Compilerbau

Stephan Schulz
stephan.schulz@dhbw-stuttgart.de

Scanner
Parser
AST
Semantik
Optimierung
Code-Generierung
Table of Contents

Preliminaries

Introduction

Refresher: Flex and Bison

Exercise 1: Scientific Calculator

Syntax Analysis
  Example Language: nanoLang
  Grammars and Derivations
  Semantic Analysis

Runtime Environment and Code Generation
Stephan Schulz

- Dipl.-Inform., U. Kaiserslautern, 1995
- Dr. rer. nat., TU München, 2000
- Visiting professor, U. Miami, 2002
- Visiting professor, U. West Indies, 2005
- Visiting lecturer (Hildesheim, Offenburg, ...) seit 2009
- Industry experience: Building Air Traffic Control systems
  - System engineer, 2005
  - Project manager, 2007
  - Product Manager, 2013
- Professor, DHBW Stuttgart, 2014

Research: Logic & Deduction
Goals for Today

- Practical issues
- Programming language survey
- Execution of languages
- Low-level code vs. high-level code
- Structure of a Compiler
- Refresher
  - Grammars
  - Flex/Bison
- Programming exercises
  - Scientific calculator revisited
This Course in Context

- **Formal languages and automata**
  - Basic theory - languages and automata
  - General grammars
  - Abstract parsing
  - Computability

  Focus on foundations

- **Compiler construction**
  - Advanced theory - parsers and languages
  - Tools and their use
  - Writing parsers and scanners
  - Code generation and run times

  Focus on practical applications
Practical issues

- Lecture time: Wednesdays, 12:30-16:45
  - Lecture (with exercises): 12:30-14:45
  - Lab: 15:00-16:45
  - Breaks will be somewhat flexible
  - No lecture on March 25th (I’m snowboarding)

- Grading:
  - Lecture Compilerbau: Written Exam, grade averaged with Formal Languages&Automata for module grade
  - Lab: Pass/Fail based on success in exercises
For practical exercises, you will need a complete Linux/UNIX environment. If you do not run one natively, there are several options:

- You can install VirtualBox (https://www.virtualbox.org) and then install e.g. Ubuntu (http://www.ubuntu.com/) on a virtual machine. Make sure to install the Guest Additions
- For Windows, you can install the complete UNIX emulation package Cygwin from http://cygwin.com
- For MacOS, you can install fink (http://fink.sourceforge.net/) or MacPorts (https://www.macports.org/) and the necessary tools

You will need at least flex, bison, gcc, grep, sed, AWK, make, and a good text editor
Resources

- Course web page

- Literature
  - Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools*
  - Kenneth C. Louden: *Compiler Construction - Principles and Practice*
  - Ulrich Hedtstück: *Einführung in die theoretische Informatik*
Exercise: Programming Languages

- Name and describe several modern programming languages!
Modern Programming Languages

Desirable properties of high-level languages

- Expressive and flexible
  - Close to application domains
  - Good abstractions
  - Powerful constructs
  - Readable

- Compact
  - Programmer productivity depends on length (!)

- Machine independent
  - Code should run on many platforms
  - Code should run on evolving platforms

- Strong error-checking
  - Static
  - Dynamic

- Efficiently executable
Low-Level Code

- Machine code
  - Binary
  - Machine-specific
  - Operations (and operands) encoded in instruction words
  - Basic operations only
  - Manipulates finite number of registers
  - Direct access to memory locations
  - Flow control via conditional and unconditional jumps (think goto)
  - Basic data types (bytes, words)

  **Directly executable by processor**

- Assembly languages
  - Textual representation of machine code
  - Symbolic names for operations and operands
  - Labels for addresses (code and data)

  **Direct one-to-one mapping to machine code**
Exercise: Low-Level Code – Minimal C

- **Predefined global variables**
  - Integers R0, R1, R2, R3, R4
  - Integer array mem[MAXMEM]
  - No new variables allowed

- No parameters (or return) for functions

- **Flow control:** Only `if` and `goto` (not `while`, `for`, ...)
  - No blocks after `if` (only one command allowed)

- **Arithmetic only between R0, R1, R2, R3, R4**
  - Result must be stored in one of R0, R1, R2, R3, R4
  - Operands: Only R0, R1, R2, R3, R4 allowed (no nested sub-expressions)
  - Unary increment/decrement is ok (R0++)
  - R0, R1, R2, R3, R4 can be stored in/loaded from mem, indexed with a fixed address or one of the variables.
Exercise: Minimal C Example

/* Compute sum from 0 to R0, return result in R1 */

void user_code(void)
{
    /* R0 is the input value and limit */
    R1 = 0;  /* Sum, value returned */
    R2 = 0;  /* Loop counter */
    R3 = 1;  /* For increments */

    loop:
        if (R2 > R0)
            goto end;
        R1 = R1+R2;
        R2 = R2+R3;
        goto loop;
    end:
        return;
}
Exercise: Low-Level Code

- Write (in Minimal C) the following functions:
  - A program computing the factorial of R0
  - A program computing the Fibonacci-number of R0 iteratively
  - A program computing the Fibonacci-number of R0 recursively

- You can find a frame for your code at the course web page, http://wwwlehre.dhbw-stuttgart.de/~sschulz/cb2015.html
Surprise!

Computers don’t execute high-level languages (directly)!
Execution of high-level programs

Compiled languages

Interpreted languages

Development Time

Run Time
Compilers translate high-level languages into low-level code!
Most computer languages are mostly context-free

Regular: vocabulary

- Keywords, operators, identifiers
- Described by regular expressions or regular grammar
- Handled by (generated or hand-written) scanner/tokenizer/lexer

Context-free: program structure

- Matching parenthesis, block structure, algebraic expressions, ...
- Described by context-free grammar
- Handled by (generated or hand-written) parser

Context-sensitive: e.g. declarations

- Described by human-readable constraints
- Handled in an ad-hoc fashion (e.g. symbol table)
High-Level Architecture of a Compiler

Source handler

Sequence of characters:
i,n,t, \texttt{, }, a,, b,, ;,, a, =, b, +,, 1, ;

Lexical analysis
(tokeniser)

Sequence of tokens:
(id, “int”), (id, “a”), (id, “b”), (semicolon), (id, “a”), (eq), (id, “b”), (plus), (int, “1”), (semicolon)

Syntactic analysis
(parser)

e.g. Abstract syntax tree

Semantic analysis

e.g. AST+symbol table

Code generation
(several optimisation passes)

e.g. assembler code

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
</tr>
<tr>
<td>b</td>
<td>int</td>
</tr>
</tbody>
</table>

ld a,b
ld c, 1
add c
...

ld a,b
ld c, 1
add c
...
Source Handler

- Handles input files
- Provides character-by-character access
- May maintain file/line/column (for error messages)
- May provide look-ahead

Result: Sequence of characters (with positions)
Lexical Analysis/Scanning

- Breaks program into token
- Typical tokens:
  - Reserved word (if, while)
  - Identifier (i, database)
  - Symbols (\{, \}, (, ), +, -, ...)

Result: Sequence of tokens
```c
int main(int argc, char* argv[])
{
    R0 = 0;
    R1 = 0;
    R2 = 0;
    R3 = 1;
    R4 = 1;
    for (int i = 0; i < MAXMEM; i++)
    {
        mem[i] = 0;
    }

    user_code();

    return 0;
}
```
Automatisation with Flex

```
ld a,b
ld c, 1
add c
...
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
</tr>
<tr>
<td>b</td>
<td>int</td>
</tr>
</tbody>
</table>

Sequence of characters:
i, n, t, *, a, ., b, ., a, =, b, +, 1, .

Sequence of tokens:
(id, "int"), (id, "a"), (id, "b"), (semicolon), (id, "a"), (eq), (id, "b"), (plus), (int, "1"), (semicolon)
Syntactical Analysis/Parsing

- Description of the language with a **context-free grammar**
- Parsing:
  - Try to build a *parse tree*/abstract syntax tree (AST)
  - Parse tree unambiguously describes structure of a program
  - AST reflects abstract syntax (can e.g. drop parenthesis)
- Methods:
  - Manual recursive descent parser
  - Automatic with a table-driven bottom-up parser

**Result:** Abstract Syntax Tree
Automatisation with Bison

Flex

Source handler

Sequence of characters:
i,n,t, ⇑, a,, b, ;, a, =, b, +, 1, ;

Lexical analysis
(tokeniser)

Sequence of tokens:
(id, "int"), (id, "a"), (id, "b"), (semicolon), (id, "a"), (eq), (id, "b"), (plus), (int, "1"), (semicolon)

Bison

Syntactic analysis
(parser)

e.g. Abstract syntax tree

Semantic analysis

e.g. AST+symbol table

Code generation
(several optimisation passes)

e.g. assembler code

Variable | Type
--- | ---
| a | int
| b | int

ld a,b
ld c, i
add c
...

ld a,b
ld c, i
add c
...

ld a,b
ld c, i
add c
...
Semantic Analysis

- Analyze static properties of the program
  - Which variable has which type?
  - Are all expressions well-typed?
  - Which names are defined?
  - Which names are referenced?

- Core tool: Symbol table

**Result:** Annotated AST
Optimization

- Transform Abstract Syntax Tree to generate better code
  - Smaller
  - Faster
  - Both

- Mechanisms
  - Common sub-expression elimination
  - Loop unrolling
  - Dead code/data elimination
  - ...
Code Generation

- Convert optimized AST into low-level code
- Target languages:
  - Assembly code
  - Machine code
  - VM code (z.B. JAVA byte-code, p-Code)
  - C (as a “portable assembler”)
  - ...

**Result:** Program in target language
Refresher: Flex
Flex Overview

- **Flex** is a **scanner generator**
- **Input:** Specification of a regular language and what to do with it
  - Definitions - named regular expressions
  - Rules - patterns+actions
  - (miscellaneous support code)
- **Output:** Source code of **scanner**
  - Scans input for patterns
  - Executes associated actions
  - Default action: Copy input to output
  - Interface for higher-level processing: `yylex()` function
Flex Overview

Development time

Definitions
Rules
Miscellaneous code

flex+gcc

Execution time

Input

scanner

Tokenized/processed output
Flex Example Task

- (Artificial) goal: Sum up all numbers in a file, separately for ints and floats
- Given: A file with numbers and commands
  - Ints: Non-empty sequences of digits
  - Floats: Non-empty sequences of digits, followed by decimal dot, followed by (potentially empty) sequence of digits
  - Command print: Print current sums
  - Command reset: Reset sums to 0.
- At end of file, print sums
<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 3.1415</td>
<td>int: 12 (&quot;12&quot;)</td>
</tr>
<tr>
<td>0.33333</td>
<td>float: 3.141500 (&quot;3.1415&quot;)</td>
</tr>
<tr>
<td>print reset</td>
<td>float: 0.333330 (&quot;0.33333&quot;)</td>
</tr>
<tr>
<td>2 11</td>
<td>Current: 12 : 3.474830</td>
</tr>
<tr>
<td>1.5 2.5 print</td>
<td>Reset</td>
</tr>
<tr>
<td>1</td>
<td>int: 2 (&quot;2&quot;)</td>
</tr>
<tr>
<td>print 1.0</td>
<td>int: 11 (&quot;11&quot;)</td>
</tr>
<tr>
<td></td>
<td>float: 1.500000 (&quot;1.5&quot;)</td>
</tr>
<tr>
<td></td>
<td>float: 2.500000 (&quot;2.5&quot;)</td>
</tr>
<tr>
<td></td>
<td>Current: 13 : 4.000000</td>
</tr>
<tr>
<td></td>
<td>int: 1 (&quot;1&quot;)</td>
</tr>
<tr>
<td></td>
<td>Current: 14 : 4.000000</td>
</tr>
<tr>
<td></td>
<td>float: 1.000000 (&quot;1.0&quot;)</td>
</tr>
<tr>
<td></td>
<td>Final 14 : 5.000000</td>
</tr>
</tbody>
</table>
Basic Structure of Flex Files

- Flex files have 3 sections
  - Definitions
  - Rules
  - User Code
- Sections are separated by `%%`
- Flex files traditionally use the suffix `.l`
%%option noyywrap

DIGIT [0-9]

{%
    int intval = 0;
    double floatval = 0.0;
%

%%
Example Code (rule section)

