.Print()

print "Find Stephan"
a.Find("Stephan").Print()
print "Find Stephan and print just him"
Person.Print(a.Find("Stephan"))
```

Example (Output)

```
BinTree Initialized
Tree in order:
TreeNode: George W.
TreeNode: Stephan
Find Stephan
TreeNode: George W.
TreeNode: Stephan
Find Stephan and print just him
Person: Stephan : Scientist
Trey to find Leonardo
Traceback (innermost last):
  File "./persons.py", line 105, in ?
    Person.Print(a.Find("Leonardo"))
  File "./persons.py", line 62, in Find
    return self.anchor.Find(key)
  File "./persons.py", line 31, in Find
    return self.lson.Find(key)
  File "./persons.py", line 35, in Find
    raise BinTreeCell.NotFound
Key not found!
```

Objects as First Class Datatypes

A First Class datatype is a datatype with full language support

In particular, they enjoy full operator support (when applicable)

Python examples:

- Assignment with `=`
- Comparison with `<, >.  ==`
- Arithmetic operations with `+,-,*,/` (where applicable)
- Nice output with `print`

Can we give the same comfort level to user-defined data type?

Operator Overloading

Operator overloading is the name of a technique to associate new functions (for new datatypes) to the standard operators of a language

– Normally does not change behaviour on existing data types

C++: Globally define special functions (e.g. `op+()`) with the desired signature

– Compiler will figure out wich one to use by looking at the type of the arguments
– You can also define e.g. addition for two C style strings (i.e. for non-class data types)

Python has no static type system, so overloading works differently:

– We can only define the behaviour for operators if at least one of the arguments is a class instance (i.e. a user-defined object)
– Overloading is done by adding special functions to the object

Note: Operator overloading sometimes is elegant, but sometimes makes pograms hard to understand and (as a consequence) inefficient

Support for Some General Operators

Support for printing:

- A class can define a method `__str__(self)` that should return a string
- If `print` is called on that object, the string returned by that function will be printed

Support for comparisons:

- Define a function `__cmp__(self,other)`
- It should return `-1` if `self` is smaller than `other`, `+1` if `self` is bigger than other, and `0` if both are the same
- This function will be used to determine the result of `>`, `>=`, `<`, `<=`, `==` comparisons
- Note: Newe Python versions (2.1 and up) also allow a more detailed specification for each individual operator

Support for Numerical Operators

You can overload all standard operators using the following names:

`__add__(self,other)`	Addition, `self + other`
`__sub__(self,other)`	Subtraction, `self - other`
`__mul__(self,other)`	Multiplication, `self * other`
`__div__(self,other)`	Divison, `self / other`
`__abs__(self)`	Absolute value, `abs(self)`
`__neg__(self)`	Negation,`-self`

Note: What happens if an operator is called as `10*obj`, where `obj` is our object, but 10 is a normal number (which does not know how to multiply itself to out object)?

Answer: If the first object does not define a proper method, the second one may provide a swapped argument version

`__radd__(self,other)`	Addition, `other + self`
`__rsub__(self,other)`	Subtraction, `other - self`
`__rmul__(self,other)`	Multiplication, `other * self`
`__rdiv__(self,other)`	Divison, `other / self`

Example: Vector Arithmetic

```
#!/usr/bin/env python
import math
import string

class Vector:
    value = []

    def __init__(self, elements):
        self.value = elements

    def __add__(self, other):
        return Vector(map(lambda x,y:x+y, self.value,other.value))

    def __mul__(self,other):
        if type(other)==type(self):
            return self.crossproduct(other)
        return Vector(map(lambda x,y=other:x*y , self.value))

    def __rmul__(self,other):
        return self.__mul__(other)
```

Example (Continued)

```
    def crossproduct(self, other):
        return reduce(lambda x,y:x+y,
                      map(lambda x,y:x*y, self.value,other.value))

    def __str__(self):
        return "("+string.join(map(lambda x:str(x) , self.value),",")+")"

    def __abs__(self):
        return math.sqrt(reduce(lambda x,y:x+y,
                                map(lambda x:x*x,self.value)))

a = Vector([1,2,0])
b = Vector([1,2,3])

print a
print a+b
print a*b
print a*10
print 10*a
print abs(a)
```

Example (Output)

