Class Announcements

- Third homework has been posted. No deadline extension possible. Due Tuesday, 21.
- Midterm 1: Friday, February 24, in class, closed book and notes

Recursive Descent Parsing

Now, we can produce a simple recursive descent parser for an LL(1) grammar.

Recursive descent is one of the simplest parsing techniques used in practical compilers:

- Each non–terminal has an associated parsing procedure that can recognize any sequence of tokens generated by that non–terminal.
- There is a main routine to initialize all **globals** (e.g.: `token`) and call the start symbol. On return, check whether `token` == eof, and whether errors occurred. (Note: left-to-right evaluation of expressions).
- Within a parsing procedure, both non–terminals and terminals can be "matched":
  - non–terminal A — call parsing procedure for A
  - token t — compare t with current input token; if match, consume input, otherwise ERROR
- Parsing procedures may contain code that performs some useful "computation" (syntax directed translation).

Syntax Directed Translation

Examples:

1. Interpreter
2. Code generator
3. Type checker
4. Performance estimator

Use hand-written recursive descent LL(1) parser

Syntax-Directed Translation Skeleton

<expr> ::= + <expr> <expr> |

<digit>

<digit> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```
...   expr:

  switch token {
    case +:    token := next_token( );
               /*1*/ expr( ); /*2*/ expr( ); /*3*/
               return;
    case 0..9: /*4*/ return digit( );
  }

...   digit:
  switch token {
    case 1:    token := next_token( );
               /*5*/
               return ;
    case 2:    token := next_token( ); /*6*/ return;
    ...
  }
```

This skeleton code implements a tree walk over the parse tree. Define return values and put code where you need it.

Example: Interpreter

<expr> ::= + <expr> <expr> |

<digit>

<digit> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```
int expr:  // returns value of expression
  int val1, val2; // values
  switch token {
    case +:    token := next_token( );
               val1 = expr( ); val2 = expr( );
               return val1+val2;
    case 0..9: return digit( );
  }

int digit:  // returns value of constant
  switch token {
    case 1:    token := next_token( );
               return 1;
    case 2:    token := next_token( );
               return 2;
    ...
  }
```

Example: Interpreter

What happens when you parse subprogram

"**+ 2 + 1 2**" ?

The parsing produces:

```
5
```

Example: Simple Code Generation

<expr> ::= + <expr> <expr> |

<digit>

<digit> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```
int expr:  // returns target register of operation
  int target_reg; // ‘‘fresh’’ register
  int reg1, reg2; // other registers
  switch token {
    case +:    token := next_token( );
               target_reg = next_register( );
               reg1 = expr( ); reg2 = expr( );
               CodeGen(ADD, reg1, reg2, target_reg);
               return target_reg;
    case 0..9: return digit( );
  }

int digit:  // returns target register of operation
  int target_reg = next_register( ); // ‘‘fresh’’ register
  switch token {
    case 1:    token := next_token( );
               CodeGen(LOADI, 1, target_reg);
               return target_reg;
    case 2:    token := next_token( );
               CodeGen(LOADI, 2, target_reg);
               return target_reg;
    ...
  }
```

Example: Simple Code Generation

What happens when you parse subprogram

"**+ 2 + 1 2**" ?

Assumption:

first call to `next_register( )` will return 1

The parsing produces:

```
loadI 2 => r2
loadI 1 => r4
loadI 2 => r5
add r4, r5 => r3
add r2, r3 => r1
```

Example: Simple Type Checker

<expr> ::= + <expr> <expr> |

<digit>

<digit> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```
string expr:  // returns type expression
  string type1, type2; // other type expressions
  switch token {
    case +:    token := next_token( );
               type1 = expr( ); type2 = expr( );
               if (type1 == ``int'' and type2 == ``int'')
                    return ``int''
               else
                    return ``error'';
    case 0..9: return digit( );
  }

string digit:  // returns type expression
  switch token {
    case 1:    token := next_token( );
               return ``int'';
    case 2:    token := next_token( );
               return ``int'';
    ...
  }
```

Example: Simple Type Checker

What happens when you parse subprogram

"**+ 2 + 1 2**" ?

The parsing produces:

```
``int''
```

Example: Basic Performance Predictor

<expr> ::= + <expr> <expr> |

<digit>

<digit> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```
int expr:  // returns cycles needed to compute expression
  int cyc1, cyc2; // subexpression cycles
  switch token {
    case +:    token := next_token( );
               cyc1 = expr( ); cyc2 = expr( );
               return cyc1+cyc2+2 // ADD takes 2 cycles;
    case 0..9: return digit( );
  }

int digit:  // returns cycles
  switch token {
    case 1:    token := next_token( );
               return 1; // LOADI takes 1 cycle
    case 2:    token := next_token( );
               return 1; // LOADI takes 1 cycle
    ...
  }
```

Example: Basic Performance Predictor

What happens when you parse subprogram

"**+ 2 + 1 2**" ?

The parsing produces:

7

Project1: tinyL Language

<program> ::= <stmtlist> .

<stmtlist> ::= <stmt> <morestmts>

<morestmts> ::= ; <stmtlist> |ϵ

<stmt> ::= <assign> |<print>

<assign> ::= <variable> = <expr>

<print> ::= # <variable>

<expr> ::= + <expr> <expr> |
- <expr> <expr> |
* <expr> <expr> |
/ <expr> <expr> |
<variable> |
<digit>

<variable> :: = a | b | c | d | e | f | g | h | i | j | k | s | t | u

<digit> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Example program:

a=3;b=5;c=/3*ab;d=+c1;#d.

Next Lecture

Things to do:

Learn C if you don't know it already.

Read Scott: Chap. 3.1 - 3.4; 8.3 ; ALSU Chap. 7.1 - 7.3

Next time:

- Introduction to imperative programming (C)
- Dynamic vs. static scoping
- Runtime environment
- access links and control links management