```c
{DIGIT}+ {  
    printf( "int: %d ("%s")\n", atoi(yytext), yytext );
    intval += atoi(yytext);
}
{DIGIT}+"."{DIGIT}* {  
    printf( "float: %f ("%s")\n", atof(yytext),yytext );
    floatval += atof(yytext);
}
reset {  
    intval = 0;
    floatval = 0;
    printf("Reset\n");
}
print {  
    printf("Current: %d : %f\n", intval, floatval);
}
\n|. {  
    /* Skip */
}
```
```c
int main( int argc, char **argv )
{
    ++argv, --argc; /* skip over program name */
    if ( argc > 0 )
        yyin = fopen( argv[0], "r" );
    else
        yyin = stdin;

    yylex();

    printf("Final %d : %f\n", intval, floatval);
}
```
Generating a scanner

```bash
> flex -t numbers.l > numbers.c
> gcc -c -o numbers.o numbers.c
> gcc numbers.o -o scan_numbers
> ./scan_numbers Numbers.txt
```

```plaintext
int:  12 ("12")
float: 3.141500 ("3.1415")
float: 0.333330 ("0.33333")
Current: 12 : 3.474830
Reset
int:   2 ("2")
int:   11 ("11")
float: 1.500000 ("1.5")
float: 2.500000 ("2.5")
...
```
Flexing in detail

> flex -tv numbers.l > numbers.c
scanner options: -tvI8 -Cem
37/2000 NFA states
18/1000 DFA states (50 words)
5 rules
Compressed tables always back-up
1/40 start conditions
20 epsilon states, 11 double epsilon states
6/100 character classes needed 31/500 words of storage, 0 reused
36 state/nextstate pairs created
24/12 unique/duplicate transitions
...
381 total table entries needed
Definition Section

- Can contain `flex` options
- Can contain (C) initialization code
  - Typically `#include()` directives
  - Global variable definitions
  - Macros and type definitions
  - Initialization code is embedded in `%{ and %}
- Can contain definitions of regular expressions
  - Format: `NAME RE`
  - Defined NAMES can be referenced later
%%option noyywrap

DIGIT [0-9]

{%
    int intval = 0;
    double floatval = 0.0;
%

%%%
Rule Section

- This is the core of the scanner!
- Rules have the form PATTERN ACTION
- Patterns are regular expressions
  - Typically use previous definitions
- THERE IS WHITE SPACE BETWEEN PATTERN AND ACTION!
- Actions are C code
  - Can be embedded in { and }
  - Can be simple C statements
  - For a token-by-token scanner, must include return statement
  - Inside the action, the variable yytext contains the text matched by the pattern
  - Otherwise: Full input file is processed
Example Code (rule section) (revisited)

```c
{DIGIT}+    {
    printf( "int:  %d ("%s")\n", atoi(yytext), yytext );
    intval += atoi(yytext);
}
{DIGIT}+"."{DIGIT}*    {
    printf( "float: %f ("%s")\n", atof(yytext),yytext );
    floatval += atof(yytext);
}
reset {
    intval = 0;
    floatval = 0;
    printf("Reset\n");
}
print {
    printf("Current: %d : %f\n", intval, floatval);
}
\n|. { /* Skip */
```
- Can contain all kinds of code
- For stand-alone scanner: must include `main()`
- In `main()`, the function `yylex()` will invoke the scanner
- `yylex()` will read data from the file pointer `yyin` (so `main()` must set it up reasonably)
```c
int main( int argc, char **argv )
{
    ++argv, --argc; /* skip over program name */
    if ( argc > 0 )
        yyin = fopen( argv[0], "r" );
    else
        yyin = stdin;

    yylex();

    printf("Final %d : %f\n", intval, floatval);
}
```
A comment on comments

- Comments in Flex are complicated
  - ...because nearly everything can be a pattern
- Simple rules:
  - Use old-style C comments /* This is a comment */
  - Never start them in the first column
  - Comments are copied into the generated code
  - Read the manual if you want the dirty details
Flex Miscellany

▶ Flex online:
  ▶ http://flex.sourceforge.net/
  ▶ REs: http://flex.sourceforge.net/manual/Patterns.html

▶ make knows flex
  ▶ Make will automatically generate file.o from file.l
  ▶ Be sure to set LEX=flex to enable flex extensions
  ▶ Makefile example:

    LEX=flex
    all: scan_numbers
    numbers.o: numbers.l

    scan_numbers: numbers.o
      gcc numbers.o -o scan_numbers
Refresher: Bison
YACC/Bison

- Yacc - Yet Another Compiler Compiler
  - Originally written ≈1971 by Stephen C. Johnson at AT&T
  - LALR parser generator
  - Translates grammar into syntax analyzer

- GNU Bison
  - Written by Robert Corbett in 1988
  - Yacc-compatibility by Richard Stallman
  - Output languages now C, C++, Java

- Yacc, Bison, BYacc, ... mostly compatible (POSIX P1003.2)
By default, Bison constructs a 1 token Look-Ahead Left-to-right Rightmost-derivation or LALR(1) parser

- Input tokens are processed left-to-right
- Shift-reduce parser:
  - Stack holds tokens (terminals) and non-terminals
  - Tokens are shifted from input to stack. If the top of the stack contains symbols that represent the right hand side (RHS) of a grammar rule, the content is reduced to the LHS
  - Since input is reduced left-to-right, this corresponds to a rightmost derivation
  - Ambiguities are solved via look-ahead and special rules
  - If input can be reduced to start symbol, success!
  - Error otherwise

LALR(1) is efficient in time and memory

- Can parse “all reasonable languages”
- For unreasonable languages, Bison (but not Yacc) can also construct GLR (General LR) parsers
  - Try all possibilities with back-tracking
  - Corresponds to the non-determinism of stack machines
Yacc/Bison Overview

- Bison reads a specification file and converts it into (C) code of a parser
- Specification file: Definitions, grammar rules with actions, support code
  - Definitions: Token names, associated values, includes, declarations
  - Grammar rules: Non-terminal with alternatives, action associated with each alternative
  - Support code: e.g. `main()` function, error handling...
  - Syntax similar to (F)lex
    - Sections separated by `%%`
    - Special commands start with `%`
- Bison generates function `yyparse()`
- Bison needs function `yylex()`
  - Usually provided via (F)lex
Yacc/Bison workflow

Development time

Bison Input File
<file>.y

Definitions file
<file>.tab.h

Parser Source
<file>.tab.c

Flex Input file
<file>.l

Definitions file
<file>.tab.h

Parser Source
<file>.tab.c

Flex

Lexical Source
<file>.c

Lexer object
<file>.o

gcc

Parser object
<file>.tab.o

linker (gcc)

Execution time

Some input to process

Final executable parser

Some output produced
Example task: Desk calculator

- Desk calculator
  - Reads algebraic expressions and assignments
  - Prints result of expressions
  - Can store values in registers R0-R99

Example session:

```
[Shell] ./scicalc
R10=3*(5+4)
  > RegVal: 27.000000
(3.1415*R10+3)
  > 87.820500
R9=(3.1415*R10+3)
  > RegVal: 87.820500
R9+R10
  > 114.820500
...
```
Abstract grammar for desk calculator (partial)

\[ G_{DC} = \langle V_N, V_T, P, S \rangle \]

- \( V_T = \{ \text{PLUS, MULT, ASSIGN, OPENPAR, CLOSEPAR, REGISTER, FLOAT, …} \} \)
  - Some terminals are single characters (+, =, …)
  - Others are complex: R10, 1.3e7
  - Terminals (“tokens”) are generated by the lexer

- \( V_N = \{ \text{stmt, assign, expr, term, factor, …} \} \)

- \( P: \)
  - \( \text{stmt} \rightarrow \text{assign} \)
    - \( \text{expr} \)
  - \( \text{assign} \rightarrow \text{REGISTER ASSIGN expr} \)
  - \( \text{expr} \rightarrow \text{expr PLUS term} \)
    - \( \text{term} \)
  - \( \text{term} \rightarrow \text{term MULT factor} \)
    - \( \text{factor} \)
  - \( \text{factor} \rightarrow \text{REGISTER} \)
    - \( \text{FLOAT} \)
    - \( \text{OPENPAR expr CLOSEPAR} \)

- \( S = \text{*handwave*} \)
  - For a single statement, \( S = \text{stmt} \)
  - In practice, we need to handle sequences of statements and empty input lines (not reflected in the grammar)
Lexer interface

- Bison parser requires `yylex()` function
- `yylex()` returns `token`
  - Token text is defined by regular expression pattern
  - Tokens are encoded as integers
  - Symbolic names for tokens are defined by Bison in generated header file
    - By convention: Token names are all CAPITALS
- `yylex()` provides optional `semantic value` of token
  - Stored in global variable `yy1val`
  - Type of `yy1val` defined by Bison in generated header file
    - Default is `int`
    - For more complex situations often a `union`
    - For our example: Union of double (for floating point values) and integer (for register numbers)
Lexer for a minimal "scientific" calculator.

Copyright 2014 by Stephan Schulz, schulz@eprover.org.

This code is released under the GNU General Public Licence Version 2.

*/

%option noyywrap

{%
    #include "scicalcparse.tab.h"
%

Lexer for desk calculator (2)

DIGIT    [0-9]
INT      {DIGIT}+
PLAINFLOT {INT}|{INT}[.]{INT}|{INT}.{INT}|{INT}.{INT}
EXP      [eE](\+|-)?{INT}
NUMBER   {PLAINFLOT}{EXP}?
REG      R{DIGIT}{DIGIT}?

%%

"*" {return MULT;}
"+" {return PLUS;}
"=" {return ASSIGN;}
"(" {return OPENPAR;}
")" {return CLOSEPAR;}
\n {return NEWLINE;}
{REG} { 
    yylval.regno = atoi(yytext+1);
    return REGISTER;
}

{NUMBER} { 
    yylval.val = atof(yytext);
    return FLOAT;
}

[ ] { /* Skip whitespace*/ }

/* Everything else is an invalid character. */ . { return ERROR;}

%%
Data model of desk calculator

- Desk calculator has simple state
  - 100 floating point registers
  - R0-R99
- Represented in C as array of doubles:
  ```c
  #define MAXREGS 100
  
  double regfile[MAXREGS];
  ```
- Needs to be initialized in support code!
Bison code for desk calculator: Header

{%
#include <stdio.h>

#define MAXREGS 100

double regfile[MAXREGS];

extern int yyerror(char* err);
extern int yylex(void);
%
%
union {
   double val;
   int regno;
}
%}
Bison code for desk calculator: Tokens