```
(1,2,0)
(2,4,3)
5
(10,20,0)
(10,20,0)
2.2360679775
```

Exercises

Extend the vector library into a linear algebra library by adding:

- Additional vector operations
- A Matrix class with matrix multiplication and matrix/vector multiplication
- Of course: Use operator overloading for the obvious operators!

CSC519
Programming Languages
Functional Programming and Scheme

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Functional Programming

Characterized by the evaluation of expressions as opposed to the execution of statements

Pure form:

- No assignments and variables (but we can give names to values)!
- No side effects (I/O)!
- Value of an expression depends only on its arguments
- No loops (just recursion)

Functions are first-order values (purest form: Everything is a function)

- Functions can be created at run time, passed as arguments, be applied to values. . .

Notice: Most functioal programming is impure to a certain degree

Scheme and LISP

LISP is one of the oldest high-level languages around (1958)

- Created by accident while playing with the λ-Calculus
- Name: List Processing language
- Mutated over time, currently popular:
  - Common LISP (ANSI-Standard, heavy lifter, huge libraries)
  - Scheme (R^{n}RS Standards (current n: 5), small, elegant, well-defined)

Scheme (and LISP) are impure functional programming languages:

- Strongly support functional programming style
- Also have global state that can be manipulated

Originally, LISP was only interpreted

- Today, you can also get very good compilers for LISP
- For development and learning, the interpreter often is preferred

Scheme in the Lab

Our Scheme implementation is `umb-scheme`, a nice interpreter based on R4RS ("Revised^4 Report on the Algorithmic Language Scheme)

- Type `umb-scheme` to start the interpreter
- You can now enter expressions and have them evaluated
- Type `^D` (Control-D) to exit from the interpreter
- If you have a scheme program in a file, `umb-scheme <filename>` will load and evaluate that program before giving you the prompt
- The only way I have found to make stand-alone programs with`umb-scheme` is to use a shell script wrapper:

```
#!/bin/sh
umb-scheme < test.scm
```

Command line editing is not as good as for Python - I recommend writing larger functions in Emacs (or another good text editor (hah!)) and copy them into the interpreter with the mouse

Scheme Basics

Every object is either an atom or a list

Atoms may have predefined values:

- Numbers evaluate to their numerical value
- Strings evaluate to the character sequence
- Some symbols predefined by the system

A Scheme program is a sequence of S-expressions, every S-expression either is atomic or composite

```
(op e1 e2 ... en)
```

- `op` is an expression that should evaluate to a function (often a predefined one, e.g. `+`, `-`, `*`, `display`)
- `e1`. . . `en` are subexpressions
- All expressions are evaluated, then the result of evaluating `op` is applied to the results of evaluating the arguments

Example Session

```
==> 10

10
==> (+ 10 20)

30

==> display

(lambda (obj . port)
  (#_display obj (if (null? port)
                     (current-output-port)
                     (car port))) )
==> (display (+ 10 20))
30
()
```

Exercises

Start the Scheme interpreter and evaluate some arithmetic expressions

CSC519
Programming Languages

Scheme Introduction

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Scheme Datatypes

Atoms are atomic (elementary) symbols

- Examples: `+`, `string->symbol`, 12
- Some types of atoms have intrinsic values:
  - Numbers: `1`, `3`, `3.14`
  - Strings: `"Hello"`, `"FooBar"`
- Other atoms can be bound to values
- Several atoms are prebound by the scheme environment:
  - `+` is bound to a function that adds numbers
  - `car` is bound to a function that returns the first element of a list
  - `if` is bound to a special form that implements a conditional

Lists are the major program and data structures

- List syntax: Enclosed in parentheses, no commata
- Examples: `()` (empty list), `(+ 2 42)`, `(() 10 (1 2 3))`

Functions accept arguments and return values

Program Structure

A Scheme-Program is a sequence of S-expressions

- Those expressions are evaluated sequentially

Each S-expression has a value and may have side effects

Side effects include:

- Binding a value to an atom (e.g. naming a new function)
- Input or output
- Destructively changing the global state ("assignment")

S-expression syntax:

- An atom is a S-expression
- A list is a S-expression, where:
  - The first element has to evaluate to a function
  - The other arguments may be S-expressions themselves
  - The value of the expression is computed by computing the values of the elements and then applying the first arguments to the other ones

Examples

```
==> 10

10
==> (+ 10 20)

30

==> +

< primitive: + Number... >
==> (+ 10 (* 10 20 30) 4)

6014
==> "Hallo"

"Hallo"
==> (display "Hallo")
Hallo
()
```

Definitions and Quotation

To define a name for a value, we use `(define name value)`

```
(define pi 3.14159265359)
   pi
(define e  2.71828182846)
   e
(define add +)
   add
(add pi e)
   5.85987448205
```

– We can define names for ordinary values and functions

Atoms and S-expressions can be values

– By default, they are evaluated immedeately
– We can quote them to stop evaluation:

```
==> a
Error: ‘a’ is undefined.
==> (quote a)
a
==> ’a
a
```

Simple I/O

Note: There is no strong emphasis on I/O in functional programming!

– We try to avoid side effects as much as possible

Output:

– `(display <arg>)` will print a representation of `<arg>`
– `(newline)` will write a newline
– Optional argument to both: Output port

Input:

– `(read)` will read a scheme object (atom or S-expression) and return it
– Optional argument: Input port (default: Read from terminal)

Defining New Functions

New functions can be defined with `define`:

```
(define (fun arg1 arg2 ... argn)
   <s-expr1>
   <s-expr2>
   ...
   <s-exprm>)
```

- Defines a function `fun`
- Formal arguments `arg1`. . . `argn` are available as defined names in the body
- The body is a sequence of S-expressions
- The value of the last S-expression is returned

Note: Functions can be recursive (and normally are)

Examples

```
(define (poly x)
   (+ (* x x)  (* 5 x) 10))

(poly 3)
   34

(poly 10)
   160

(poly (* 7 10))
   5260
```

Examples

```
(define (one)
    (display "One")
       1)

