NTNU – Trondheim
Norwegian University of
Science and Technology

Recitation lecture: problem set 2

- Theory part
- Practical part
- C specifics

Content of the archive

- src/ - contains C source files
- include/ - contains C header files
- vsl_programs/ - contains example VSL programs for testing
  - Contains a makefile to run your vslc
  - ‘make’ to make all or ‘make <path>.ast to run on a single file
- ‘make’ – builds the the compiler as src/vslc
  - Add ‘clean’ to remove intermediate files, or ‘purge’ to remove binaries as well

Things to implement

- Scanner in **src/scanner.l**
  - Needs to return all types of tokens
- Parser in **src/parser.y**
  - Constructs syntax tree as tokens are received
  - Matched text available through **yytext** and special variables $1, $2..
- Auxiliary functions in **src/tree.c**
  - Construction and deletion of dynamically allocated nodes
  - **node_t** struct defined in **include/ir.h**
  - **node_print** is already implemented

Yacc and Lex

- Lex is a specification for scanner generators, **flex** is one implementation
- Yacc is a specification for parser generators, **bison** is one implementation
- Install: **sudo apt install flex bison**
  - Assuming Ubuntu based OS or WSL distribution

Lex specifications

```
definitions
%%
regular expression { matching action }
…
%%
other code
```

- Regular C code can be embedded, enclosed between ‘**%{**’ and ‘**%}**’
- Helpful directives: **yylineno** and friends
- Code section may be practically empty, keeping logic section in parser

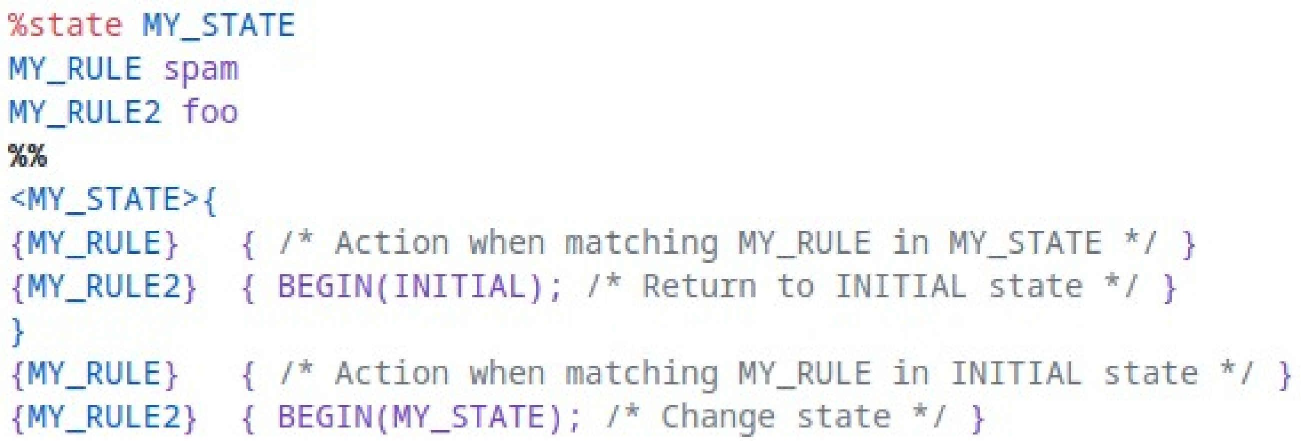
Status of the scanner

- Three rules are already implemented
  - **{WHITESPACE}+** eliminates all whitespace.
  - **{COMMENT}** eliminates comments (named regex).
  - **.** sends catches all remaining characters and returns them one by one.
- Symbolic names for multi-character tokens are defined in a header generated from the **%token** directive used in **src/parser.y**
- Add regex for remaining tokens

NTNU – Trondheim
Norwegian University of
Science and Technology

Token names

- Tokens are mostly named after their keywords
- Exception: BEGIN and END are named OPENBLOCK and CLOSEBLOCK
  - Flex macro BEGIN switches internal state: **BEGIN(**<new state>**)**