```
%start stmtseq

%left PLUS
%left MULT
%token ASSIGN
%token OPENPAR
%token CLOSEPAR
%token NEWLINE
%token REGISTER
%token FLOAT
%token ERROR

```
Actions in Bison

- Bison is based on syntax rules with associated actions
  - Whenever a `reduce` is performed, the action associated with the rule is executed
- Actions can be arbitrary C code
- Frequent: `semantic actions`
  - The action sets a `semantic value` based on the semantic value of the symbols reduced by the rule
  - For terminal symbols: Semantic value is `yy1val` from Flex
  - Semantic actions have “historically valuable” syntax
    - Value of reduced symbol: `$$`
    - Value of first symbol in syntax rule body: `$1`
    - Value of second symbol in syntax rule body: `$2`
    - ...`
    - Access to named components: `$<val>1`
```plaintext
stmtseq: /* Empty */
   | NEWLINE stmtseq {}  
   | stmt  NEWLINE stmtseq {}
   | error NEWLINE stmtseq {}; /* After an error, start afresh */
```

- **Head:** sequence of statements
- **First body line:** Skip empty lines
- **Second body line:** separate current statement from rest
- **Third body line:** After parse error, start again with new line
Bison code for desk calculator: Grammar (2)

stmt: assign  {printf("> RegVal: %f\n", $<val>1);}
      |expr    {printf("> %f\n", $<val>1);};

assign: REGISTER ASSIGN expr {regfile[$<regno>1] = $<val>3;
                              $<val>$ = $<val>3;} ;

expr: expr PLUS term {$_<val>$ = $<val>1 + $<val>3;} 
     | term  {$_<val>$ = $<val>1;};

term: term MULT factor {$_<val>$ = $<val>1 * $<val>3;} 
     | factor {$_<val>$ = $<val>1;};

factor: REGISTER {$_<val>$ = regfile[$<regno>1];}
      | FLOAT  {$_<val>$ = $<val>1;}
      | OPENPAR expr CLOSEPAR {$_<val>$ = $<val>2;};
int yyerror(char* err)
{
    printf("Error: %s\n", err);
    return 0;
}

int main (int argc, char* argv[])
{
    int i;

    for(i=0; i<MAXREGS; i++)
    {
        regfile[i] = 0.0;
    }
    return yyparse();
}
Reminder: Workflow and dependencies

Bison Input File
<file>.y

Definitions file
<file>.tab.h

Parser Source
<file>.tab.c

Flex Input file
<file>.l

Lexer Source
<file>.c

Lexer object
<file>.o

Parser object
<file>.tab.o

Final executable
parser

Some input
to process

Some output
produced
Building the calculator

1. Generate parser C code and include file for lexer
   - `bison -d scicalcparse.y`
   - Generates `scicalcparse.tab.c` and `scicalcparse.tab.h`

2. Generate lexer C code
   - `flex -t scicalclex.l > scicalclex.c`

3. Compile lexer
   - `gcc -c -o scicalclex.o scicalclex.c`

4. Compile parser and support code
   - `gcc -c -o scicalcparse.tab.o scicalcparse.tab.c`

5. Link everything
   - `gcc scicalclex.o scicalcparse.tab.o -o scicalc`

6. Fun!
   - `./scicalc`
Exercise

- **Exercise 1 (Refresher):**
  - Download `scicalcparse.y` and `scicalclex.l`
  - Build the calculator
  - Run and test the calculator

- **Exercise 2 (Warm-up):**
  - Add support for division and subtraction `/`, `-`
  - Add support for unary minus (the negation operator `-`)

- **Exercise 3 (Bonus):**
  - Change the desk calculator so that it converts its input into a C program that will perform the same actions that the calculator performed interactively!
Review: Goals for Today

- Practical issues
- Programming language survey
- Execution of languages
- Low-level code vs. high-level code
- Structure of a Compiler
- Refresher
  - Grammars
  - Flex/Bison
- Programming exercises
  - Scientific calculator revisited
Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?
Goals for Today

- Refresher
- Reminder: Grammars and Chomsky-Hierarchy
  - Grammars
  - Regular languages and expressions
  - Context-free grammars and languages
- Syntactic structure of programming languages
- *nanoLang*
- Programming exercise: Tokenizing *nanoLang*
Some properties of programming languages and implementations

- Object oriented vs. Procedural
- Imperative vs. Functional
- Statically typed vs. dynamically typed (vs. „no types“)
- Compiled vs. interpreted

High-level level languages

- Expressive/Complex functionality
- Features correspond to application concepts
- Good abstraction

Low-level languages

- Simple operations
- Features dictated by hardware architecture
- (Close to) what processors can execute
- Limited abstraction
Refresher

- Structure of compiler
  - Tokenizer
  - Parser
  - Semantic analysis
  - Optimizer
  - Code generator
  - ...

- Some applications of compiler technology
  - Implementation of programming languages
  - Parsing of data formats/serialization
    - E.g. Word documents - may include optimization!
    - HTML/XML for web pages/SOA
    - XSLT document transformers
    - \LaTeX
    - ATCCL
    - ...

- Flex & Bison
Refresher: Grammars
Formal grammars describe formal languages!

- Derivative approach
  - A grammar has a set of rules
  - Rules replace words with words
  - A word that can be derived from a special start symbol is in the language of the grammar

In the concrete case of programming languages, “Words of the language” are syntactically correct programs!
Grammars: Examples

\[ S \rightarrow aA, \quad A \rightarrow bB, \quad B \rightarrow \varepsilon \]
generates \(ab\) (starting from \(S\)): \(S \rightarrow aA \rightarrow abB \rightarrow ab\)

\[ S \rightarrow \varepsilon, \quad S \rightarrow aSb \]
generates \(a^n b^n\)
Noam Chomsky defined a grammar as a quadruple

\[ G = \langle V_N, V_T, P, S \rangle \] (1)

with

1. the set of **non-terminal** symbols \( V_N \),
2. the set of **terminal** symbols \( V_T \),
3. the set of **production rules** \( P \) of the form

\[ \alpha \rightarrow \beta \] (2)

with \( \alpha \in V^* V^*_N V^*, \beta \in V^*, V = V_N \cup V_T \)

4. the distinguished **start symbol** \( S \in V_N \).
For the sake of simplicity, we will be using the short form

\[ \alpha \rightarrow \beta_1 | \cdots | \beta_n \] replacing \[ \alpha \rightarrow \beta_1 \]

\[ \vdots \]

\[ \alpha \rightarrow \beta_n \]
Example: C identifiers

We want to define a grammar

\[ G = \langle V_N, V_T, P, S \rangle \quad (4) \]

to describe identifiers of the C programming language:
- alpha-numeric words
- which must not start with a digit
- and may contain an underscore ( `_ `)

\[ V_N = \{ I, R, L, D \} \text{ (identifier, rest, letter, digit),} \]
\[ V_T = \{ a, \cdots, z, A, \cdots, Z, 0, \cdots, 9, _ \}, \]
\[ P = \{ \]
\[ I \rightarrow LR|_R|L|_ \]
\[ R \rightarrow LR|DR|_R|L|D|_ \]
\[ L \rightarrow a|\cdots|z|A|\cdots|Z \]
\[ D \rightarrow 0|\cdots|9 \} \]

\[ S = I. \]
**Formal grammars: derivation**

### Derivation: description of operation of grammars

Given the grammar

\[ G = \langle V_N, V_T, P, S \rangle, \]  

we define the relation

\[ x \Rightarrow_G y \] \quad (6)

iff \( \exists u, v, p, q \in V^* : (x = upv) \land (p \rightarrow q \in P) \land (y = uqv) \) \quad (7)

pronounced as “\( G \) derives \( y \) from \( x \) in one step”.

We also define the relation

\[ x \Rightarrow^*_G y \] \quad (8)

iff \( \exists w_0, \ldots, w_n \) with \( w_0 = x, w_n = y, w_{i-1} \Rightarrow_G w_i \) for \( i \in \{1, \ldots, n\} \)

pronounced as “\( G \) derives \( y \) from \( x \) (in zero or more steps)”.
Formal grammars: derivation example I

\[ G = \langle V_N, V_T, P, S \rangle \quad (9) \]

with

1. \( V_N = \{ S \}, \)
2. \( V_T = \{ 0 \}, \)
3. \( P = \{ S \rightarrow 0S, \quad S \rightarrow 0 \}, \)
4. \( S = S. \)

Derivations of \( G \) have the general form

\[ S \Rightarrow 0S \Rightarrow 00S \Rightarrow \cdots \Rightarrow 0^{n-1}S \Rightarrow 0^n \quad (10) \]

Apparently, the language produced by \( G \) (or the language of \( G \)) is

\[ L(G) = \{ 0^n | n \in \mathbb{N}; \ n > 0 \}. \quad (11) \]
Formal grammars: derivation example II

\[ G = \langle V_N, V_T, P, S \rangle \]  \hspace{1cm} (12)

with

1. \( V_N = \{ S \} \),
2. \( V_T = \{ 0, 1 \} \),
3. \( P = \{ S \rightarrow 0S1, \quad S \rightarrow 01 \} \),
4. \( S = S \).

Derivations of \( G \) have the general form

\[
S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow \cdots \Rightarrow 0^{n-1}S1^{n-1} \Rightarrow 0^n1^n. \]  \hspace{1cm} (13)

The language of \( G \) is

\[
L(G) = \{ 0^n1^n | n \in \mathbb{N}; \; n > 0 \}. \]  \hspace{1cm} (14)

Reminder: \( L(G) \) is not regular!
The Chomsky hierarchy (0)

Given the grammar

\[ G = \langle V_N, V_T, P, S \rangle, \]  

(15)

we define the following grammar/language classes

- G is of **Type 0 or unrestricted**

All grammars are Type 0!
The Chomsky hierarchy (1)

\[ G = \langle V_N, V_T, P, S \rangle, \quad (16) \]

- **G is Type 1 or context-sensitive** if all productions are of the form

  \[ \alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2 \text{ with } A \in V_N; \alpha_1, \alpha_2 \in V^*, \beta \in VV^* \quad (17) \]

Exception:

\[ S \rightarrow \varepsilon \in P \text{ is allowed if } \]

\[ \alpha_1, \alpha_2 \in (V\backslash\{S\})^* \text{ and } \beta \in (V\backslash\{S\})(V\backslash\{S\})^* \quad (18) \]

- If \( S \rightarrow \varepsilon \in P \), then \( S \) is not allowed in any right hand side
- Consequence: Rules (almost) never derive shorter words
The Chomsky hierarchy (2)

\[ G = \langle V_N, V_T, P, S \rangle \]  \hspace{1cm} (19)

- \( G \) is of **Type 2 or context-free** if all productions are of the form

\[ A \rightarrow \beta \text{ with } A \in V_N; \beta \in VV^* \]  \hspace{1cm} (20)

Exception:

\[ S \rightarrow \varepsilon \in P \text{ is allowed, if } \beta \in (V\setminus\{S\})(V\setminus\{S\})^* \]  \hspace{1cm} (21)

- Only single non-terminals are replaced
- If \( S \rightarrow \varepsilon \in P \), then \( S \) is not allowed in any right hand side
The Chomsky hierarchy (3)

\[ G = \langle V_N, V_T, P, S \rangle \quad (22) \]

- \( G \) is of **Type 3 or right-linear (regular)** if all productions are of the form

  \[ A \rightarrow aB \text{ or } \quad (23) \]

  \[ A \rightarrow a \text{ with } A, B \in V_N; a \in V_T \]

  Exception:

  \[ S \rightarrow \varepsilon \in P \text{ is allowed, if } B \in V_N \setminus \{S\} \quad (24) \]
The Chomsky hierarchy: exercises

\[ G = \langle V_N, V_T, P, S \rangle \]  \hspace{1cm} (25)

with

1. \[ V_N = \{ S, A, B \} \],
2. \[ V_T = \{ 0 \} \],
3. \[ P : \]
   \[
   \begin{align*}
   S & \rightarrow \varepsilon & 1 \\
   S & \rightarrow ABA & 2 \\
   AB & \rightarrow 00 & 3 \\
   0A & \rightarrow 000A & 4 \\
   A & \rightarrow 0 & 5
   \end{align*}
   \]
4. \[ S = S. \]

a) What is \( G \)'s highest type?
b) Show how \( G \) derives the word 000000.
c) Formally describe the language \( L(G) \).
d) Define a regular grammar \( G' \) equivalent to \( G \).
An octal constant is a finite sequence of digits starting with 0 followed by at least one digit ranging from 0 to 7. Define a regular grammar encoding exactly the set of possible octal constants.
Context-free grammars

- Reminder: $G = \langle V_N, V_T, P, S \rangle$ is context-free, if all $l \rightarrow r \in P$ are of the form $A \rightarrow \beta$ with
  - $A \in V_N$ and $\beta \in VV^*$
  - (special case: $S \rightarrow \epsilon \in P$, then $S$ is not allowed in any $\beta$)

- Context-free languages/grammars are highly relevant
  - Core of most programming languages
  - Algebraic expressions
  - XML
  - Many aspects of human language
Grammars in Practice

- Most programming languages are described by context-free grammars (with extra “semantic” constraints)
- Grammars generate languages
- PDAs and e.g. CYK-Parsing recognize words
- For compiler we need to . . .
  - identify correct programs
  - and understand their structure!
Lexing and Parsing

▶ Lexer: Breaks programs into tokens
  ▶ Smallest parts with semantic meaning
  ▶ Can be recognized by regular languages/patterns
  ▶ Example: 1, 2, 5 are all Integers
  ▶ Example: i, handle, stream are all Identifiers
  ▶ Example: >, >=, * are all individual operators

▶ Parser: Recognizes program structure
  ▶ Language described by a grammar that has token types as terminals, not individual characters
  ▶ Parser builds parse tree
Introduction: nanoLang
Our first language: *nanoLang*

- Simple but Turing-complete language
- Block-structured
  - Functions with parameters
  - Blocks of statements with local variables
- Syntax C-like” but simplified
  - Basic flow control (if, while, return)
- Simple static type system
  - Integers (64 bit signed)
  - Strings (immutable)
nanoLang  “Hello World”

# The first ever nanoLang program

Integer main()
{
    print "Hello World\n";
    return 0;
}
More Substantial *nanoLang* Example

```plaintext
Integer hello(Integer repeat, String message) {
    Integer i;
    i = 0;
    while (i < repeat) {
        print message;
        i = i + 1;
    }
    return 0;
}

Integer main() {
    hello(10, "Hello \n");
    return 0;
}
```
**nanoLang Lexical Structure**

- **Reserved words:**
  - if, while, return, print, Integer, String

- **Comments:** # to the end of the line

- **Variable length tokens:**
  - Identifier (letter, followed by letters and digits)
  - Strings (enclosed in double quotes ("This is a string")
  - Integer numbers (non-empty sequences of digits)

- **Other tokens:**
  - Brackets: (,), {,}
  - Operators: +, -, *, /
  - Comparison operators: >, >=, <, <=, !=
  - Equal sign = (used for comparison and assignments!)
  - Separators: ,, ;
**nanoLang Program Structure**

- A *nanoLang* program consists of a number of definitions
  - Definitions can define global variables or functions
  - All symbols defined in the global scope are visible everywhere in the global scope
- Functions accept arguments and return values
  - Functions consist of a header and a statement block
  - Local variables can be defined in statement blocks
- **Statements:**
  - `if`: Bedingte Ausführung
  - `while`: Schleifen
  - `return`: Return value from function
  - `print`: Print value to Screen
  - Assignment: Set variables to values
  - Function calls (return value ignored)
- **Expressions:**
  - Integers: Variables, numbers, +, -, *, /
  - Booleans: Compare two values of equal type
Exercise: Fibonacci in *nanoLang*

- Write a recursive and an iterative implementation of Fibonacci numbers in *nanoLang*
%start prog

%token OPENPAR CLOSEPAR
%left MULT DIV
%left PLUS MINUS
%token EQ NEQ LT GT LEQ GEQ
%token OPENCURLY CLOSECURLY
%token SEMICOLON COMA

%token <ident> IDENT
%token <string> STRINGLIT
%token <intval> INTLIT
%token INTEGER STRING
%token IF WHILE RETURN PRINT

%token ERROR
**nanoLang** Grammar (Bison format) (1)

```plaintext
prog: /* Nothing */
    | prog def
; ;

def: vardef
    | fundef
; ;

vardef: type IDENT SEMICOLON
; ;

fundef: type IDENT OPENPAR params CLOSEPAR body
; ;

type: STRING
    | INTEGER
; ;
```
nanoLang Grammar (Bison format) (2)

```bison
params: /* empty */
    | paramlist
;

paramlist: type IDENT
    | type IDENT COMA paramlist
;

body: OPENCURLY vardefs stmts CLOSECURLY
;

vardefs: /* empty */
    | vardefs vardef
;

stmts: /* empty */
    | stmts stmt
;```

101,
nanoLang Grammar (Bison format) (3)

stmt: while_stmt
    | if_stmt
    | ret_stmt
    | print_stmt
    | assign
    | funcall_stmt