(define (factorial i)
   (if (= i 0)
       1
       (* i (factorial (- i 1)))))

(poly (one))
   One
   16
one
   (lambda ()
     (display "One")
     1)
(one)
   One
   1
(factorial 10)
   3628800
```

Exercises (Assignment on Wednesday)

Write a function that computes the Fibonacci-Numbers (boring, I know...)

Write a function that computes the Stirling numbers of the first kind

Write a function that computes and prints the Binomial Coefficients up to a maximum

CSC519
Programming Languages

Scheme
Arithmetic and Lists

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Anonymous Functions: `lambda`

LISP and Scheme are build around a theoretical foundation called the λ-calculus

- Very simple calculus
- Turing-complete

λ-expressions can be used to define functional objects without name

- Syntax:

  `(lambda (<arg1> ... <argn>) <s-expr1> ... <s-expn>)`

  * This creates a function with formal arguments `<arg1> ... <argn>`
  * The value returned is that of the last expression in the body

Note that e.g. `(define (fun x) (* 2 x))` is only shorthand for `(define fun (lambda (x) (* 2 x)))`

Examples

```
==> (lambda () 1)
(lambda ()
  1 )

==> ((lambda () 1))
1

==> (define poly
            (lambda (x) (+ (* x x)  (* 5 x) 10)))
poly

==> (poly 5)
60

==> ((lambda (x y) (+ x y)) 10 20)
30
```

The Special Form `if`

`if` is used for conditional evaluation

- It takes either 2 or three arguments
- If the first argument evaluates to true, the second one is evaluated, and its value is returned
- If it evaluates to false, the third argument (if any) is evaluated, and its value returned
- If first argument is false and third argument not present, the result is unspecified

`if` is a special form or a syntax

- Not all its arguments are evaluated (they are for S-expressions):

```
(define (my-if test val1 val2)
   (if test val1 val2))
(my-if (= 10 10) (display "Branch1") (display "Branch2"))
   Branch1Branch2
(if (= 10 10) (display "Branch1") (display "Branch2"))
   Branch1
```

Basic Arithmetic

Scheme supports a hierarchy of numeric types with corresponding type predicated:

- `(number? <arg>)` returns `#t` if `<arg>` is a number
- `(complex? <arg>)` returns `#t` if `<arg>` is a complex number
- `(real? <arg>)` returns `#t` if `<arg>` is a real (machine) number
- `(rational? <arg>)` returns `#t` if `<arg>` is a rational number (i.e. expressable as a fraction)
- `(integer? <arg>)` returns `#t` if `<arg>` is an integer number. Scheme (normally) supports bignums and transparently converts to them if necessary

Many arithmetic functions are supported for all numeric types:

- `=` returns `#t`, if all arguments are the same
- `+`, `-`, `*`, `/` implement normal addition, subtraction, multiplication and division
- All operators support 2 or more arguments and are left-associative

Note on truth values:

- `#f` is explicit false
- Everything else is true, `#t` is the explicit truth value for true

Example

```
(define (rollercoaster n count)
   (display "At: ") (display n)
   (newline)
   (if (= n 1)
       count
       (if (= (remainder n 2) 0)
           (rollercoaster (/ n 2) (+ count 1))
           (rollercoaster (+ (* 3 n) 1) (+ count 1)))))

(rollercoaster 3 0)
At: 3
At: 10
At: 5
At: 16
At: 8
At: 4
At: 2
At: 1
7
```

More Arithmetic

Comparisons:

- `(= x1 x2 ...)`: `#t`, if all arguments are equal
- `(< x1 x2 ...)`: `#t`, if arguments are increasing monotonically
- `(> x1 x2 ...)`: `#t`, if arguments are decreasing monotonically
- `(<= x1 x2 ...)`: `#t`, if no argument is bigger than its successor
- `(>= x1 x2 ...)`: `#t`, if no argument is smaller than its successor

Predicates:

- `(zero? x)`
- `(positive? x)`
- `(negative? x)`
- `(odd? x)`
- `(even? x)`

Conversion

- `(string->number string)`
- `(number->string x)`

Lists and `cons`-Pairs

Lists are defined recursively:

- The empty list (written as `()`) is a list
- If `l` is a list, then `(cons x l)` is a list

Note: `cons` is a function that takes two elements and returns a cons pair

- The first element of the cons-pair is called the car of the pair
- The second element is called the cdr of the pair

We can also create lists with the `list` function

- It returns the list containing its arguments

Note that we now have three different ways of writing the same list:

```
- '(1 2 3)
- (list 1 2 3)
- (cons 1 (cons 2 (cons 3 '())))
```

Taking Lists Apart

Given any cons pair `p`:

- `(car p)` returns the car
- `(cdr p)` returns the cdr

For non-empty lists:

- `car` returns the first element
- `cdr` returns the rest of the list

Often, we want to get e.g. the second element of a list

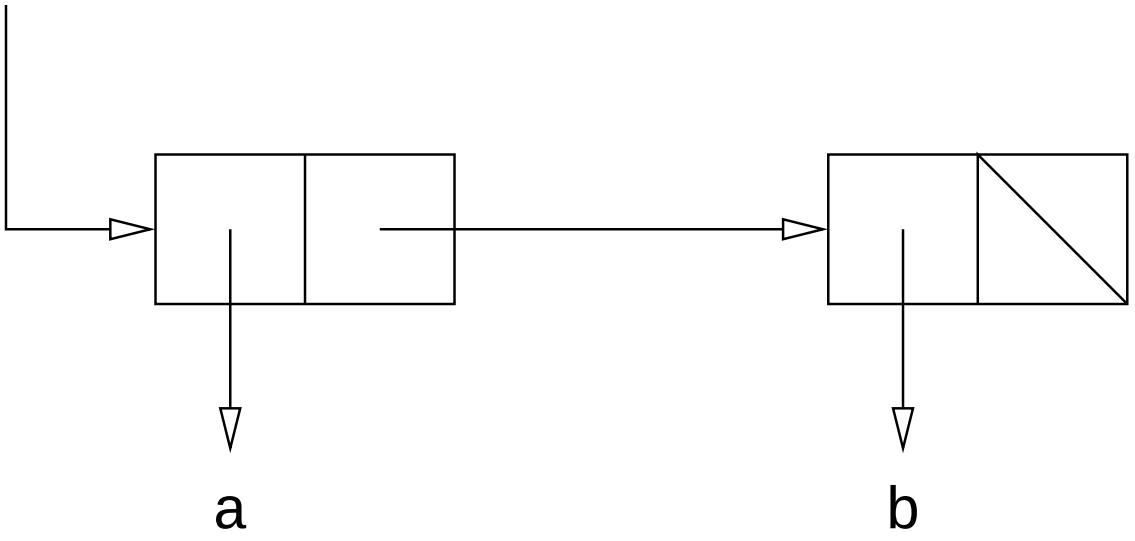
- `(car (cdr l))`
- Alternative: `(cadr l)`
- Up to 4 `a` and `d` can be stacked up...

Note: We can destructively change the members of a cons pair:

- `(set-car! p new)` changes the first element
- `(set-cdr! p new)` changes the second element

List Structure

Lists are actually simply linked lists of cons cells: `(list 'a 'b)` produces



This structure can surprise us if we use destructive programming

- Lists can share substructures
- Changing one list may (unexpectedly) change others

More List Functions

Predicates

- `(list? obj)` returns `#t` if `obj` is a list
- `(pair? obj)` returns `#t` if `obj` is a cons pair (every non-empty list is a cons pair, but not every cons pair is a list)
- `(null? l)` returns `#t`, if `l` is the empty list

Some list functions

- `(length l)`: Number of elements in `l`
- `(append l ...)`: Appends all lists. Notice that this reuses the last argument list for building the new one
- `(list-ref l k`: Return the `k`th element of `l`

Examples

```
(define (revert l)
   (if (null? l)
       l
       (append (revert (cdr l)) (list (car l)))))

==> (revert '(1 2 3 4))

(4 3 2 1)

(define (invert l)
   (if (null? l)
       l
       (cons (/ 1 (car l))
             (invert (cdr l)))))

==> (invert '(4 3 2 1))

(1/4 1/3 1/2 1)
```

Concepts of Equality (I)

Scheme supports various different equality tests!

Comparing addresses: `eq?`

- (Generally): Returns true if the two objects have the same address in memory (for symbols, that is true if the are spelled the samy way)
- Very efficient!
- Well-defined for non-empty lists, cons-pairs, functions, symbols. . .
- Implementation-defined behaviour for numbers, strings. . .

Doing the right thing: `eqv?`

- Intention: Be generally useful, but still efficient
- Everything that is `eq?` is also `eqv?`
- Compares numbers, characters and the empty list by value, not by location

Concepts of Equality (I)

Recursive descent: `equal?`

- `equal?` recursively decomposes structures into its elements and compares them on the atom level
- Everything that is `eqv?` is also `equal?`
- It generally returns `#t`, if the two objects would print the same way

Specialized equality checks:

- `string=?` compares strings by value
- `=` compares numbers by value
- `char=?` compare characters by value

Assignment

Write a function that unscrambles a simple puzzle (or set of functions)

- Input is a list of 9 elements, the numbers 1-8 in any order and the atom `E`
- It is interpreted as a scrambled 3×3 matrix:

```
(1 E 3 4 2 6 7 5 8) ==>  1 E 3
                         4 2 6
                         7 5 8
```

- A legal move exchanges the E with one of it neighbours. We denote this by one of the atoms `'u`, `'d`, `'l`, `'r`, depending on wether the digit has to slide up, `down`, `left`, `right` to take the place of the `E`

Your program should find and print a sequence of moves that transforms the scrambled puzzle to an unscrambled one:

Assignment (2)

```
Start:    (1 E 3 4 2 6 7 5 8)        1 E 3
                                     4 2 6
                                     7 5 8

Up 'u:    (1 2 3 4 E 6 7 5 8)        1 2 3
                                     4 E 6
                                     7 5 8

Up 'u:    (1 2 3 4 5 6 7 E 8)        1 2 3
                                     4 5 6
                                     7 E 8