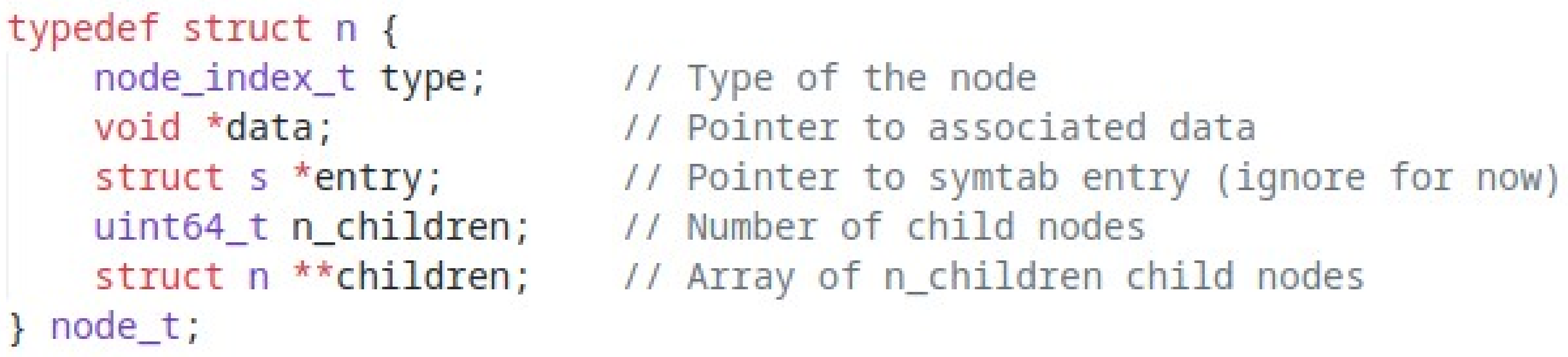


```
%state MY_STATE
MY_RULE spam
MY_RULE2 foo
%%
<MY_STATE>{
{MY_RULE}   { /* Action when matching MY_RULE in MY_STATE */ }
{MY_RULE2}  { BEGIN(INITIAL); /* Return to INITIAL state */ }
}
{MY_RULE}   { /* Action when matching MY_RULE in INITIAL state */ }
{MY_RULE2}  { BEGIN(MY_STATE); /* Change state */ }
```

Want Yacc and Lex syntax highlighting? Recommend 'yash' for VS Code

Structure node_t

- Used to build the syntax tree
- Bit of tricky pointer acrobatics



```
typedef struct n {
    node_index_t type;      // Type of the node
    void *data;             // Pointer to associated data
    struct s *entry;        // Pointer to symtab entry (ignore for now)
    uint64_t n_children;    // Number of child nodes
    struct n **children;    // Array of n_children child nodes
} node_t;
```

The auxiliary functions

- Initializer function for **node_t** takes a node (pre-allocated), type, data,n_children and a variable amount of **node_t** objects (va-list)

```
void node_init (
    node_t *nd, node_index_t type, void *data, uint64_t n_children, ...
);
```

- VA-list from **stdarg.h** (included in **vslc.h**) will have to be read
  - \`…\` syntax probably familiar from the printf/scanf function family

```
va_list valist;                              // Initialize valist
va_start(valist, n_children);                // Set boundaries
for (int i = 0; i < n_children; i++) {       // Iterate list
    node_t *child_n = va_arg(valist, node_t *); // Extract argument (valist, type)
}
```

The auxiliary functions

- **node_finalize** and **subtree_destroy**
  - Use **subtree_destroy** as a recursive destructor in order to free the whole tree.
- All heap allocated objects need to be freed when done.
- **Valgrind** is a useful tool to check for memory leaks

Why are arguments passed by reference?

