04/07/2011

Recitation #11

Administrative

- Final Project (due **April 26 @ 11:59 p.m.**)
- In the middle of grading assignment 4. Hopefully done by the beginning of next week.

SLR and LALR Parser

- You can generate the parsing tables with the **Parsing Emulator** tool available on my Wiki page
- Show SLR examples from Recitation #9 and expression grammar from Dr. Hughes' slides

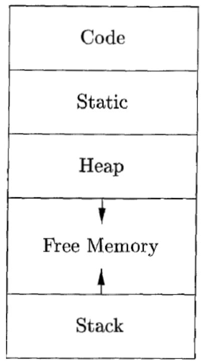
Attributes

- In Recitation #10 we talked about synthesized and inherited attributes → **Review**
- Go over the small example of inherited attributes to illustrate the concepts

Final Project – Rascal Compiler

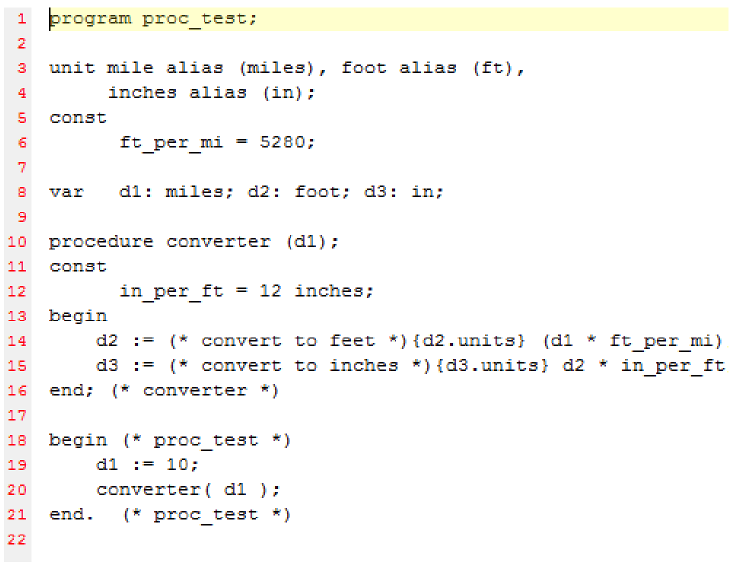
- Brings together material from previous assignments and expands upon Assignment 4
- Dr. Hughes will only give you a binary reference solution this time → Skeleton Flex/Bison files don't contain any action code
- You have a lot of freedom in terms of actual data structures and individual implementation
- Project description is evolving, so please check the project folder on the course website often
- **Lessons learned** (hopefully ☺) from previous assignments (harkens back to recitation 1):
    - Modularity
    - Graphical Debugging
    - Write your own test cases

- **Constants**
    - Constant values that are read-only
    - When referenced in code generate CON intermediate code
    - No assignment to constants possible → generate semantic error
- **For-Loop** construct
    - Implementation follows standard
    - Look at grammar for details
- No Do/While, but Repeat/Until
- **Units**
    - Aliases are allowed; you don't know them until you parse the input program
    - Implementation of aliases
        1. Use symbol table to store types and what original type they implement.
        2. Only store original unit. Use data structure to map all aliases to the original unit → Union-Find ideal for that purpose, in C++ use std::map, in C simple array with pointers should work as well



   - Casting (explicit, implicit)
- **Procedures** → No functions
   - Stack-allocation of space
      1. Each time procedure is called, space for local variables is pushed onto the stack
      2. When procedure terminates, space is popped off the stack
      3. Memory space is shared by procedures whose durations do not overlap in time
      4. Relative address of its variables is always the same
   - Calls and returns
   - Local variables → Shadowing of global variables is an issue, i.e. allow identical names
- **Semantic error detection**
   - Errors will mostly be encountered in relation to operations on units
- **Symbol table**
   - Expanded from assignment 4
   - You decide on elements or if you want to store some information in external structures
- **Optimizations**
   - Dr. Hughes lists some in his project description
   - Bonus points for clever optimizations that you can come up with

- Show **example of generated code** for some variables and procedure call
   - Caveat that this is my version of how I save information
   - I chose to put all information in a global symbol table
      1. Aliases are just lookups to original units
      2. Constants need to save their value
      3. Procedures save entry triple in "offset", local storage size in "size" and number of arguments in "Args"
   - Questions about this global implementation:
      1. How do you distinguish nested function calls, i.e. local variables in function A shouldn't be accessible in function B, but we only use one StoreType?
      2. Lots of unused fields. How can we make that more efficient? → Use unions in C
   - **You should decide on your own data structures and implementations**



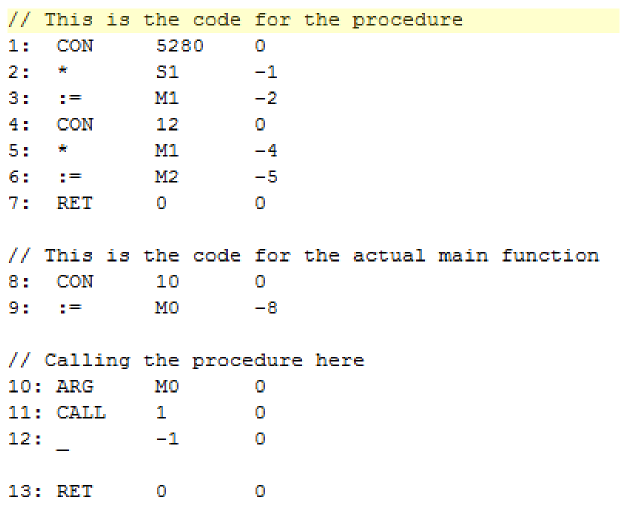
```
program proc_test;

unit mile alias (miles), foot alias (ft),
     inches alias (in);
const
      ft_per_mi = 5280;

var   d1: miles; d2: foot; d3: in;

procedure converter (d1);
const
      in_per_ft = 12 inches;
begin
    d2 := (* convert to feet *){d2.units} (d1 * ft_per_mi)
    d3 := (* convert to inches *){d3.units} d2 * in_per_ft
end; (* converter *)

begin (* proc_test *)
    d1 := 10;
    converter( d1 );
end.  (* proc_test *)
```



```
// This is the code for the procedure
1:  CON     5280    0
2:  *       S1      -1
3:  :=      M1      -2
4:  CON     12      0
5:  *       M1      -4
6:  :=      M2      -5
7:  RET     0       0

// This is the code for the actual main function
8:  CON     10      0
9:  :=      M0      -8

// Calling the procedure here
10: ARG     M0      0
11: CALL    1       0
12: _       -1      0

13: RET     0       0
```

Resulting (global) symbol table for my example:

Identifier	Class	Unit	StoreType	Offset	Size	Args	Value	Low	High
miles	alias	mile	-	-	-	-	-	-	-
ft	alias	foot	-	-	-	-	-	-	-
in	alias	inches	-	-	-	-	-	-	-
ft_per_mi	const	NUMBER	0	-	0	-	5280	-	-
d1	var	mile	0	0	0	-	-	-	-
d2	var	foot	0	1	0	-	-	-	-
d3	var	inches	0	2	0	-	-	-	-
converter	proc	-	-	-1	0	1	-	-	-
d1	var	mile	1	1	0	-	-	-	-
in_per_ft	const	inches	1	-	0	-	12	-	-