Left 'l:  (1 2 3 4 5 6 7 E 8)        1 2 3
                                     4 5 6
                                     7 8 E

Output: (u u l)
```

Assignment (3)

Hints:

- Write a recursive function
- Arguments are two lists of tuples, one for processed states, one for unprocessed states
  - Each tuple contains a configuration (list of 9 elements)
  - It also contains a sequence of moves to get there
- Initially, the first list is `(((init-config) ()))` and the second one is empty
- At each step:
  - Get the first element from unprocessed list
  - If it is ordered, you are done. Otherwise
  - Add it to the processed list
  - Generate the 2, 3, or 4 possible successors
  - For each successor: If the configuration is not yet on one of the lists, add it to unprocessed
  - Otherwise, ignore it

Bonus: Write a function that accepts any size of square!

CSC519
Programming Languages

Scheme
Interesting Stuff

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Introducing Temporary Names

Often, we want to assign temporary names to values

- Avoids costly recomputations
- May make code more transparent

In Scheme, we use the `let` construct:

```
(let ((var1 val1)
      (var2 val2)
      ...       )
  expr1
  ....)
```

- First, the `vals` are evaluated (in an arbitrary order)
- Then the new names are created and the values bound to them
- Finally, the expressions are evaluated with those bindings in place
- The value of the `let` is the value of the last expression

Variants: `let*` and `letrec`

Syntax: Exactly as for `let`

Semantics of `let*`:

- The first value is computed
- The first binding is performed
- The second value is computed (with the first variable already defined)
- The second binding is performed
- ...

Semantics of `letrec`:

- First, all the variables are created (with unspecified values)
- Then the values are computed (and can reference, but not evaluate the variables)
- Then the bindings are performed

In both cases the expressions in the body are evaluated sequentially in the final environment, and the value of the last expression is returned

Examples for `let` and `let*`

```
(define a 10)
(define b 20)

(let ((a (+ a b))
      (c a))
  (display a)(newline)
  (display c)(newline)
  (+ a c))
30
10

40

(let* ((a (+ a b))
       (c a))
  (display a)(newline)
  (display c)(newline)
  (+ a c))
30
30

60
```

A Non-Trivial Example

```
(define (split l l1 l2)
  (if (null? l)
      (list l1 l2)
      (if (null? (cdr l))
          (list (cons (car l) l1) l2)
          (split (cddr l)
                 (cons (car l) l1)
                 (cons (cadr l) l2)))))

(define (merge l1 l2)
  (if (null? l1)
      l2
      (if (null? l2)
          l1
          (let ((e1 (car l1))
                (e2 (car l2)))
            (if (< e1 e2)
                (cons e1 (merge (cdr l1) l2))
                (cons e2 (merge l1 (cdr l2))))))))
```

Example (2)

```
(define (sort l)
  (if (<= (length l) 1)
      l
      (let ((s (split l '() '())))
        (merge (sort (car s))
               (sort (cadr s))))))


==> (sort '(1 3 2 5 4 6 8 7 9))
(1 2 3 4 5 6 7 8 9)

==> (sort '(21 2 34 3 90 93 28 20 23 4 2 1 4 5 6 42 11))
(1 2 2 3 4 4 5 6 11 20 21 23 28 34 42 90 93)
```

Example for `letrec`

```
(letrec ((fak (lambda (x)
                (if (= x 0)
                    1
                    (* x (fak (- x 1)))))))
   (fak 10))
```

```
3628800
```

Compare:

```
(let ((fak (lambda (x)
             (if (= x 0)
                 1
                 (* x (fak (- x 1)))))))
   (fak 10))
```

```
Error: ‘fak’ is undefined.
```

Code from Carnivore

Acutally, today hackers hacked into the NSA computers, stole the code from project Carnivore, and posted it on the Internet

Surprise! It's written in Scheme!

Code from Carnivore

Acutally, today hackers hacked into the NSA computers, stole the code from project Carnivore, and posted it on the Internet

Surprise! It's written in Scheme!

Proof: Here are the last couple of lines. . .

```
)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

Optional Arguments and `apply`

Functions like +, >, `list` can take different numbers of arguments

We can do the same for user-defined functions:

```
(define (fun req1 req2 . optional)
   <body>)
```

- The required arguments are explicitly listed and bound as usual
- All the extra arguments are packed into a list and bound to the name after the dot
- Note special case of only optional arguments:
  `(define (fun . r) <body>)`
- Also: `(lambda ( . r) <body)` has just optional arguments

We also have a way of applying a function to an arbitrary length list of arguments:

```
(apply fun l)
```

- If `l` equals `(e1 e2 e3)`, the result is the same as `(fun e1 e2 e3 ...)`

`map` and `for-each`

Syntax: `(map f l1 ...)`

- The first argument is a function that takes n argument
- There have to be n additional arguments, all lists of the same length
- `f` is applied elementwise to the elements of the list
- The value of the `map` expression is the list of results
- The order in which the computations are performed is undefinded
- Example:

```
==> (map + '(1 2 3 4) '(10 20 30 40))
(11 22 33 44)
```

`foreach` has the same syntax

- Difference: Computations are performed left-to-right
- The value of a `foreach` expression is unspecified

