Week 4
Code Generation

Paul Chew
CS 212 – Spring 2004

Recall
- We use recursive descent parsing to go from program to AST (Abstract Syntax Tree)

```
x = 1; y = 1;
do 5:
x = x * y;
y = y + 1;
end;
end.
```

Prog(Assign(x,1),Assign(y,1),do(5,Assign(x,*(x,y)),Assign(y,+(y,1))))

Recall the Example Grammar

```
program → { statement } end.
statement → name = expression ;
statement → do expression ; { statement } end ;
expression → part [{ + | - | * | / } part]
part → ( name | number | ( expression ) )
nname → ( x | y | z )
```

Recursion
- The grammar drives the design of the parser
- The AST drives the design of the code generator

- We write a parsing method for each nonterminal
- We write a code-generation method for each AST node-type

- Within the method, each terminal token is checked; the nonterminals can take care of themselves (via recursive calls)
- Within the method, we generate code for the node; the subtrees can take care of themselves (via recursive calls)

Code for Expressions

- Goal is to leave expression's value on top of the SaM stack
- For our example, there are 3 kinds of expression nodes:
  - Numbers (e.g., 42)
  - Variables (e.g., x)
    - We assume x is at mem 0, y at mem 1, and z at mem 2
  - Operators (e.g., +)

Desired code
- Number
  - PUSHIMM 42
- Variable
  - PUSHOFF 0
- Operator
  - <code for left subtree>
  - <code for right subtree>
  - ADD

Example Expression Code

```
+           PUSHOFF 0
x           PUSHOFF 0
TIMES
y
```

PUSHIMM 8
SUB
ADD
**Code For Assignment Statements**

- **Goal is to store the value of the <expression> into the <variable> (e.g., y)**
  - We already have the code to place the expression’s value on top of the stack

- **Desired code**
  - \(<\text{code for expression}>\)
  - \(\text{STOREOFF 1}\)

**Example:** \(y = x + 5;\)

**Desired code**

\[
\text{PUSHOFF 0} \quad \text{PUSHIMM 5} \quad \text{ADD} \quad \text{STOREOFF 1}
\]

**Code For Do Statements**

- **This is harder because we have to maintain a counter**
  - **Goal is to**
    - Place do <expression> on top of stack to act as counter
    - If counter has reached zero we remove counter from stack and leave the loop
    - Generate code for all <statements> within the do-statement
    - Decrement the counter

- **Desired code**
  - \(<\text{code for expression}>\)
  - \(\text{loop: DUP}\)
  - \(\text{NOT}\)
  - \(\text{JUMPC endloop}\)
  - \(\text{<code for statements>}\)
  - \(\text{PUSHIMM 1}\)
  - \(\text{SUB}\)
  - \(\text{JUMP loop}\)
  - \(\text{endloop: ADDSP -1}\)

- **Mistake:** Code is wrong if <expression> is negative

**Example Program and Resulting Code**

- **do1:** \(\text{DUP}\)
  - \(\text{NOT}\)
  - \(\text{JUMPC end1}\)

- **x = 1; y = 1; do 5:**
  - \(\text{WRITE}\)
  - \(\text{WRITE}\)
  - \(\text{WRITE}\)
  - \(\text{END}\)
  - \(\text{ADDSP 3}\)
  - \(\text{PUSHIMM 1}\)
  - \(\text{STOREOFF 0}\)
  - \(\text{PUSHIMM 1}\)
  - \(\text{SUB}\)
  - \(\text{ADDSP 3}\)
  - \(\text{PUSHIMM 1}\)
  - \(\text{STOREOFF 1}\)
  - \(\text{PUSHIMM 1}\)
  - \(\text{SUB}\)

- **STOP**

**EBNF**

- **BNF = Backus-Naur Form**
  - A way of representing a grammar for a programming language
  - Originally Backus Normal Form
  - Switched at suggestion of Knuth (partly because not really a normal form)
  - Naur was editor of Algol 60 document which used BNF

- **EBNF = Extended BNF**
  - Basically, BNF with some extra simplifying notation
  - There is an official standard, but common to modify it

- **Typical constructs**
  - Way to distinguish between terminals and nonterminals
  - ( ) for repetition
  - [ ] for optional
  - ( | ) for choice

**Example Grammar Notation: Java**

**Statement:**
- Block
- if ParExpression Statement [else Statement]
- for ( ForInitOpt; Expression; ForUpdateOpt ) Statement
- while ParExpression Statement
- try Block / Catches / finally Block
- switch ParExpression ( SwitchBlockStatementGroups )
- synchronized ParExpression Block
- return (Expression); throw Expression;
- break [Identifier]
- continue [Identifier]
- ExpressionStatement
- Identifier : Statement
Example Grammar Notation: Python

```plaintext
if_stmt ::= "if" expression "::=" suite
     ( "elif" expression "::=" suite )*
     [ "else" "::=" suite ]

while_stmt ::= "while" expression "::=" suite
    [ "else" "::=" suite ]

for_stmt ::= "for" target_list "in" expression_list
    "::=" suite
    [ "else" "::=" suite ]
```

Grammar for Bali (Version for Part 2)

```
program -> mainFunction
mainFunction ->
    int main ( ) functionBody
functionBody ->
    ( variableDeclaration )
    ( statement )

type -> int | boolean
variableDeclaration ->
    type ( , name ) ;
```

Rest of the Grammar for Bali (Part 2)

```
statement -> name = expression ;
statement -> return [ expression ] ;
statement -> { statement * } ;
statement -> if expression then statement
    [ else statement ]
statement -> while expression do statement
    do statement while expression
statement -> expression ;
statement -> print expression ;
expression ->
    expPart [ binaryOp expPart ]
expPart -> integer | true | false
expPart -> readInt ( )
expPart -> name
expPart -> ( expression )
expPart -> unaryOp expPart
binaryOp -> arithmeticOp | comparisonOp | booleanOp
arithmeticOp -> + | - | * | / | %
comparisonOp -> < | > | <= | >= | == | !=
booleanOp -> & & | || | ^
unaryOp -> - | !
```