; 

while_stmt: WHILE OPENPAR boolexpr CLOSEPAR body
;

if_stmt: IF OPENPAR boolexpr CLOSEPAR body
;

ret_stmt: RETURN expr SEMICOLON
;
nanoLang Grammar (Bison format) (4)

print_stmt: PRINT expr SEMICOLON ;

assign: IDENT EQ expr SEMICOLON ;

funcall_stmt: funcall SEMICOLON ;

boolexpr: expr EQ expr |
  | expr NEQ expr |
  | expr LT expr |
  | expr GT expr |
  | expr LEQ expr |
  | expr GEQ expr ;
expr: funcall
  | INTLIT
  | IDENT
  | STRINGLIT
  | OPENPAR expr CLOSEPAR
  | expr PLUS expr
  | expr MINUS expr
  | expr MULT expr
  | expr DIV expr
| MINUS expr
;

nanoLang Grammar (Bison format) (5)
funcall: IDENT OPENPAR args CLOSEPAR
;

args: /* empty */
    | arglist
;

arglist: expr
    | expr COMA arglist
;
Exercise

- Write a *flex*-based scanner for *nanoLang*
  - At minimum, it should output the program token by token
  - Bonus: Find a way to keep track of line numbers for tokens
  - Superbonus: Also keep track of columns
- Reminder: Compiling *flex* programs:
  flex -t myflex.l > myflex.c
gcc -o myflex myflex.c

Example output for Hello World

Integer = 277
main = 274
( = 258
) = 259
{ = 270
print = 282
"Hello World\n" = 275
; = 272
return = 281
0 = 276
; = 272
} = 271
Review: Goals for Today

- Refresher
- Reminder: Grammars and Chomsky-Hierarchy
  - Grammars
  - Regular languages and expressions
  - Context-free grammars and languages
- Syntactic structure of programming languages
- \textit{nanoLang}
- Programming exercise: Tokenizing \textit{nanoLang}
Feedback round

▷ What was the best part of today's lecture?
▷ What part of today's lecture has the most potential for improvement?
  ▷ Optional: how would you improve it?
Goals for Today

- Refresher
- Syntax analysis revisited
  - The truth about Context-Free Grammars
  - Derivations and Parse Trees
  - Abstract Syntax Trees
- Programming exercise: Parsing *nanoLang*
Refresher

- Refresher
- Reminder: Grammars and Chomsky-Hierarchy
  - Grammars
  - Regular languages and expressions
  - Context-free grammars and languages
- Syntactic structure of programming languages
- \textit{nanoLang}
- Programming exercise: Tokenizing \textit{nanoLang}
Reminder: $G$ is of Type 2 or context-free if all productions are of the form

$$A \rightarrow \beta \text{ with } A \in V_N; \beta \in VV^*$$  \hspace{1cm} (26)

Exception:

$$S \rightarrow \varepsilon \in P \text{ is allowed, if } \beta \in (V\setminus\{S\})(V\setminus\{S\})^*$$  \hspace{1cm} (27)

- Only single non-terminals are replaced
- If $S \rightarrow \varepsilon \in P$, then $S$ is not allowed in any right hand side
Question: Is *nanoLang* context-free?
Question: Is the *nanoLang* grammar context-free?

Yes/No, but ...

Problem:

```
prog: /* Nothing */
    | prog def
;
```

- prog is the start symbol
  - prog → ε
  - prog → prog def
Chomsky’s original definition:

\( G \) is of **Type 2 or context-free**

if all productions are of the form

\[ A \rightarrow \beta \text{ with } A \in V_N; \beta \in V^* \]  

(28)

Fact: Every Chomsky-CF-Grammar can be converted into a FLA-CF-Grammar!
Exercise: Eliminating $\epsilon$ rules

Consider the following productions:

1. $S \rightarrow \epsilon$
2. $S \rightarrow A; S$
3. $A \rightarrow i = n$

Upper-case letters are non-terminals, $S$ is the start symbol

- Specify $V_N$ and $V_T$
- Create an equivalent FLA-CF-Grammar
- Can you give a general method to convert Chomsky-CF-grammars to FLA-CF-gammars?
We will consider the set of well-formed expressions over $x, +, \ast, (, )$ as an example, i.e. $L(G)$ for $G$ as follows:

- $V_N = \{E\}$
- $V_T = \{(, ), +, \ast, x\}$
- Start symbol is $E$
- Productions:
  1. $E \rightarrow x$
  2. $E \rightarrow (E)$
  3. $E \rightarrow E + E$
  4. $E \rightarrow E \ast E$
**Definition:** Assume a Grammar $G$. A derivation of a word $w_n$ in $L(G)$ is a sequence $S \implies w_1 \implies \ldots \implies w_n$ where $S$ is the start symbol, and each $w_i$ is generated from its predecessor by application of a production of the grammar.

- Example: Consider our running example. We bold the replaced symbol. The following is a derivation of $x + x + x \ast x$:

  \[
  \begin{align*}
  E & \implies E + E \\
  & \implies E + E + E \\
  & \implies E + E + E + E \\
  & \implies x + E + E + E \\
  & \implies x + x + E + E + E \\
  & \implies x + x + x + E \ast E \\
  & \implies x + x + x + x \ast E \\
  & \implies x + x + x \ast x
  \end{align*}
  \]
Rightmost/Leftmost Derivations

Definition:

- A derivation is called a **rightmost** derivation, if at any step it replaces the **rightmost** non-terminal in the current word.
- A derivation is called a **leftmost** derivation, if at any step it replaces the **leftmost** non-terminal in the current word.

Examples:

- The derivation on the previous slide is neither leftmost nor rightmost.
- \[ E \rightarrow E + E \rightarrow E + E + E \rightarrow E + E + E \ast E \rightarrow E + E + E \ast x \rightarrow E + E + x \ast x \rightarrow E + x + x \ast x \rightarrow x + x + x \ast x \] is a **rightmost derivation**.
**Parse trees**

**Definition:** A parse tree for a derivation in a grammar $G = \langle V_N, V_T, P, S \rangle$ is an ordered, labelled tree with the following properties:

- Each node is labelled with a symbol from $V_N \cup V_T$.
- The root of the tree is labelled with the start symbol $S$.
- Each inner node is labelled with a single non-terminal symbol from $V_N$.
- If an inner node with label $A$ has children labelled with symbols $\alpha_1, \ldots, \alpha_n$, then there is a production $A \rightarrow \alpha_1 \ldots \alpha_n$ in $P$.

- The parse tree represents a derivation of the word formed by the labels of the leaf nodes.
- It abstracts from the order in which productions are applied.
Consider the following derivation:

\[ E \rightarrow E + E \rightarrow E + E + E \rightarrow E + E + E \ast E \rightarrow E + E + E \ast x \rightarrow E + E + x \ast x \rightarrow E + x + x \ast x \rightarrow x + x + x \ast x \]

It can be represented by a sequence of parse trees:
Ambiguity

**Definition:** A grammar \( G = \langle V_N, V_T, P, S \rangle \) is ambiguous, if it has multiple different parse trees for a word \( w \) in \( L(G) \).

- Consider our running example with the following productions:
  1. \( E \to x \)
  2. \( E \to (E) \)
  3. \( E \to E + E \)
  4. \( E \to E \ast E \)

- The following 2 parse trees represent derivations of \( x + x + x \):
Exercise: Ambiguity is worse...

- Consider our example and the parse trees from the previous slide:
  1. $E \rightarrow x$
  2. $E \rightarrow (E)$
  3. $E \rightarrow E + E$
  4. $E \rightarrow E \ast E$

- Provide a rightmost derivation for the right tree.
- Provide a rightmost derivation for the left tree.
- Provide a leftmost derivation for the left tree.
- Provide a leftmost derivation for the right tree.
Consider our running example with the following productions:

1. \( E \rightarrow x \)
2. \( E \rightarrow (E) \)
3. \( E \rightarrow E + E \)
4. \( E \rightarrow E \ast E \)

Define a grammar \( G' \) with \( L(G) = L(G') \) that is not ambiguous, that respects that \( \ast \) has a higher precedence than \( + \), and that respects left-associativity for all operators.
Flex/Bison Interface

- Bison calls function `yylex` to get the next token
- `yylex` executes user rules (pattern/action)
  - User actions return token (integer value)
  - Additionally: `yylval` can be set and is available in Bison via the `$$/$1/ldots` mechanism
- `yylval` provides the semantic value of a token
  - For complex languages: Use a pointer to a struct
    - Content: Position, string values, numerical values, ...
  - Type of `yylval` if set in `Bison` file!
    ```plaintext
    %define api.value.type {YourType}
    ```
Grading Exercise 2

- Write a Bison parser for \textit{nanoLang}
  - Bonus: Translate \textit{nanoLang} into Abstract Syntax Trees (will be required next week!)
Review: Goals for Today

- Refresher
- Syntax analysis revisited
  - The truth about Context-Free Grammars
  - Derivations and Parse Trees
  - Abstract Syntax Trees
Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?
Goals for Today

- Refresher
- Revisiting derivations, parse trees, abstract syntax trees
- Walk-through: Parsing expressions in practice
- Programming exercise: ASTs for nanoLang
Refresher

- Refresher
- Syntax analysis revisited
  - The truth about Context-Free Grammars
  - Derivations and Parse Trees
  - Abstract Syntax Trees
- Programming exercise: Parsing *nanoLang* (i.e. writing a program that accepts syntactically correct *nanoLang* programs and rejects syntactically incorrect ones (due next week)
**Parse trees**

**Definition:** A parse tree for a derivation in a grammar $G = \langle V_N, V_T, P, S \rangle$ is an ordered, labelled tree with the following properties:

- Each node is labelled with a symbol from $V_N \cup V_T$.
- The root of the tree is labelled with the start symbol $S$.
- Each inner node is labelled with a single non-terminal symbol from $V_N$.
- If an inner node with label $A$ has children labelled with symbols $\alpha_1, \ldots, \alpha_n$, then there is a production $A \rightarrow \alpha_1 \ldots \alpha_n$ in $P$.