Remark: Most often, both are applied to just a single list

Functional Definition for `map`

```
(define (simple-map f l)
  (if (null? l)
      '()
      (cons (f (car l)) (simple-map f (cdr l)))))


(define (full-map f req . opt)
  (if (null? req)
      '()
      (let ((r (cons req opt)))
        (cons (apply f (simple-map car r))
              (apply full-map (cons f (simple-map cdr r)))))))


==> (full-map + '(1 2 3 4) '(10 20 30 40))
(11 22 33 44)

==> (full-map (lambda (x) (* x x x)) '(1 2 3 4 5 6 7 8 9))
(1 8 27 64 125 216 343 512 729)
```

Exercises

Write a function `compose` that takes two functions `f` and `g` as arguments and returns the composition `f`$\circ$`g`

- Definition: $(\texttt{f} \circ \texttt{g})(\texttt{x}) = \texttt{f}(\texttt{g}(\texttt{x}))$ for all `x`
- Keep in mind that `x` can be a vector (i.e. `g` may take more than one argument)

Write a function `curry` that takes a function of n arguments and turns it into a function that accepts a single argument and returns a function of $n-1$ arguments

- `f:(x1, x2, x3) -> y`, but `(curry f):x1 -> (x2,x3)->y`
- Hint: It's a 4-liner with two `lambdas`

Write generic sort function that accepts a `>` parameter (a function) and a list, and sorts the list accordingly. Use `curry` to turn it into a number of different sort functions taking just the list.

CSC519
Programming Languages

Excursion: Turing Machines

Stephan Schulz

Department of Computer Science

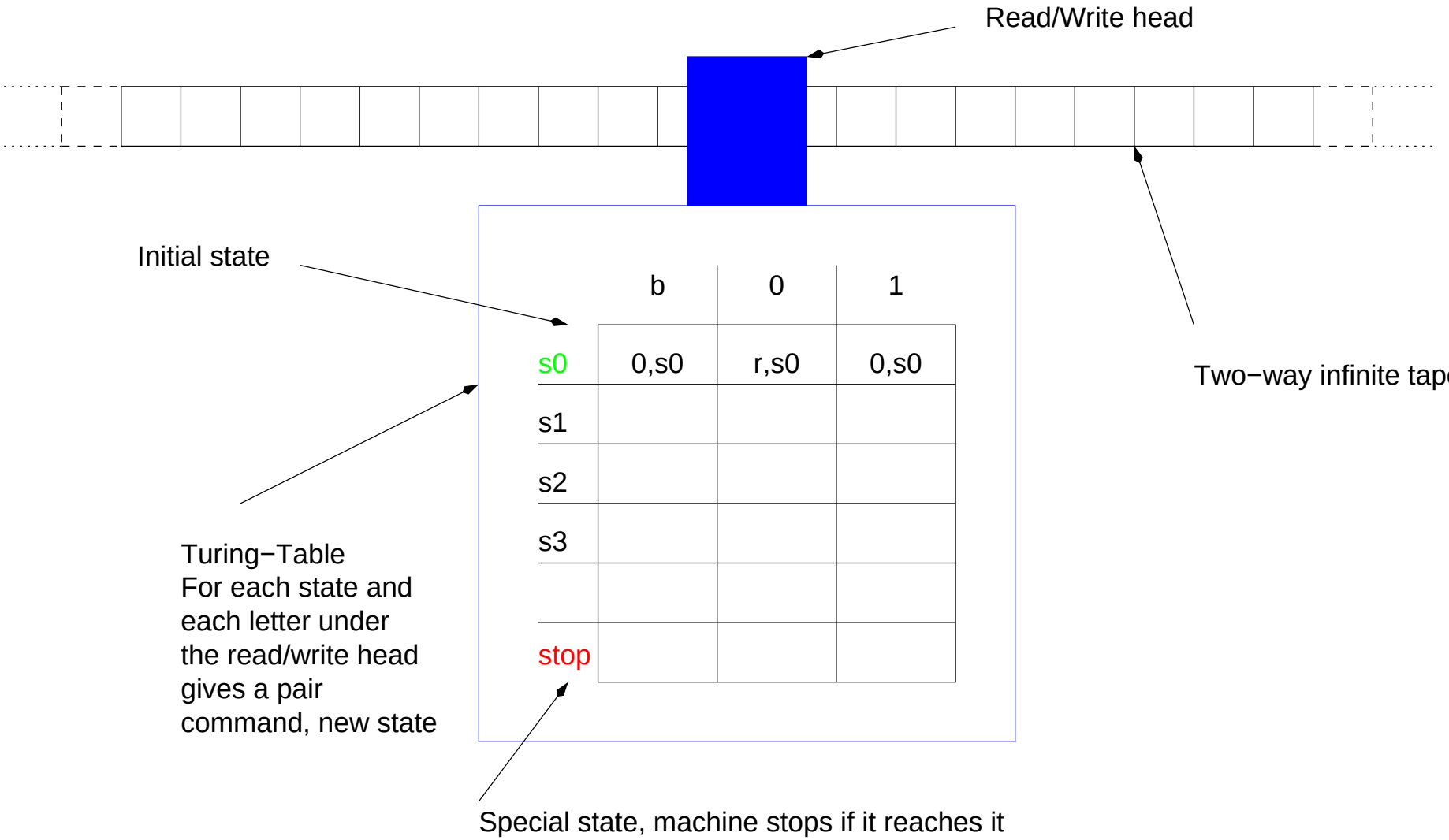
University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Turing Machines



