Principles of Programming Languages

CS 314

Recitation 5

RUTGERS

Topics Today

- Project 1
  - tinyL language
- C Debug

TinyL Language

```
<program>     ::=  <stmt_list> .
<stmt_list>   ::=  <stmt> <morestmts>
<morestmts>   ::=  ; <stmt_list>  |  ϵ
<stmt>        ::=  <assign>  |  <read>  |  <print>
<assign>      ::=  <variable> = <expr>
<read>        ::=  ! <variable>
<print>       ::=  # <variable>
<expr>        ::=  + <expr> <expr>  |
                   − <expr> <expr>  |
                   ∗ <expr> <expr>  |
                   <variable>  |
                   <digit>
<variable>    ::=  a  |  b  |  c  |  d  |  e
<digit>       ::=  0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |  8  |  9
```

TinyL Language

- FIRST sets of tinyL language

FIRST(<*program*>) = {“a”, “b”, “c”, “d”, “e”, “!”, “#”}

FIRST(<*stmt_list*>) = {“a”, “b”, “c”, “d”, “e”, “!”, “#”}

FIRST(<*morestmts*>) = {“;”, ε}

FIRST(<*stmt*>) = {“a”, “b”, “c”, “d”, “e”, “!”, “#”}

FIRST(<*assign*>) = {“a”, “b”, “c”, “d”, “e”}

FIRST(<*read*>) = {“!”}

FIRST(<*print*>) = {“#”}

FIRST(<*expr*>) = {“+”, “-”, “*”, “a”, “b”, “c”, “d”, “e”, “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”}

FIRST(<*variable*>) = {“a”, “b”, “c”, “d”, “e”}

FIRST(<*digit*>) = {“0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”}

TinyL Language

- FOLLOW sets of tinyL language

FOLLOW(<*program*>) = {eof}

FOLLOW(<*stmt_list*>) = {“.”}

FOLLOW(<*morestmts*>) = {“.”}

FOLLOW(<*stmt*>) = {“;”, “.”}

FOLLOW(<*assign*>) = {“;”, “.”}

FOLLOW(<*read*>) = {“;”, “.”}

FOLLOW(<*print*>) = {“;”, “.”}

FOLLOW(<*expr*>) = {“;”, “.”,“+”, “-”, “*”, “a”, “b”, “c”, “d”, “e”, “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”}

FOLLOW(<*variable*>) = {“;”, “.”, “=”,“+”, “-”, “*”, “a”, “b”, “c”, “d”, “e”, “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”}

FOLLOW(<*digit*>) = {“;”, “.”,“+”, “-”, “*”, “a”, “b”, “c”, “d”, “e”, “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”}

TinyL Language

- PREDICT sets of tinyL language (1/3)

1. PREDICT(<*program*> ::= <*stmt_list*>) = {“a”, “b”, “c”, “d”, “e”, “!”, “#”}
2. PREDICT(<*stmt_list*> ::= <*stmt*> <*morestmts*>) = {“a”, “b”, “c”, “d”, “e”, “!”, “#”}
3. PREDICT(<*morestmts*> ::= ; <*stmt_list*>) = {“;”}
4. PREDICT(<morestmts> ::= ε) = {“.”}
5. PREDICT(<*stmt*> ::= <*assign*>) = {“a”, “b”, “c”, “d”, “e”}
6. PREDICT(<*stmt*> ::= <*read*>) = {“!”}
7. PREDICT(<*stmt*> ::= <*print*>) = {“#”}
8. PREDICT(<*assign*> ::= <*variable*> = <*expr*>) = {“a”, “b”, “c”, “d”, “e”}
9. PREDICT(<*read*> ::= ! <*variable*>) = {“!”}
10. PREDICT(<*print*> ::= # <*variable*>) = {“#”}

- PREDICT sets of tinyL language (2/3)

11. PREDICT(<*expr*> ::= + <*expr*> <*expr*>) = {“+”}
12. PREDICT(<*expr*> ::= - <*expr*> <*expr*>) = {“-”}
13. PREDICT(<*expr*> ::= * <*expr*> <*expr*>) = {“*”}
14. PREDICT(<*expr*> ::= <*variable*>) = {“a”, “b”, “c”, “d”, “e”}
15. PREDICT(<*expr*> ::= <*digit*>) = {“0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”}
16. PREDICT(<*variable*> ::= a) = {“a”}
17. PREDICT(<*variable*> ::= b) = {“b”}
18. PREDICT(<*variable*> ::= c) = {“c”}
19. PREDICT(<*variable*> ::= d) = {“d”}
20. PREDICT(<*variable*> ::= e) = {“e”}