- The parse tree represents a derivation of the word formed by the labels of the leaf nodes.
- It abstracts from the order in which productions are applied.
Parse trees: Example

Consider the following derivation:

\[ E \rightarrow E + E \rightarrow E + E + E \rightarrow E + E + E \ast E \rightarrow E + E + E \ast E \rightarrow E + E + E \ast x \rightarrow E + E + x \ast x \rightarrow E + x + x \ast x \rightarrow x + x + x \ast x \]

It can be represented by a sequence of parse trees:
**Definition:** A grammar \( G = \langle V_N, V_T, P, S \rangle \) is ambiguous, if it has multiple different parse trees for a word \( w \) in \( L(G) \).

- Consider our running example with the following productions:
  1. \( E \rightarrow x \)
  2. \( E \rightarrow (E) \)
  3. \( E \rightarrow E + E \)
  4. \( E \rightarrow E \ast E \)

- The following 2 parse trees represent derivations of \( x + x + x \):
Exercise: Ambiguity is worse...

Consider our example and the parse trees from the previous slide:
1. $E \rightarrow x$
2. $E \rightarrow (E)$
3. $E \rightarrow E + E$
4. $E \rightarrow E \ast E$

Provide a rightmost derivation for the right tree.
Provide a rightmost derivation for the left tree.
Provide a leftmost derivation for the left tree.
Provide a leftmost derivation for the right tree.
Abstract Syntax Trees

- **Abstract Syntax Trees** represent the structure of a derivation without the specific details.
- Think: “Parse trees without the syntactic sugar”
- Example:

  **Parse Tree:**

  ![Parse Tree Diagram]

  **Corresponding AST:**

  ![Corresponding AST Diagram]
From text to AST in practice: Parsing *nanoLang* expressions

- Example for syntax analysis and building abstract syntax trees
- Language: *nanoLang* expressions (without function calls)
- Structure of the project
- Building
- Code walk-through
Exercise: Building exprcc

- Go to
  http://wwwlehre.dhbw-stuttgart.de/~sschulz/cb2015.html
- Download NANOEXPR.tgz
- Unpack, build and test the code
- To test:
  - ./exprcc expr1.nano
  - ./exprcc --sexpr expr1.nano
  - ./exprcc --dot expr1.nano
exprcc Overview

- AST definitions: `ast.h`
- Parser specification: `nanoparse.y`
- Lexer specification: `nanolex.l`
- Parserdefs: `nanoparse.tab.h`
- Lexer implementation: `nanolex.c`
- AST implementation: `ast.c`
- Parser implementation: `nanoparse.tab.c`
- AST object file: `ast.o`
- Parser object file: `nanoparse.tab.c`
- Lexer object file: `nanolex.o`
- Executable: `exprcc`
Test expression: $-a + b \times (c + d)$

Corresponding AST?
Simplified *nanoLang* expression syntax

\[
\text{expr: INTLIT} \\
| \text{IDENT} \\
| \text{STRINGLIT} \\
| \text{OPENPAR expr CLOSEPAR} \\
| \text{expr PLUS expr} \\
| \text{expr MINUS expr} \\
| \text{expr MULT expr} \\
| \text{expr DIV expr} \\
| \text{MINUS expr}
\]

;
Alternative notation

\[ expr \rightarrow \text{INTLIT} \]
\[
| \text{IDENT} \\
| \text{STRINGLIT} \\
| ( \text{expr} ) \\
| \text{expr} + \text{expr} \\
| \text{expr} - \text{expr} \\
| \text{expr} * \text{expr} \\
| \text{expr} / \text{expr} \\
| - \text{expr}
\]

Question: Is the grammar unambiguous?

➤ How do we solve this?
Code: *nanoparse.y* token definitions

The trick with unary –
Implementing ASTs

- Code: ast.c, ast.h
Lexical Analysis

- Code: nanolex.l
Building ASTs

- Code: `nanoparse.y` syntax rules and semantic actions
Extend the *nanoLang* parser to generate abstract syntax trees *nanoLang* programs.

- You can use your own parser or extend the expression parser from this lecture.
- Due date: Our lecture on April 22nd.
Review: Goals for Today

- Refresher
- Revisiting derivations, parse trees, abstract syntax trees
- Walk-through: Parsing expressions in practice
- Programming exercise: ASTs for nanoLang
Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?
Goals for Today

- Refresher
- Semantic properties
  - Names, variables, identifiers
  - Visibility and scopes
  - Simple types and type systems
- Symbol tables
- Memory organisation and storage locations
Refresher

- Formal definition of parse trees
- Ambiguity and derivation types
- Abstract syntax trees
- Syntax analysis in practice
  - *nanoLang* expression parser
  - Abstract syntax tree datatype and algorithms
  - Parsing *nanoLang* expressions with Bison
- Programming exercise: Parsing *nanoLang* into abstract syntax trees
High-Level Architecture of a Compiler

Source handler

Lexical analysis (tokeniser)

Syntactic analysis (parser)

Semantic analysis

Code generation (several optimisation passes)

Sequence of characters:
i,n,t, ⊥, a,, b, ;, a, =, b, +, 1, ;

Sequence of tokens:
(id, “int”), (id, “a”), (id, “b”), (semicolon), (id, “a”), (eq), (id, “b”), (plus), (int, “1”), (semicolon)

e.g. Abstract syntax tree

e.g. AST+symbol table+attributes

ld a,b
ld c, 1
add c
...

ld a,b
ld c, 1
add c
...

Variable | Type |
---|---|
a | int |
b | int |

Bison

C Code

Flex
Semantic Constraints
Group Exercise: Spot the Bugs (1)

```java
Integer fun1(Integer i, Integer i)
{
    Integer i;

    if(i > 0)
    {
        print j;
    }
}

Integer main()
{
    fun1(1, 2);
    fun2(1, 2);
    return 0;
}
```
Group Exercise: Spot the Bugs (2)

```java
Integer fun1(Integer i, Integer j)
{
    Integer i;

    if (i > "0")
    {
        print j+"12";
    }

    return 1;
}

Integer main()
{
    fun1(1, "Hello");
    fun2(1, 2, 3);
    return 0;
}
```
Group Exercise: Spot the Bugs (3)

```plaintext
Integer fun1(Integer i, Integer j)
{
    while (j > i)
    {
        Integer j;

        print j;
        j = j + 1;
    }
    return 1;
}
```
Semantic constraints of *nanoLang* (V 1.0)

- Every name has to be defined before it can be used
- Every name can only be defined once in a given scope
- Functions must be called with arguments of the right type in the right order
- Operands of comparison operators must be of the same type
- Operands of the arithmetic operators must be of type Integer
- Every program must have a `main()`
- (Every function must have a return of proper type)
Managing Symbols

- Properties of identifiers are stored in a symbol table
  - Name
  - Type

- Properties of identifiers depend on part of the program under consideration!
  - Names are only visible in the scope they are declared in
  - Names can be redefined in new scopes

Symbol tables need to change when traversing the program/AST for checking properties and generating code!
Definitions:

- **Variable**: A location in memory (or “in the store”) that can store a value (of a given type).
  - Variables can be statically or dynamically allocated.
    - Typically: global variables are statically allocated (and in the data segment of the process).
    - Local variables are dynamically managed and on the stack.
    - Large data structures and objects are stored in the heap.

- **Name**: An identifier that identifies a variable (in a given scope).
  - The same name can refer to different variables (recursive function calls).
  - Different names can refer to the same variables (depends on programming languages - aliasing).
Scopes and Environments

- The environment establishes a mapping from names to variables.
- **Static scope**: Environment depends on block structure of the language.
  - In any situation, the name refers to the variable defined in the nearest surrounding block in the program text.
  - Examples: C (mostly), Pascal, Java, modern LISPs (mostly).
- **Dynamic scope**: Environment depends on calling sequence in program.
  - Name refers to the same variable it referred to in the calling function.
  - Traditional LISP systems (Emacs LISP).
#include <stdio.h>
i nt a=10;
i nt b=10;
#define adder(x) (x)+a

void machwas(int a, int c)
{
    printf("adder(a)=%d\n", adder(a));
    printf("adder(b)=%d\n", adder(b));
    printf("adder(c)=%d\n", adder(c));
    
    int c = 5;
    printf("adder(c)=%d\n", adder(c));
}

int main(void)
{
    machwas(1, 2);
    machwas(2, 3);

    return 0;
}
Example: Scopes in *nanoLang*

```plaintext
Integer i;

Integer fun1(Integer loop)
{
    Integer i;

    i=0;
    while (i<loop)
    {
        i=i+1;
        print "Hallo";
    }
}

Integer main()
{
    i = 5;
    fun1(i);
}
```
Scopes in *nanoLang*

- Global scope
  - Global variables
  - Functions
- Function scope
  - Function parameters
- Block scope
  - block-local variables
Example: Scopes in *nanoLang*

```
Integer i;

Integer fun1(Integer loop)
{
    Integer i;

    i=0;
    while (i<loop)
    {
        i=i+1;
        print "Hallo";
    }
}

Integer main()
{
    i = 5;
    fun1(i);
}
```
Walking the AST
Static type checking

- Types are associated with variables
- Types are checked at compile time or development time
- Advantages:
  - ?
- Disadvantages:
  - ?
- Typically used in:
  - ?
Dynamic type checking

- Types are associated with values
- Types are checked at run time
- Advantages:
  - ?
- Disadvantages:
  - ?
- Typically used in:
  - ?
No type checking

- Programmer is supposed to know what (s)he does
- Types are not checked at all
- Advantages:
  - ?
- Disadvantages:
  - ?
- Typically used in:
  - ?
Exercise: How many types occur in this example?

```c
Integer i;

Integer fun1(Integer loop)
{
    Integer i;

    i=0;
    while(i<loop)
    {
        i=i+1;
        print "Hallo";
    }
}

Integer main()
{
    i = 5;
    fun1(i);
}
```
Symbol tables

▶ Store name (identifier) and type

▶ Nested for each scope:
  ▶ Global scope
  ▶ Each new scope entered will result in a new symbol table for that scope, pointing to the preceding (larger scope)

▶ Local operations:
  ▶ Add name/type (error if already defined)

▶ Global operations:
  ▶ Find name (finds “nearest” matching entry, or error)
  ▶ Enter new scope (creates a new empty symbol table pointing to the predecessor (if any)
  ▶ Leave scope (remove current scope)
Walking the AST
Representing types

- Types table:
  - Numerical encoding for each type
  - Recipe for each type
    - *nanoLang* basic types are atomic
    - Atomic types can also be addressed by name
    - Function types are vectors of existing types

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Type</th>
<th>Recipe</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>String</td>
<td>atomic</td>
</tr>
<tr>
<td>1</td>
<td>Integer</td>
<td>atomic</td>
</tr>
<tr>
<td>2</td>
<td>Integer fun(Integer, String)</td>
<td>(1, 1, 0)</td>
</tr>
</tbody>
</table>

- E.g.