Commands are either r,l (for moving the read/write head), or a letter (which will be written to the tape)

History and Purpose

Invented by Alan Turing as an Gedankenexperiment to discuss decidability issues

Famous paper: *On Computable Numbers, With an Application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society, Series 2, Volume 42, 1936

Turing machine is extremely simple

– Infinite tape
– Finite number of states
– Finite input alphabet
– Read/write head can read from tape, write to tape
– Commands are left, right, write something
– After each command, go to a new state

Can compute every thing PL can compute

– . . . and hence everything C or Scheme can

Thesis of Church/Turing: All reasonable programming paradigms have the same power

Formal Definition

We need:

- The blank character, often written $\mathtt{b}$
- A a finite input alphabet Σ, e.g. $\{\mathtt{0},\mathtt{1}\}$
- A tape alphabet $\tau = \Sigma \cup \{\mathtt{b}\}$
- A set of states S
- A start state $s_0 \in S$
- A special state $stop \in S$
- A Turing table

The Turing table is a function $T : (S \times \tau) \rightarrow ((\{l, r\} \cup \tau\}) \times S)$

- For each state and each character of the tape alphabet, it yields an action and the new state
- Actions:
  - l: Move head to the left
  - r: Move head to the right
  - Character from the tape alphabet: Write that character to the current field of the tap

Configurations

We always require that only a finite number of fields on the tape contain a non-blank character

Then we can write the configuration of a turing machine as a 3-tuple (α, s, ω)

- $\alpha \in \tau$ contains the non-empty part of the tape left of the read-write head (may contain additional blanks on the left)
- s is the current state
- ω contains the non-empty part of the tape under and to the right of the head, i.e. the first character of ω is the character under the head (may contain additional blanks on the right)

Convention for input:

- Arguments are separated by a single blank
- The read/write head is on the blank left of the first argument

Convention for output:

- Head is to the left of the first output word

Example

Turing machine with states s_0, s_1, s_2, s_3, s_4, input alphabet is $\{1\}$

Computation started with $(b, s_0, b111b)$:

- $(b, s_0, b111b)$
- $(bb, s_1, 111b)$
- $(bbb, s_2, 11b)$
- $(bbbb, s_1, 1b)$
- $(bbbbb, s_2, b)$
- $(bbbb, s_4, bb)$
- $(bbbb, stop, bb)$

Example Solution

Replaces argument with either 1 or empty string

– If argument has even number of characters, writes 1
– Otherwise writes empty word

Computation started with $(b, s_0, b11b)$:

– $(b, s_0, b11b)$
– $(bb, s_1, 11b)$
– $(bbb, s_2, 1b)$
– $(bbbb, s_1, b)$
– (bbb, s_3, bb)
– $(bbb, stop, 1b)$

Facts

The size of the input alphabet does not affect what functions you can compute (as long as it contains at least one character)

Giving the machine multiple tapes does not change what functions you can compute

Making the tape finite in one direction does not affect the set of computable functions

There exists a universal Turing machine:

- Takes as input an encoding of a turing machine and a configuration
- Stops with an encoding of the final configuration of that machine (if it stops)

The halting problem for Turing machines is undecidable!

- Given a machine and a start configuration, we cannot always decide if it is going to stop

Simple Turing Machine Simulator

```
#! /usr/bin/env python
import curses
import sys

screen = curses.initscr()
screen.clear()

def makestring(char, length):
    return reduce(lambda x,y:x+y, map(lambda x,y=char:y, range(length)

def printconfig(config):
    alpha, s, omega = config
    alpha = (makestring('b', 40)+alpha)[-40:]
    omega = (omega+makestring('b', 40))[:40]
    screen.addstr(1,0,alpha+omega)
    screen.refresh()
    screen.addstr(2,40,"^")
    screen.addstr(3,40,"|")
    screen.addstr(4,40,s)
```

Simple Turing Machine Simulator (2)

```
def newconfig(config, table):
    alpha, s, omega = config
    command,state = table[s][omega[0]]
    if command == 'l':
        return 'b'+alpha[:-1], state, alpha[-1]+omega
    if command == 'r':
        return alpha+omega[0], state, omega[1:]+"b"
    return alpha, state, command+omega[1:]

tt ={}
tt["s0"] = { 'b' : ("r", "s1"),
             '1' : ("1", "stop")}
tt["s1"] = { 'b' : ("b", "stop"),
             '1' : ("r", "s1")}

config = 'b', "s0", 'b111'
```

Simple Turing Machine Simulator (3)

```
printconfig(config)
while config[1]!="stop":
    config = newconfig(config, tt)
    printconfig(config)
    screen.getch()
curses.endwin()
```

Exercises

A busy beaver is a Turing machine with an input alphabet of $\{1\}$ (i.e. a tape alphabet of $\{1, b\}$ that starts with an empty tape and either

1. Tries to run for as many steps as possible (while still terminating!) or
2. Marks as many cells with 1 as possible

Current record for a busy beaver with 5 states writes 4098 1s in 47,176,870 steps

Write a good busy beaver with 3, 4, or 5 states, depending on your ambition

CSC519
Programming Languages

Excursion: The Halting Problem

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

The Halting Problem

The problem: Given a program P (in a reasonable programming language) and an input X, will P terminate on input X?