- Objects and arrays can be very large, wasteful to copy into a function call.
- Pointers are always a 32/64 bit address.
- Passing allocated **node_t*** to initializer
  - Could as well have allocated the node inside the function and returned a pointer to the newly created object
  - Convention to let the caller decide how the object is allocated

Yacc specifications

- Yacc has the same structure as Lex
- Rules are implemented similarly to the Backus-Naur form (more examples in skeleton)

```
expr :
        expr '+' expr { /* parsed an addition */ }
      | expr '-' expr { /* parsed a subtraction */ }
      ;
```

- $1, $2 etc refer to the n'th token in a production.
- $$ refers to the object returned by the production (type node_t*)
- \`**expr '+' expr**\` $1 and $3 are node_t * objects representing the two expressions
  - All **expr op expr** will look identical in the syntax tree, remember to stash the operator in the data field.

NTNU – Trondheim
Norwegian University of Science and Technology

Status of the parser

- Most supporting structures
  - Tokens
  - Error handling
- Some dummy produtions
  - These are *in no way* correct for the parser you are writing, but serve as a demonstration of the Yacc syntax.

Bottom of the tree

- The smallest reductions like STRING and INTEGER have just a token on r.h.s.
- $$ is a node_t but INTEGER is just a token
- The semantic rule has to create a leaf node containing the data
  - Parse the content of **yytext**
  - The content of **yytext** will change as parsing continues, so remember to copy the data. (functions to consider: **strcpy**, **strdup**, **sscanf**, **strtol**)

Parsing data

- **int64_t my_int = strtol(yytext, NULL, 10);**
  - Will parse a 64-bit integer (**atoi** is deprecated)
- Arguments are
  - **char *buffer** ← where text is found
  - **char *end** ← where translation stops (Not needed now)
  - **int base** ← base (we use base 10 integers)
- **char *data = strdup(yytext);**
  - Mild violation of “caller allocates” rule, but it’s a common exception. An alternative is the more cumbersome
    char *data = malloc (strlen(yytext)+1);
    strcpy(data, yytext);
  - **$$->data = strdup(yytext);**

NTNU – Trondheim
Norwegian University of Science and Technology

VSL expressions

- The arithmetic expressions define an ambiguous sub-grammar
- Instead of having to disambiguate the grammar, Yacc supports precedence rules:

  %left ‘+’ ‘-’
  %left ‘*’ ‘/’
  %nonassoc UMINUS

  – Assign left associativity for binary operations, and assigns UMINUS the highest precedence, while add/sub gets the lowest
- Same goes for if-else (*dangling else* problem)

  %nonassoc IF THEN
  %nonassoc ELSE
- Take a moment to appreciate this feature

How I would attack it

- Isolate the scanner
  - The main function calls **yyparse**, comment it out and call **yylex** while completing the scanner
- Test the auxiliary functions in main while getting comfortable with them
- Connect back with the parser
  - Reintroduce **yyparse** instead of **yylex**
  - Add one production at the time, e.g. let **program** catch an integer, then extend to a declaration, then a list etc in your preferred order
- Apply your preferred code style
  - Your hand-in does not have to look like what was handed out, but please be consistent in you coding style.

How I would attack it

- What you put in the **data** field will vary, the context of what it contains is given by the node's type
- Don't get tempted to use **void*** as a character literal (remember it is a pointer)
  Dangerous: **$$->data = (void*)'+';**
  Better:
  $$->data = (char *)malloc(1);
  ***(char*)$$->data = '+';**
  ...
  char my_data = (char*)node->data;

NTNU – Trondheim
Norwegian University of Science and Technology

Touch typing class

- One aim of this exercise is to get the hang of handling trees in dynamic memory
- Once you get the idea, the rest is mainly a matter of typing variations of a theme – large, but not particularly difficult
- Secondary point: Just how quickly the complexity of a language grows
- Tip: Macros can save you a lot of typing

```
#define MY_MACRO(x, y, z) do { \
        do_something(x, y, z); \
while (false)
```

GL/HF

- Ask questions
- Good Luck
- Hopefully have a little fun as well

Looking forward

- The generated tree contains redundant information
  - Left recursive productions make deep trees out of lists
  - Expressions with all constants could be reduced to simple integers
  - Etc.
- We will tidy up later
  - Straight forward parsing keeps the parser code as simple as possible and is OK for now
  - **entry** field is currently unused. We will use this later for creating symbol tables. It can be NULL for now