- PREDICT sets of tinyL language (3/3)

21. PREDICT(<*digit*> ::= 0) = {“0”}
22. PREDICT(<*digit*> ::= 1) = {“1”}
23. PREDICT(<*digit*> ::= 2) = {“2”}
24. PREDICT(<*digit*> ::= 3) = {“3”}
25. PREDICT(<*digit*> ::= 4) = {“4”}
26. PREDICT(<*digit*> ::= 5) = {“5”}
27. PREDICT(<*digit*> ::= 6) = {“6”}
28. PREDICT(<*digit*> ::= 7) = {“7”}
29. PREDICT(<*digit*> ::= 8) = {“8”}
30. PREDICT(<*digit*> ::= 9) = {“9”}

- Parse Table

	.	;	!	#	+	-	*	a	b	c	d	e	0	1	2	3	4	5	6	7	8	9
program			1	1				1	1	1	1	1										
stmt_list			2	2				2	2	2	2	2										
morestmts	4	3																				
stmt			6	7				5	5	5	5	5										
assign								8	8	8	8	8										
read			9																			
print				10																		
expr					11	12	13	14	14	14	14	14	15	15	15	15	15	15	15	15	15	15
variable								16	17	18	19	20										
digit													21	22	23	24	25	26	27	28	29	30

- Debugger
  - Provides:
    - Stepping (step and step-into)
    - Breakpoint
    - Tracking of every object's state
  - gdb

C Debug

gdb

Step 1: compile with the debugging option –g

$ gcc –g factorial.c

Step 2: launch gdb with the executable

$ gdb a.out

Step 3: execute the program in gdb

(gdb) run

Step 4: quit gdb

(gdb) quit

```c
#include <stdio.h>

int main() {
    int res = 1;
    int num;

    printf("Enter the number:");
    scanf("%d", &num);

    int i;
    for (i = 1; i <= num; i++)
        res *= i;

    printf("The factorial of %d is %d\n", num, res);

    return 0;
}
```

```
➜ c_debug gcc -g factorial.c
➜ c_debug ./a.out
Enter the number:3
The factorial of 3 is 6
➜ c_debug gdb a.out
(gdb) run
Starting program: /ilab/users/yc827/chenyh64/Course/CS314/c_debug/a.out
Enter the number:3
The factorial of 3 is 6
[Inferior 1 (process 25580) exited normally]
(gdb) quit
➜ c_debug
```

C Debug

gdb – Breakpoint, Print

Set up a break point for the C program

Syntax:

break *lineNumber*

break *fileName:lineNumber*

break *fileName:funcName*

```c
#include <stdio.h>

int main() {
    int res = 1;
    int num;

    printf("Enter the number:");
    scanf("%d", &num);

    int i;
    for (i = 1; i <= num; i++)
        res *= i;

    printf("The factorial of %d is %d\n", num, res);

    return 0;
}
```

Print out the variable values inside gdb

Syntax:

print *variable*

```
(gdb) break 12
Breakpoint 1 at 0x4005ca: file factorial.c, line 12.
(gdb) run
Starting program: /ilab/users/yc827/chenyh64/Course/CS314/c_debug/a.out
Enter the number:3

Breakpoint 1, main () at factorial.c:12
12              res *= i;
(gdb) print res
$1 = 1
(gdb) print i
$2 = 1
```

C Debug

gdb – Step, Next, Finish, Continue

step: Go to next instruction (source line), diving into function.

next: Go to next instruction (source line) but don‘t dive into functions.

finish: Continue until the current function returns.

continue: Continue normal execution.

```c
#include <stdio.h>

int main() {
    int res = 1;
    int num;

    printf("Enter the number:");
    scanf("%d", &num);

    int i;
    for (i = 1; i <= num; i++)
        res *= i;

    printf("The factorial of %d is %d\n", num, res);

    return 0;
}
```

C Debug

```
(gdb) break 7
Breakpoint 1 at 0x40059c: file factorial.c, line 7.
(gdb) run
Starting program: /ilab/users/yc827/chenyh64/Course/CS314/c_debug/a.out

Breakpoint 1, main () at factorial.c:7
7           printf("Enter the number:");
(gdb) step
__printf (format=0x400690 "Enter the number:") at printf.c:29
29      {
(gdb) step
33        va_start (arg, format);
(gdb) finish
Run till exit from #0  __printf (format=0x400690 "Enter the number:")
    at printf.c:33
main () at factorial.c:8
8           scanf("%d", &num);
Value returned is $1 = 17
(gdb) next
Enter the number:3
11          for (i = 1; i <= num; i++)
(gdb) continue
Continuing.
The factorial of 3 is 6
[Inferior 1 (process 31255) exited normally]
```

```c
#include <stdio.h>

int main() {
    int res = 1;
    int num;

    printf("Enter the number:");
    scanf("%d", &num);

    int i;
    for (i = 1; i <= num; i++)
        res *= i;

    printf("The factorial of %d is %d\n", num, res);

    return 0;
}
```