- Operations:
  - Find-or-insert type
    - Return encoding for a new type
    - If type does not exist yet, create it
Programming Exercise

- Develop data structures for representing *nanoLang* types
- Develop data structures for implementing nested symbol tables
Review: Goals for Today

- Refresher
- Semantic properties
  - Names, variables, identifiers
  - Visibility and scopes
  - Simple types and type systems
- Symbol tables
- Memory organisation and storage locations
Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?
Goals for Today

- Refresher
- Symbol Tables in practice
- Type inference and type checking
- Exercise: Build symbol tables
Refresher

- Semantic properties
  - Names, variables, identifiers
  - Visibility and scopes
  - Simple types and type systems
- Symbol tables
- Memory organisation and storage locations
The Big Picture: Type Checking

- We need to know the **type** of every expression and variable in the program
  - ...to detect semantic inconsistencies
  - ...to generate code

- Some types are simple in **nanoLang**
  - String constants are type `String`
  - Integer constants are type `Integer`
  - Results of arithmetic operations are `Integer`

- Harder: What to do with identifiers?
  - Type of the return value of a function?
  - Types of the arguments of a function?
  - Types of the values of a variable?

The answers depend on the definitions in the program!
Symbol Tables

- Symbol tables associate identifiers and types
- Symbol tables form a hierarchy
  - Symbols can be redefined in every new context
  - The “innermost” definition is valid
- Symbol tables are filled top-down
  - Outermost symbol-table contains global definitions
  - Each new context adds a new layer
  - Search for a name is from innermost to outermost symbol table
Building Symbol Tables

Program

```
Integer i;

Integer fun1(Integer loop) {
   Integer i;
   i=0;
   while(i<loop) {
      i=i+1;
      print "Hallo";
   }
}

Integer main() {
   i = 5;
   fun1(i);
}
```

Symbol table

```
<table>
<thead>
<tr>
<th>i</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun1</td>
<td>(Integer)→Integer</td>
</tr>
<tr>
<td>main</td>
<td>()→Integer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loop</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun1</td>
<td>(Integer)→Integer</td>
</tr>
<tr>
<td>main</td>
<td>()→Integer</td>
</tr>
</tbody>
</table>
```

178
Simplified Type Handling

- Handling complex types directly is cumbersome
- Better: Manage types separately
  - Types are stored in a separate table
  - Symbol table only needs to handle indices into type table
Symbol Tables and Type Tables

Program

```
Integer i;

Integer fun1(Integer loop)
{
   Integer i;
   i=0;
   while(i<loop)
   {
      i=i+1;
      print "Hallo";
   }
}

Integer main()
{
   i = 5;
   fun1(i);
}
```

Symbol table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>fun1</td>
<td>2</td>
</tr>
<tr>
<td>main</td>
<td>3</td>
</tr>
<tr>
<td>loop</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Type table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Integer</td>
</tr>
<tr>
<td>2</td>
<td>(Integer)→Integer</td>
</tr>
<tr>
<td>3</td>
<td>()→Integer</td>
</tr>
</tbody>
</table>
Type Inference in \textit{nanoLang}

- **Goal**: Determine the (result) type of every expression in the program
- **Process**: Process AST bottom-up
  - **Constants**: “natural” type
  - **Variables**: Look up in symbol table
  - **Function calls**: Look up in symbol table
    - If arguments are not of proper type, error
    - Otherwise: return type of the function
  - **Arithmetic expressions**:
    - If arguments are \texttt{Integer}, result type is \texttt{Integer}
    - Otherwise: error
Contrast: Aspects of Type Inference in C

- **Arithmetic expressions:**
  - Roughly: arithmetic types are ordered by size (char < int < long < long long < float < double)
  - Type of \( a + b \) is the greater of the types of \( a \) and \( b \)

- **Arrays**
  - If \( a \) is an array of int, than \( a[1] \) is of type int

- **Pointers**
  - If \( a \) is of type char*, then \( *a \) is of type char

- **Many more cases:**
  - Structures
  - Enumerations
  - Function pointers
Symbol Tables and the AST

Integer i;
Integer fun1(Integer loop)
{
    Integer i;
    i=0;
    while(i<loop)
    {
        i=i+1;
        print "Hallo";
    }
}

Integer main()
{
    i = 5;
    fun1(i);
}
Implementation Examples

- `main()` in `nanoparse.y`
- `STBuildAllTables()` in `semantic.c`
- `symbols.h` and `symbols.c`
- `types.h` and `types.c`
Grading Exercise 4

Extend your compiler project by computing the relevant symbol tables for all nodes of your AST

- Develop a type table date type for managing different types
- Define a symbol table data type for managing symbols and their types
  - Use a hierarchical structure
  - Suggested operations:
    - EnterScope()
    - LeaveScope()
    - InsertSymbol() (with type)
    - FindSymbol() (return entry including type)
    - ...

- Traverse the AST in a top-down fashion, computing the valid symbol table at each node
- Annotate each AST node with the symbol table valid at that node

At the end, print all symbols and types of the global, top-level symbol table!
Example Output

> ncc NANOEXAMPLES/scopes.nano

Global symbols:
---------------
i : Integer
fun1 : (Integer) -> Integer
main : () -> Integer

Types:
------
0: NoType
1: String
2: Integer
3: (Integer) -> Integer
4: () -> Integer
Review: Goals for Today

- Symbol Tables in practice
- Type inference and type checking
- Exercise: Build symbol tables
Feedback round

► What was the best part of today's lecture?
► What part of today's lecture has the most potential for improvement?
  ► Optional: how would you improve it?
Goals for Today

- Refresher
- Excursion: `assert()` in C
- Code generation considerations
  - Parameter passing
  - Assignments
  - Calling conventions
  - Runtime support
  - Libraries
- `nanoLang` runtime
  - Parameter passing
  - `nanoLang` string semantics
  - `nanoLang` library functions and OS interaction
- Exercise: Type checking
Refresher

- Symbol Tables in practice (top-down traversal)
- Type inference and type checking (bottom-up traversal)
- Example code walk-through
- Exercise: Build symbol tables
Excursion: assert()

- `assert()` is a facility to help debug programs
  - Part of the C Standard since C89
  - To use, `#include <assert.h>`
- `assert(expr)` evaluates `expr`
  - If `expr` is false, then an error message is written and the program is aborted
  - Otherwise, nothing is done
- Hints:
  - Particularly useful to check function parameter values
  - To disable at compile time, define the macro `NDEBUG` (e.g. with the compiler option `-DNDEBUG`)
  - Useful idiom: `assert(expr && "What’s wrong");`
  - More information: `man assert`
Before we can compile a language, we must understand its semantics.

Important questions:
- How are parameters passed into functions?
- Related: How do assignments work?

Before we can compile a language, we must understand the target language and environment.
- How are parameters and local variables handled?
- How does the program interact with the OS and the environment?
Parameter Passing

- **Call by value**
  - Formal parameters become new local variables
  - Actual parameters are evaluated and used to initialize those variables
  - Changes to variables are irrelevant after function terminates

- **Call by reference**
  - Only *references* to existing variables are passed
  - In effect, formal parameters are bound to *existing* variables
  - Actual parameters that are not variables themselves are evaluated and placed in anonymous new variables
  - Changes to parameters in functions change the original variable

- **Call by name**
  - Only historically interesting
  - Semantics mostly similar to *call-by-value*
Parameter Passing - Advantages and Disadvantages?

- Call by value?
- Call by reference?
- For your consideration:

```c
int fun(int a, int b)
{
    a++; 
    b++; 
    return a+b; 
}

int main(void)
{
    int i =0; 

    fun(i, i); 
    printf("i=%d\n", i); 
}
```
Parameter Passing in C/C++/Pascal/Scheme?

- C?
- C++?
- Pascal?
- LISP/Scheme?
- Others?
Assignments

What happens if $a = b;$ is encountered?

- If both are integer variables?
- If both are string variables?
- If both have an object type?
- If both are arrays?
Calling conventions

- How are parameters values passed at a low level?
  - Registers?
  - Stack?
  - Other?
- Who is responsible for preserving registers?
  - Caller?
  - Callee?
- In which order are parameters passed?
- How is the old context (stack frame and PC) preserved and restored?

For our \textit{nanoLang} compiler, we rely on C to handle these things!
Runtime system and OS Integration

- Runtime system provides the glue between OS and program
  - Translates OS semantics/conventions to compiled language and back
- Runtime system provides execution support for program semantics
  - Higher-level functions/data types
  - Memory management
  - Library functions
Parameter passing and assignments in *nanoLang*

- Suggestion: All parameter passed “as if” by value
- Integer: Pass by value
- *Immutable strings*
  - Can be passed by reference
  - Need to be memory-managed (reference counting, a job for the runtime system)
  - Alternative is not simpler!
nanoLang OS integration

- Command line arguments
  - Suggestion: `main()` takes arbitrary number of string arguments
  - These are filled from the command line
  - Spare arguments are represented by the empty string
- Exit and return value
  - Library function `Exit(val)` terminates program and returns integer value
  - `return` from `main()` has the same effect
Suggested function to make things interesting:

- **StrIsInt**(str): Returns 1 if str encodes a valid integer, 0 otherwise
- **StrToInt()**: Converts a string to an integer. If str is not an integer encoding, result is undefined
- **IntToStr**(int): Returns a string encoding of the given integer
- **StrLength**(str): Returns the lengths of str

More suggestions?