Quite important problem:

– If we can prove termination, we can prove correctness
– Solving the halting problem is equivalent to finding dead code in a program

Definitions:
– We write $P(X)$ to denote the output of program P on input X
– If P does not halt on X, we write $P(X) \uparrow$
– If P does halt on X, we write $P(X) \downarrow$ (or just $P(X)$ halts)

A reasonable programming language has (at least) the following properties:
– (There exists an interpreter I with the property that $I(P, X)$ is the same as $P(X)$)
– We can use programs as subprograms
– We have `if` and `while` (or equivalent constructs – `loop` and `goto` in PL qualify)

Proof

Claim: There is no program H with the property that $H(P, X)$ prints 1 whenever $P(X) \downarrow$ and prints 0 otherwise

Proof: Let us assume H exists. We will show that this leads to a contradiction.

– We define a program $S(X)$ as follows:

```
S(X): if(H(X,X)==1)
          while(1); /* Loop forever */
      else
          print "Hello" /* Terminate */
```

Now let us compute $H(S, S)$:

– If $S(S)$ halts, $H(S, S) = 1$, in which case S goes into the infinite loop (i.e. $S(S) \uparrow$
– If $S(S) \uparrow$, $H(S, S) = 0$, in which case $S(S)$ prints `"Hello"` and terminates (i.e. $S(S) \downarrow$)

Result: S cannot exist. But S is trivially constructed from H, so H cannot exist

q.e.d.

CSC519
Programming Languages

Final Review

Stephan Schulz

Department of Computer Science

University of Miami

schulz@cs.miami.edu

http://www.cs.miami.edu/~schulz/CSC519.html

Prerequisites: CSC 517

Final Examn

Place and Time:

- Room MM 205 (the normal room)
- Wednesday, Dec. 18th, 5:00–7:30 p.m.

Topics:

- Everything we covered in this class
- Emphasis will be on second half

You may bring:

- Lecture notes, your own notes, books, printouts of your (or my solutions)to the exercises. . .
- . . . but no computers, PDAs, mobile phones (switch them off and stow them away) or similar items

Note: I'll only review material from the second half of the semester today

- Check lecture notes, pages 301–316 for overview of first half

Python

Object oriented scripting language

- Dynamically typed (Implementation issues)
- Garbage collection (Implementation issues)
- Module system and class system

Block structure and flow control

- Blocks denoted by indention
- `if`,`while`, `for` (iterates over sequences!)

Function definitions with `def`

Type system:

- Numbers (hierarchy)
- Strings
- Classes
- Sequence types (strings with regular expressions, lists, tuples) and dictionaries (associative arrays)

Error handling with exceptions

Modules

Why modules?

- Reduce complexity by reducing interaction
- Information hiding
- Code reuse

Python modules:

- Namespace based
- One module per file
- `import <module>` imports namespace
- `from <module> import <name>` imports `<name>` into local namespace
- Modules may contain initialization code and definitions for functions and variables

Classes

Classes are implemented via namespaces

```
class example(parent1,...parentn):
    <definitions>
```

Classes can inherit properties from superclasses

- Listed after name, inherited left to right
- Names are searched for depth first, left-to-right
- All functions are virtual

Classes support:

- Constructors and destructors (note: Superclass functions are not called automatically)
- Operator overloading

Instances are created by calling the class name as a function

Scheme

Functional language: Functions can be used as values!

- Dynamically typed, gabage collected
- Central data type: Lists (with `cons`, `car`, `cdr`

Important constructs:

- `if` for conditional evaluation
- `let` for local variables
- `map` for applying a function to lists of arguments element by element
- `apply` to apply a fucntion to arguments stored in a list
- `define` for defining names for values

Syntax and evaluation:

- S-expressions (lists where the value of the first element should be a function) and symbols (which may have intrinsic values or be bound to a value)
- Evaluates all arguments, then applies first argument to the other ones

Function Definitions

`(define (fun arg1...argn) <body>)`

- `fun` is the name of the function
- `arg1...argn` are formal arguments
- `<body>` is a sequence of expressions, value returned is the value of the last expression
- Anonymous functions: `(lambda (formals) <body>)`

Example:

```
(define (listify l)
   (if (null? l)
       '()
       (cons (list (car l)) (listify (cdr l)))))
```

```
> (listify '(1 2 3 4 5 6 7))
((1) (2) (3) (4) (5) (6) (7))
> (apply append (listify '(1 2 3 4 5 6 7)))
(1 2 3 4 5 6 7)
> (map list '(1 2 3 4 5 6 7))
((1) (2) (3) (4) (5) (6) (7))
```