Goals

• understand the structure of a compiler
• understand how the components operate
• understand the tools involved
  • scanner generator, parser generator, etc.

• understanding means
  • [theory] be able to read source code
  • [practice] be able to adapt/write source code

Format: “werkcollege” + practicum

• 14 x 2 hours of interactive lectures 1 sp
  • book “Modern Compiler Design”
  • schedule: see blackboard
  • handouts: see blackboard
• assignment 2 sp
  • groups of 2 students
  • modify reference compiler
• oral exam 1 sp

What is a compiler?

program in some source language
  front-end analysis
  semantic representation
  back-end synthesis
  executable code for target machine

Why study compiler construction?

• curiosity
• better understanding of programming language concepts
• wide applicability
  • transforming “data” is very common
  • many useful data structures and algorithms
• practical application of “theory”

Overview lecture 1

• [introduction]
• compiler structure
  • exercise

------------------ 15 min. break ------------------

• lexical analysis
• exercise
Compiler structure

Semantic representation

Limitations of modular approach

AST example

Symbolic representation

parse tree: \( b^2 - 4ac \)

AST: \( b^2 - 4ac \)
**Compiler construction 2002**

### annotated AST: \( b^2 - 4a^2c \)

- \( b \) \( \Rightarrow \) identifier, type: real, loc: reg1
- \( b \) \( \Rightarrow \) constant, type: real, loc: sp+16
- \( c \) \( \Rightarrow \) term, type: real, loc: sp+24
- \( b \) \( \Rightarrow \) expression, type: real, loc: reg2
- \( 4 \) \( \Rightarrow \) constant, type: real, loc: const
- \( a \) \( \Rightarrow \) factor, type: real, loc: sp+8
- \( \ast \) \( \Rightarrow \) term, type: real, loc: reg1
- \( \ast \) \( \Rightarrow \) factor, type: real, loc: sp+8

### AST exercise (5 min.)

- expression grammar
  - \( \text{expression} \rightarrow \text{expression} + \text{term} \mid \text{expression} - \text{term} \mid \text{term} \)
  - \( \text{term} \rightarrow \text{term} \ast \text{factor} \mid \text{term} / \text{factor} \mid \text{factor} \)
  - \( \text{factor} \rightarrow \text{identifier} \mid \text{constant} \mid (\text{expression}) \)
- example expression
  - \( b^2 - (4a^2c) \)
- draw parse tree and AST

### Front-end: from program text to AST

- covert stream of **characters** to stream of **tokens**
- what is a token?
  - sequence of characters with a semantic notion, see language definition
  - rule of thumb: two characters belong to the same token if inserting white space changes the meaning.

```c
digit = *ptr++ - '0';
digit = *ptr+ + '0';
```

### Tokens

- attributes
  - type
  - lexeme
  - value
  - file position

```c
typedef struct {
    int class;
    char *repr;
    file_pos position;
  } Token_Type;
```

- examples
  - IDENTIFIER: foo, t3, ptr
  - NUMBER: 15, 082, 666
  - REAL: 1.2, .002, 1e6
Non-tokens

- white spaces
  spaces, tabs, newlines
- comments
  /* a C-style comment */
  // a C++ comment
- preprocessor directives
  #include "lex.h"
  #define is_digit(d) ('0' <= (d) && (d) <= '9')

Regular expressions

Basic patterns
- the character x
- any character, usually except a newline
- any of the characters a,b,c and the range A-Z

Repetition operators
- an R or nothing (= optionally an R)
- zero or more occurrences of R
- one or more occurrences of R

Composition operators
- an R1 followed by an R2
- either an R1 or an R2

Grouping
- R itself

Examples of regular expressions

- an integer is a sequence of digits:
  [0-9]+

- an identifier is a sequence of letters and digits; the first character must be a letter:
  [a-z][a-z0-9]*

Regular descriptions

- structuring regular expressions by introducing named sub expressions
  letter → [a-zA-Z]
  digit → [0-9]
  letter_or_digit → letter | digit
  identifier → letter letter_or_digit*

- define before use

Exercise (5 min.)

- write down regular descriptions for the following descriptions:
  - an integral number is a non-zero sequence of digits optionally followed by a letter denoting the base class (b for binary and o for octal).
  - a fixed-point number is an (optional) sequence of digits followed by a dot (\'.\') followed by a sequence of digits.
  - an identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter, but may not be used as the first or last character.

Answers
**Lexical analysis**

- covert stream of characters to stream of tokens
- tokens are defined by a regular description
- tokens are demanded one-by-one by the syntax analyzer

**interface**

```c
extern Token_Type Token;
/* Global variable that holds the current token. */

void start_lex(void);
/* Must be called before the first call to */
/* get_next_token(). */

void get_next_token(void);
/* Load the next token into the global */
/* variable Token. */
```

**lexical analysis by hand**

- read complete program text into memory for simplicity
- avoids buffering and arbitrary limits
- variable length tokens
- `get_next_token()` dispatches on the next character

```c
int main() { printf( "hello world\n");

void get_next_token(void) {
    int start_dot;
    skip_layout_and_comment();
    /* now we are at the start of a token or at end-of-file, so: */
    note_token_position();
    /* split on first character of the token */
    if (is_end_of_input(input_char)) {
        Token.class = EoF; Token.repr = "<EoF>": return;
    }
    if (is_letter(input_char)) {recognize_identifier();}
    else if (is_digit(input_char)) {recognize_integer();}
    else if (is_operator(input_char) || is_separator(input_char)) {
        Token.class = input_char; next_char();
    }
    else {Token.class = ERRONEOUS; next_char();}
    Token.repr = input_to_zstring(start_dot, dot-start_dot);
}
```

**Character classification & token recognition**

```c
#define is_end_of_input(ch)     ((ch) == '\0')
#define is_layout(ch)           (!is_end_of_input(ch) && (ch) <= ' ')
#define is_uc_letter(ch)        ('A' <= (ch) && (ch) <= 'Z')
#define is_lc_letter(ch)        ('a' <= (ch) && (ch) <= 'z')
#define is_letter(ch)           (is_uc_letter(ch) || is_lc_letter(ch))
#define is_digit(ch)            ('0' <= (ch) && (ch) <= '9')
#define is_letter_or_digit(ch)  (is_letter(ch) || is_digit(ch))
#define is_underscore(ch)       ((ch) == '_')
#define is_operator(ch)         (strchr("+-*/", (ch)) != NULL)
#define is_separator(ch)        (strchr(";,,(){}", (ch)) != NULL)

void recognize_integer(void) {
    if (is_digit(input_char)) (next_char());
    while (is_digit(input_char)) (next_char());
}
```

**Summary**

- compiler is a structured toolbox
  - front-end: program text → annotated AST
  - back-end: annotated AST → executable code
- lexical analysis: program text → tokens
  - token specifications
  - implementation by hand
- exercises
  - AST
  - regular descriptions
Homework

- find a partner for the “practicum”
- register your group
  - send e-mail to koen@pds.twi.tudelft.nl
- print handout lecture 2 [blackboard]