CS212

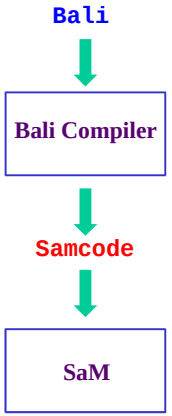
Compilers & Parsing

Announcements

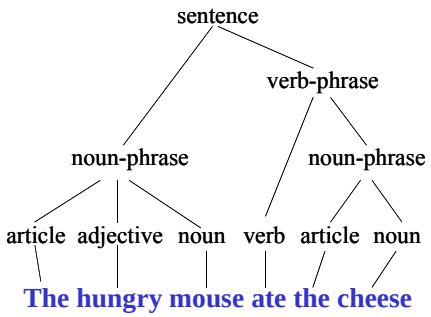
- Part 1 grades done
- Part 2:
  - 2a due date moved up (need TA meetings)
  - 2b unchanged
- Things for posting:
  - 2a, 2b
  - templates
  - grammar
  - Jar help

Languages

- High-level programs are written in languages
  - C++
  - Java
  - Bali
- Compiler:
  - Program that translates high-level to low-level
  - assembly code, byte code, Samcode...
- Compiler needs to understand:
  - ***Syntax*** (structure)
  - ***Semantics*** (meaning)
- Useful parallel: human languages



Human Language: English



Language checker:
- Lexical analysis
  - tokenizing
  - spelling
- Parsing
  - grammar
  - syntax checking
- Semantic action
  - understanding

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 alphabetic
  - 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 – Break sentence into words	• Lexical Analysis – Break program into tokens
• Parsing – Analyze word arrangement – Discover structure	• Parsing – Analyze token arrangement – Discover structure
• Semantic action – Understand sentence	• Semantic action – Generate target code

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 → num
```

- Some notation:
  - terminals, nonterminals
  - regexes
  - grouping
  - start symbol

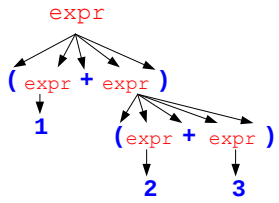
Semantics

- Reminder: meaning
- For CS212:
  - explanations of grammar and additional rules
  - why? some specifications make grammar too difficult to understand
  - you must use both syntactic and semantic descriptions, though we tend to focus on syntax first
  - more to come later....

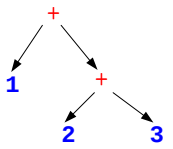
Trees for Parsing

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:

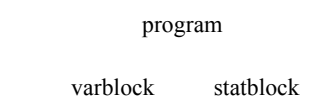
```
main {

  { // begin varblock:
    int rv; nt 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:
  - internally, each instruction has an **integer address**
  - SaM shows addresses for Samcode
  - if you provide a label, the label becomes an alias for it's instruction's address
- To jump to another instruction:

```
JUMP label
```

SaM goes to the instruction with address ***label***

- How to model selection and repetition?

Selection

- 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

Selection Example: Bali--

```
main {

  {
    int rv ; int x ; int flag ;
  }

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

}
```

What does this Bali-- program do? return?

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)
```