南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CS323 Lab 8

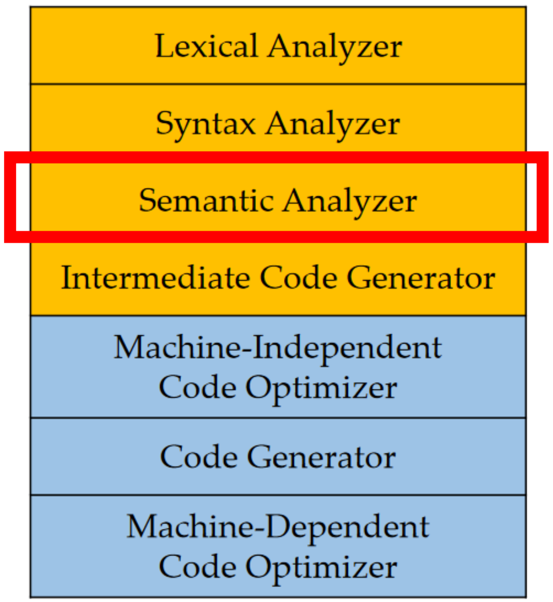
Yepang Liu

liuyp1@sustech.edu.cn

Outline

- Overview of Project Phase 2
- Understand the parsers generated by Bison

Phase 2



- Build a semantic analyzer to check if the given SPL program satisfies semantic rules by analyzing parse tree and symbol table
  - No need to output anything if there is no rule violation
  - Otherwise, report the semantic errors (type, line number, error message, etc.)

Assumptions

Assumption 1 `char` variables only occur in assignment operations or function parameters/arguments

Assumption 2 only `int` variables can do boolean operations

Assumption 3 only `int` and `float` variables can do arithmetic operations

Assumption 4 no nested function definitions

Assumption 5 field names in `struct` definitions are unique (in any scope), i.e., the names of `struct` fields, and variables never overlap

Assumption 6 there is only the global scope, i.e., all variable names are unique

Assumption 7 using *named equivalence* to determine whether two `struct` types are equivalent

```
struct st {int x;}

struct st {char y;}
```

The two structures are of the same type if we adopt named equivalence

Required rules

Type 1 a variable is used without a definition

Type 2 a function is invoked without a definition

Type 3 a variable is redefined in the same scope

Type 4 a function is redefined (in the global scope, since we don't have nested functions)

Type 5 unmatching types appear at both sides of the assignment operator (`=`)

Type 6 rvalue appears on the left-hand side of the assignment operator

Type 7 unmatching operands, such as adding an integer to a structure variable

Type 8 a function's return value type mismatches the declared type

Type 9 a function's arguments mismatch the declared parameters (either types or numbers, or both)

Type 10 applying indexing operator (`[…]`) on non-array type variables

Type 11 applying function invocation operator (`foo(…)`) on non-function names

Type 12 array indexing with a non-integer type expression

Type 13 accessing members of a non-structure variable (i.e., misuse the dot operator)

Type 14 accessing an undefined structure member

Type 15 redefine the same structure type

Test cases

- We will provide test cases on GitHub (under `project/phase2/test/`)

- You are required to pass all of them

Example

```
1  struct Apple
2  {
3      int weight;
4      float round;
5  };
6  int test_2_r07()
7  {
8      struct Apple aa;
9      float weight_test = 1.0;
10     aa.weight = aa + 2;
11     return 0;
12 }
```

```
Error type 7 at Line 10: binary operation on non-number variables
Error type 5 at Line 10: unmatching type on both sides of assignment
```

What to submit?