C Common Mistakes

```c
#include <stdio.h>

int main()
{
   int a;
   scanf("%d", a);
   printf("%d", a);
   return 0;
}
```

```c
#include <stdio.h>

int main()
{
   int a;
   scanf("%d", &a);
   printf("%d", a);
   return 0;
}
```

C Common Mistakes

```c
#include <stdio.h>

int main()
{
    int* p = (int*)malloc(sizeof(int) * 4);
    for (int i = 0; i < 4; i++)
        p[i] = i;
    return 0;
}
```

C Common Mistakes

```c
#include <stdio.h>

int main()
{
    int* p = (int*)malloc(sizeof(int) * 4);
    int i = 0;
    for (i = 0; i < 4; i++)
        p[i] = i;
    return 0;
}
```

C Common Mistakes

```
#include <stdio.h>

int main()
{
    int* p = (int*)malloc(sizeof(int) * 4);
    int i;
    for (i = 0; i < 4; i++)
        p[i] += 100;
    return 0;
}
```

C Common Mistakes

```
#include <stdio.h>

int main()
{
    int* p = (int*)calloc(4, sizeof(int));
    int i;
    for (i = 0; i < 4; i++)
        p[i] += 100;
    return 0;
}
```

C Common Mistakes

```
void f()
{
   int* ptr = (int*)malloc(sizeof(int));
        /* Do some work */
   return;
}
```

C Common Mistakes

```
void f()
{
   int* ptr = (int*)malloc(sizeof(int));
        /* Do some work */
   free(ptr);
   return;
}
```

C Common Mistakes

```c
#include <stdlib.h>
int fun()
{
    int x = 10;
    return &x;
}

int main()
{
    int* p = fun();
    *p = 20;
    return 0;
}
```

C Common Mistakes

```c
#include <stdio.h>
void printArray(int arr[])
{
    int i;
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    for (i = 0; i < arr_size; i++) {
        printf("%d ", arr[i]);
    }
}
int main()
{
    int arr[4] = { 1, 2, 3, 4};
    printArray(arr);
    return 0;
}
```

C Common Mistakes

```c
#include <stdio.h>
void printArray(int arr[], int arr_size)
{
    int i;
    for (i = 0; i < arr_size; i++) {
        printf("%d ", arr[i]);
    }
}
int main()
{
    int arr[4] = { 1, 2, 3, 4};
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    printArray(arr, arr_size);
    return 0;
}
```

Tips and Techniques

- Start off with a working algorithm
- Incremental coding/test early
- Simplify the problem
- Explain the bug to someone else
- Fix bugs as you find them
- Recognize common bugs (such as using ‘=’ instead of ‘==’, using ‘==’ instead of equals( ), dereferencing null, etc.)
- Recompile everything
- Test boundaries
- Test exceptional conditions
- Take a break

Backup: Algorithm for Building FIRST sets

- For each X as a terminal, then FIRST(X) is {X}
- If X ::= ε, then ε ∈ FIRST(X)
- For each X as a non-terminal, initialize FIRST(X) to ∅
- ***Iterate until*** no more terminals or ε can be added to any FIRST(X):
  For each rule in the grammar of the form $X ::= Y_1 Y_2 \ldots Y_k$
  add a to FIRST(X) if $a \in FIRST(Y_1)$
  add a to FIRST(X) if $a \in FIRST(Y_i)$ and $\varepsilon \in FIRST(Y_j)$
  for all $1 \leq j \leq i\text{-}1$ and $i \geq 2$
  add ε to FIRST(X) if $\varepsilon \in FIRST(Y_i)$ for all $1 \leq i \leq k$
  EndFor
  End iterate

Backup: Algorithm for Building FOLLOW sets

To Build FOLLOW(X) for non-terminal X:

- Place EOF in FOLLOW(<start>)
- For each X as a non-terminal, initialize FOLLOW(X) to ∅

Iterate until no more terminals can be added to any FOLLOW(X):

For each rule *p* in the grammar

If p is of the form A ::= αBβ, then

if ε ∈ *FIRST*(β)

Place {FIRST(β) - ε, FOLLOW(A)} in FOLLOW(B)

else

Place {FIRST(β)} in FOLLOW(B)

If p is of the form A ::= αB, then

Place FOLLOW(A) in FOLLOW(B)

End iterate