CS420 Fall 2016

Programming Assignment 1 & 2

Parsing, Abstract Syntax Tree and Symbol Table

Objectives

This assignment is to create a parser which builds an abstract syntax tree (AST) and symbol tables from reading a miniC program. In the process of parsing, you need to build an AST and symbol table. At the end of parsing, you must print out an AST and symbol tables in two different text files. When you print out AST, try to print out C like format – it does not have to be the exact C but resembles the structure of source code and reflects the AST.

How to grade your submission?

1) Comparing a meaning of parsing result and input source code
2) Correctness of symbol tables

Due & files to be included in your submission

Please submit your source code packaged together through KLMS. Source code must be handed in one zip file and it also needs to include a README file which describes how to build and run your submission. The zip file also needs to include source codes, and test C code. Your grade will be mainly based on your program outputs which will be built and tested according to your instructions. Make your zip file name "YOURSTUDENTID.zip" when you submit it to KLMS.

First step: Due October 17 9:00AM
Build your parser for miniC grammar. Write your input for parser generator and scanner generator you like. Submit your package.

The second step: Due TBA
Design your AST specification. Modify your parser specification to generate the proper AST for miniC. Write the visitor to print your AST in C like format. Write a module to dump your symbol table in proper format, which will be useful debug emitted machine instruction later.

miniC Grammar

// (a)$^+$: one or more repetition of *a*

// (a)* : zero or more repetition of *a*

// (a)? : *a* optionally exists

// **unop** : **-** // **binop** : *, /, +, -, <, >, <=, >=, ==, !=

// **id** : [A-Za-z][A-Za-z0-9_]* // **intnum** : [0-9]$^+$ // **floatnum** : [0-9]$^+$**.**[0-9]$^+$

Program := (*DeclList*)? (*FuncList*)? // *DeclList FuncList* | *DeclList* | *FuncList* | ε

DeclList := (*Declaration*)$^+$ // *Declaration* | *DeclList Declaration*

FuncList := (*Function*)$^+$

Declaration := *Type IdentList* **;**

IdentList := *identifier* (, *identifier*)* // *identifier* | *IdentList* , *identifier*

identifier := **id** | **id** [**intnum**] // (Note) [,] are not symbols used in regular expression

Function := *Type* **id** **(** (*ParamList*)? **)** *CompoundStmt*

ParamList := *Type identifier* (**,** *Type identifier*)*

Type := `int` | `float`

CompoundStmt := **{** (*DeclList*)? *StmtList* **}**

StmtList := (*Stmt*)*

Stmt := *AssignStmt* | *CallStmt* | *RetStmt* | *WhileStmt* | *ForStmt* | *IfStmt* | *SwitchStmt* | *CompoundStmt* | **;**

AssignStmt :=*Assign* **;**

Assign := **id** = *Expr* | **id** **[** *Expr* **]** = *Expr*

CallStmt **:=** *Call* **;**

Call := **id** **(** (*ArgList*)? **)**

RetStmt := `return` (*Expr*)? **;**

WhileStmt := `while` **(** *Expr* **)** *Stmt* | `do` *Stmt* `while` **(** *Expr* **)** **;**

ForStmt := `for` **(** *Assign* **;** *Expr* **;** *Assign* **)** *Stmt*

IfStmt := `if` **(** *Expr* **)** *Stmt* (`else` *Stmt*)?

SwitchStmt **:=** `switch` **(***identifier***)** **{** *CaseList* **}**

CaseList`:=` `(case` **intnum**`:` *StmtList* `(break;)?)`* `(default:` *StmtList* `(break;)?)?`

Expr := **unop** *Expr* | *Expr* **binop** *Expr* | *Call* | **intnum** **|** **floatnum** | **id** | **id** **[** *Expr* **]** | **(** *Expr* **)**

ArgList := *Expr* (, *Expr*)*

Operator Precedence and Associativity

Precedence	Operator	Description	Associativity
1	( )	Function call	Left-to-right
2	-	Unary minus	Right-to-left
3	*, /	Multiplication and division	Left-to-right
4	+, -	Addition and subtraction	Left-to-right
5	<, >, <=, >=	Relational operators	Left-to-right
6	==, !=	Equality operators	Left-to-right
7	=	Assignment	Right-to-left

Symbol Table (refer to Chap. 5.5 in the textbook)

Symbol table entry := <symbol name, symbol info>

Symbol info := <type, location, pointer to its associated declaration>

location := level of nested blocks and order of blocks

Sample Compiler Skeleton

Included zip file contains a sample compiler skeleton used in "Modern Compiler Implementation in Java" written by Andrew Appel. Even though it is for Mini Java language, the program structure will help you to organize your directories and corresponding modules. You need to adjust some details, such as TEMINALs in Grm.cup, AST definitions in Absyn directory, but it will be useful.