- Your semantic analyzer source code + a brief report
- Five test cases each team
  - At least four violations of our semantic rules
  - No lexical or syntax errors
- If you implement additional rule checkers:
  - Document the features in the project report
  - Provide test cases
- Deadline: 10:00 PM, December 3, 2023

Outline

- Overview of Project Phase 2
- Understand the parsers generated by Bison

Example Grammar *G*

- S → CC
- C → cC | d

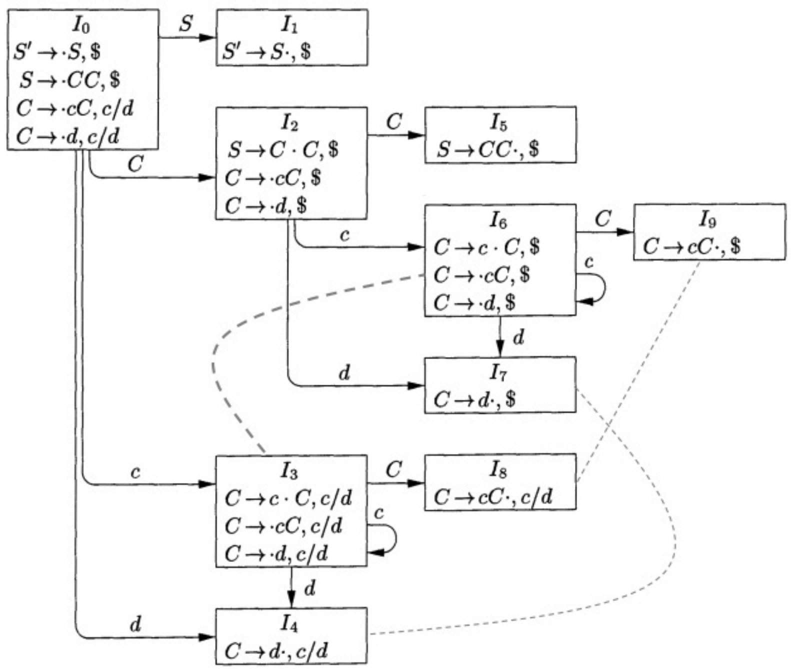
Step 1: Constructing LALR(1) Automaton

- Manually construct the LR(1) item sets following the algorithm below

```
void items(G') {
        initialize C to{CLOSURE({[S' → ·S, $]})};
        repeat
                for ( each set of items I in C )
                        for ( each grammar symbol X )
                                if ( GOTO(I, X) is not empty and not in C )
                                        add GOTO(I, X) to C;
        until no new sets of items are added to C;
}
```

Step 1: Constructing LALR(1) Automaton

- Merge the item sets with the same core



Merge the following item sets:

- I_3 and I_6
- I_4 and I_7
- I_8 and I_9

Step 2: Generate the LALR(1) Automaton Using Bison

- Write a Flex program to recognize two patterns “c” and “d”
- Write a Bison program to recognize the language $L(G)$
- We provide sample programs on GitHub if you cannnot finish the above two programs by yourself
  - Under the `lab8` directory
- Build the Bison program using the commands below:
  - `bison -d syntax.y --report all` (will generate a file “syntax.output”)
  - `flex lex.l`
  - `gcc syntax.tab.c -lfl -ly -o test.out`
  - The test.out executable can recognize strings such as “dd” (can be tested with `echo “dd” | ./test.out`)

Step 3: Understand the Generated LALR(1) Automaton

- Check the “syntax.output” file to understand the automaton

```
Grammar

    0 $accept: S $end

    1 S: C C

    2 C: c C
    3  | d
```

```
Terminals, with rules where they appear

$end (0) 0
error (256)
c (258) 2
d (259) 3
```

```
Nonterminals, with rules where they appear

$accept (5)
    on left: 0
S (6)
    on left: 1, on right: 0
C (7)
    on left: 2 3, on right: 1 2
```

```
State 0

    0 $accept: . S $end
    1 S: . C C
    2 C: . c C
    3  | . d

    c  shift, and go to state 1
    d  shift, and go to state 2

    S  go to state 3
    C  go to state 4
```

Questions

- The automaton contructed by Bison has one more state than the one constructed manually by you.
  - Which state in the automaton generated by Bison does not appear in the automaton manually constructed by you?
  - Why is there such a difference?

- Is the automaton generated by Bison equivalent to the one manually contructed by you in terms of language recognition ability?