CS212

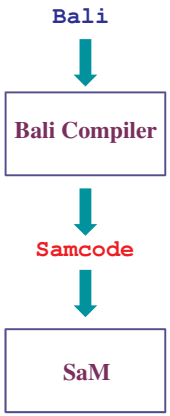
Compilers & Parsing

Announcements

- Part 1 due
- Part 2:
  - what is it?
  - subparts:
    - 2a: TA meetings
    - 2b:
- Helpful website:
  http://www.cs.cornell.edu/courses/cs211/2006sp/Sections/S3/grammars.html

Languages

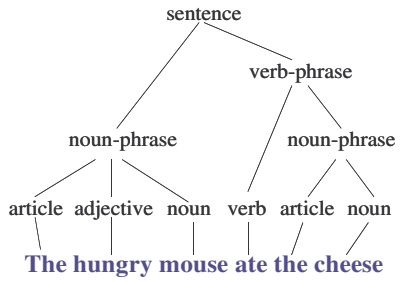
- High-level programs are written in languages (examples?)
- Compiler: what is it?
- Compiler needs to understand:
  - ***Syntax***: _____
  - ***Semantics***: _____
- Useful parallel:
  - human languages
  - look at next slide
  - why are you still reading this? stop!



Human Language: English

What is this thing below?

punctuation?



Language checker:
- Lexical analysis:
- Parsing:
- Semantic action:

How easy are these?

Syntax of statement vs semantics of a sentence?

Simple English Grammar

- ***Grammar***:
  - http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?grammar
    - set of strings over an alphabet of symbols
  - syntax specification with production rules
- Example:

  `sentence` → `nounphrase verbphrase`
  `nounphrase` → `article [adjective] noun`
  `verbphrase` → `verb nounphrase`
  `noun` → *lots of words...*
  and more...
- Elements:
  - terminals (or tokens)
  - non-terminas
- ***Context-free grammar***
  - RHS can replace LHS regardless of location

Natural vs Computer Languages

Natural Language	Computer Language
• Lexical Analysis	• Lexical Analysis
• Parsing	• Parsing
• Semantic action	• Semantic action

Lexical Analysis

- Goal: Divide program into tokens
- Token:
  - Smallest element in a language that conveys meaning
  - examples: operators, names, strings, keywords, numbers
- Tokens are typically specified using ***regular expressions***

  `a*` = repeat **`a`** zero or more times
  `a+` = repeat **`a`** one or more times
  `[abc]` = choose one of **`a`**, **`b`**, or **`c`**
  `?` = matches any one character
  `ab` = **`a`** followed by **`b`**
- Tokenizer:
  - we provide it
  - other examples:
    - Unix: **`lex`** (short for lexical analyzer)
    - Java: **`java.io.StreamTokenizer`** and **`java.util.scanner`**

Parsing

- Goals
  - Check if input string conforms to grammar (***syntax***)
  - Build a ***parse tree*** or ***abstract syntax tree*** for input string that can be used by semantic action (***semantics***)
  - use grammar
- Example:

  `expr` → **`(`** `expr` **`(+|-)`** `expr` **`)`**
  `expr` → **`int`**
- Some notation:
  - terminals, nonterminals
  - regexes
  - grouping
  - start symbol

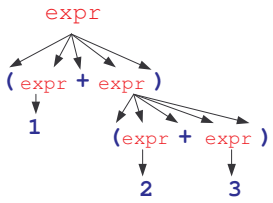
Semantics

- CS212 Bali specifications:
  - explanations of grammar and additional rules
  - why so long? why so much in English?
  - notions: ambiguity and brevity...

Trees for Parsing

Example: `( 1 + ( 2 + 3 ) )`

Parse Tree



Abstract Syntax Tree

Not Your Part 2 Grammar: Bali--

This grammar is NOT what you should use for your project!
Bali-- is simply a simplified example to help you get started!

```
program → mainfunction
mainfunction → main { declblock statblock }
declblock → { decl* }
decl → int name ;
statblock → { statement* }
statement → if ( expr ) statblock ;
statement → while ( expr ) statblock ;
statement → name = expr ;
expr → ( expr + expr )
expr → ( expr < expr )
expr → int | name
```

Some semantics: require Bali-- programs to have variable **`rv`** that stores the return value of **`main`**.

Recursive Descent Parsing

- Main idea:
  - Use grammar to recursively build AST
  - Depth-first approach
- Huh? A bit more detail:
  - Start at starting symbol
  - Parse each nonterminals (left-to-right)
    - Parse left-to-right
    - Stop when reaching a terminal
- Yes, all of this involves recursion!

Program AST

Bali-- example:

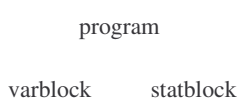
```
main {

  { // begin varblock:
    int rv; int x; int y ;
  } // end varblock

  { // begin statblock:
    x = 1 ;
    y = 2 ;
    rv = ( x + y ) ;
  } // end statblock

}
```

AST for this example?



Generating Samcode

- Each node in AST usually represents some Samcode:
  - integers: **PUSHIMM *int***
  - variables: **PUSHOFF *address***
  - see templates document for suggestions/requirements
- Example:

```
(x + ( 2 + 3 ) )
PUSHOFF 291 // wherever x happens to be
PUSHIMM 2
PUSHIMM 3
ADD
ADD
```


Assignment Statements

- Reminder from Chapter 1:
  - As you parse, you must keep track of variables
    - Each variable has space allocated on stack
    - Variable addresses must be remembered
  - Use ***symbol table*** (see **HashMap** in Java API)
- Pattern:
  - Grammar: `var = expr ;`
  - Samcode:

```
code for expr
STOREOFF varaddress
```

- Example: **x = (5 + y);** **(assume x at address 1, y @ address 2)**

```
PUSHIMM 5
PUSHOFF 2
ADD
STOREOFF 1
```


Control Structures

- Samcode labels:

```
label:
instruction
```

or

```
label: instruction
```

- Provides mechanism for jumping to other instructions (how are labels represented internally?)
- To jump to another instruction:

```
JUMP label
```

- How to model selection and repetition?

Selection Example: Bali--

- Pattern for **`if`**-statement:

```
code for expr
JUMPC label:
code for when expr is false
label:
code for when expr is true
```

- What does **`JUMPC label`** do?
  - sets **program counter** register to label
  - so, SaM effectively jumps to the instruction with address label

```
main {  { int rv ; int x ; int flag ; }
  { x = 3 ;
    flag = 0 ;
    if ( ( 2 < x ) ) {
      flag = 1 ;
    }
    rv = flag ; // return rv
  }
}
```


Selection Example: Samcode

```
// Step 1: Start program and set variables
ADDSP 3      // adjust SP to account for rv, x, and flag
PUSHIMM 3    // push value of 3
STOREOFF 1   // store the value 3 in address 1 for x
PUSHIMM 0    // push the value of 0 (false)
STOREOFF 2   // store the value 0 in address 2 for flag

// Step 2: Check if 2 < x
PUSHIMM 2    // push the value 2 to compare with x (Vbot)
PUSHOFF 1    // push the value of x (Vtop)
LESS         // Push result of (Vbot < Vtop) to top of stack

// Step 3: Process if statement
JUMPC correct  // check if result of GREATER is true (1) or false (0)
               // false:
               //    if you had an else in Bali, you would handle it here
JUMP continue  //    continue with remaining program
correct:       // true:
PUSHIMM 1      //    push the value 1 (true)
STOREOFF 2     //    store the value true for flag
JUMP continue  //    continue with remaining program
continue:      // continue with program:
PUSHOFF 2      //    push the value of flag
STOREOFF 0     //    store the value of flag in rv
ADDSP -2       //    reset the SP
STOP           //    done with the program
```


Repetition

- Idea of **`while`** **`(`** `expr` **`)`** `statblock`:
  - if expression is true, do statement block
  - check expression again,
    - if true, repeat
    - otherwise, stop and resume rest of program
- **Bali--** Example:

```
main {
  { int x ; int rv ; }
  {
    x = 1 ;
    while ( ( x < 5 ) ) {
      x = ( x + 1 ) ;
    }
    rv = x ;
  }
}
```


Repetition Samcode

```
ADDSP 2          // leave space for x and rv
PUSHIMM 1        // push 1 on the stack
STOREOFF 1       // store the value 1 for x
looplabel:       // label the loop starting at the condition
PUSHOFF 1        // retrieve the value of x
PUSHIMM 5        // push the value to compare x with
LESS             // is x < 5 ? push 1 if so; otherwise, 0
JUMPC continue   // if x < 5, do statements under continue
done:            // move to statement after the while-block
PUSHOFF 1        // retrieve the value of x
STOREOFF 0       // store the value of x as the rv
ADDSP -1         // deallocate x
STOP             // stop processing and return the rv value
continue:        // the block of statements that follow the loop
PUSHOFF 1        // retrieve the value of x
PUSHIMM 1        // push 1 onto the stack
ADD              // add 1 to the current value of x
STOREOFF 1       // store the new value of x
JUMP looplabel   // repeat the loop (goto loop condition)
```