- **String StrFront**(str, int) - return first int characters as new string
- **String StrRest**(str, int) - return all but first int characters
- **String StrCat**(str, str) - concatenate strings, return as new string
- **Integer StrToASCII**(str) - only for strings of length 1, return ASCII value
- **String ASCIIToStr**(int) - reverse of the above
Temptation: Use C `char*`

- Fails as soon as strings can be dynamically created
- Suggestion: Structure with reference counting
  - String value - the actual string (`malloc()`ed `char*`)
  - Length (maybe)
  - Reference count - how many places have a reference to the string?
    - Increase if string is assigned to a variable or passed to a function
    - Decrease, if a variable is reassigned or goes out of scope
    - Free string, if this reaches 0
Extend your compiler project by computing the types of all expressions in your system and check type constraints

- Check that variables are only assigned values of the right type
- Check that functions are only called with correctly typed parameters
- Check that operators have compatible types
- Check that comparisons only happen between expressions of the same type
- Bonus: Check that functions (always) return the correct type
Review: Goals for Today

- Excursion: `assert()` in C
- Code generation considerations
  - Parameter passing
  - Assignments
  - Calling conventions
  - Runtime support
  - Libraries
- `nanoLang` runtime
  - Parameter passing
  - `nanoLang` string semantics
  - `nanoLang` library functions and OS interaction
Feedback round

▶ What was the best part of today's lecture?
▶ What part of today's lecture has the most potential for improvement?
  ▶ Optional: how would you improve it?
Goals for Today

- Refresher
- Coding Hints
- Code generation *nanoLang* to C
- (Simple) Optimizations
- Exercise: Code generation (I)
Refresher

- `assert()`
- General considerations for code generation
  - Semantics of parameters/assignments
  - Function calls
  - Runtime support and libraries
- Special considerations for `nanoLang`
  - Strings
  - Command line processing
  - Built-in library functions
Coding hints

- The *nanoLang* compiler is a non-trivial piece of software
  - Several modules
  - Several different data types (AST, Types, Symbols)
- It helps to follow good coding practices
  - The big stuff: Good code structure
    - One function per function
    - Not more than one screen page per function
  - The small stuff
    - Clean formatting (including vertical space)
    - Use expressive names for functions and variables
    - Reasonable comments (don’t over-comment, though!)
    - Use `assert()`
    - Compile with warnings enables (Makefile: CFLAGS = -Wall)
The Final Phase

Source handler

Sequence of characters:
\texttt{i,n,t, =, a,, b, ;, a, =, b, +, 1, ;}

Lexical analysis (tokeniser)

Sequence of tokens:
\texttt{(id, "int"), (id, "a"), (id, "b"), (semicolon), (id, "a"), (eq), (id, "b"), (plus), (int, "1"), (semicolon)}

Syntactic analysis (parser)

\textit{e.g. Abstract syntax tree}

Semantic analysis

\textit{e.g. AST+symbol table+attributes}

Code generation (several optimisation passes)

\textit{e.g. assembler code}

\begin{verbatim}
ld a,b
ld c, 1
add c
...
\end{verbatim}
Suggestion: Code generation uses separate phases

- Initial boilerplate
- Global variable definitions
- Function declarations
- Constant library code
- Translation of function definitions
- C `main()` function
To avoid name conflicts, nanoLang identifier should use a standard naming scheme

Suggestion:
- Atomic type names are prepended with $N_-$
- Function and variable names are prepended with $n_-$
Initial boilerplate

▶ Emit constant code needed for each translated *nanoLang* program
  ▶ Comment header
  ▶ Standard system includes
  ▶ Type definitions
  ▶ Possibly macro definitions

▶ Implementation via printing constant string
  ▶ Easiest way
  ▶ Alternative: Read from file
Visibility difference between *nanoLang* and C

- Globally defined *nanoLang* identifiers are visible throughout the program
- C definitions are visible from the point of definition only
- Hence we need to declare variables (and functions) upfront

Implementation suggestion:

- Iterate over all symbols in the global symbol table
- For each variable symbol, emit a declaration
Function declarations

- The same visibility difference between *nanoLang* and C affects functions
  - We need to declare all functions upfront!
- Implementation suggestion:
  - Iterate over all symbols in the global symbol table
  - For each function symbol, emit a declaration

Suggestion: For simplicity and consistency, we should insert the *nanoLang* standard library functions (Exit(), StrIsInt(), StrToInt, ...) into the symbol table (and do so before semantic analysis to stop the user from inadvertently redefining them!)
The *nanoLang* runtime will need various pieces of code

- Data types and helper functions to handle e.g. Strings
- Implementations of the build-in functions

Implementation options

- Just insert plain C code here (Alternative 0, but this may be lengthy)
- Alternative 1: Read this C code from a file
- Alternative 2: Just `#include` the full C code
- Alternative 3: `#include` only header with declarations, then require linking with a run time library later
Translation of function definitions

- This is the heart of the compiler!
- Go over the AST and emit a definition for each function
  - nanoLang functions become C functions
  - Local nanoLang variables become C variables of an appropriate type
  - nanoLang blocks become C blocks
  - nanoLang instructions are translated into equivalent C statement sequences
- Mostly straightforward
  - print requires case distinction
  - String comparisons require library calls

More complex: Proper string handling
C main() function

- Generate an appropriate main() function
- Tasks:
  - Read commandline and initialize parameters for nanoLang main()
  - Call nanoLang main
  - Exit program, returning value from nanoLang main() to OS
Ideas for optimization

- Constant subexpression evaluation
- Common subexpression elimination
  - To do this well, we need to identify *pure* functions!
- Shift unneeded computations out of loop
- Eliminate computations of unused values

```c
while (i < 10)
{
    a = 3*10*i;
    b = 3*10*fun(a);
    i=i+1;
}
return  a;
```
Extend your compiler project to generate a basic C program

- Compile *nanoLang* statements into equivalent C statements
- Compile *nanoLang* definitions into C declarations and definitions
- Generate a basic `main()`
- For now, you can treat `String` as an immutable `char*` - we’ll do the library next week
Review: Goals for Today

- Refresher
- Coding Hints
- Code generation *nanoLang* to C
- Optimizations
- Exercise: Code generation (I)
Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?
Goals for Today

- Refresher
- Practical aspects of *nanoLang* code generation
- An introduction to top-down recursive descent parsing
Refresher

- Coding Hints
- Code generation *nanoLang* to C
  - Name handling
  - Global definitions
  - Libraries
  - Functions
  - ...
- Optimizations
  - Constant subexpressions
  - Common subexpressions (purely functional functions!)
  - Lift invariant expression out of loops
  - Eliminate computation of unused results
  - ...

223
Practical code generation for nanoLang
/*
 * Automatically generated by the nanoLang compiler ncc.
 * The boilerplate and library code is released under the GNU General Public
 * Licence, version 2 or, at your choice, any later version. Other code is
 * governed by the license of the original source code.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef long long N_Integer;
typedef char    *N_String;

#define NANO_MAKESTR(s) s
#define NANO_STRASSIGN(l, r) (l) = (r)
#define NANO_STRVAL(s) s

/* Global user variables */
nanoLang C preamble (2)

/* Function declarations */

N_Integer n_EXIT(N_Integer);
N_Integer n_StrIsInt(N_String);
N_Integer n_StrToInt(N_String);
N_Integer n_StrLen(N_String);
N_String n_IntToStr(N_Integer);
N_String n_StrFront(N_String, N_Integer);
N_String n_StrRest(N_String, N_Integer);
N_String n_StrCat(N_String, N_String);
N_Integer n_StrToASCII(N_String);
N_String n.ASCIIIToStr(N_Integer);
N_String n_testfun(N_Integer, N_String);
N_Integer n_main(N_String, N_String);

/* nanoLang runtime library code */

/* String functions */

N_String n_StrCat(N_String arg1, N_String arg2)
{
    size_t len = strlen(arg1) + strlen(arg2) + 1;
    char *res = malloc(len);
    strcpy(res, arg1);
    strcat(res, arg2);
    return res;
}
[...]
\textbf{nanoLang} and its translation (2)

\begin{verbatim}
String testfun(Integer count, String message)
{
    Integer i;
    String res;

    i = 0;
    res = "";

    while (i < count)
    {
        print " Schleifendurchlauf_";
        print i;
        print " \n";
        res = StrCat(res, message);
        i = i + 1;
    }
    return res;
}
\end{verbatim}

\begin{verbatim}
N_String n_testfun(N_Integer n_count, N_String n_message)
{
    N_Integer n_i = 0;
    N_String n_res = 0;
    N_i = (0);
    NANO_STRASSIGN(n_res, (NANO_MAKESTR("")));
    n_res = (NANO_MAKESTR(""));
    while ((n_i) < (n_count)) {
        printf("%s", NANO_STRVAL((
            NANO_MAKESTR(" Schleifendurchlauf_"))));
        printf("%lld", (n_i));
        printf("%s", NANO_STRVAL((NANO_MAKESTR(" \n"))));
        NANO_STRASSIGN(n_res, (n_StrCat((n_res),
            (n_message))));
        n_res = (n_StrCat((n_res), (n_message)));
        n_i = ((n_i) + (1));
    }
    return (n_res);
}
\end{verbatim}
nanoLang and its translation (2)

```plaintext
Integer main(String arg1, String arg2)
{
    Integer limit;
    limit = 10;

    if (StrIsInt(arg1) == 1) {
        limit = StrToInt(arg1);
    }

    printf(testfun(limit, arg2);
    printf("\n");

    return 0;
}

N_Integer n_main(N_String n_arg1, N_String n_arg2)
{
    N_Integer limit = 0;
    n_limit = (10);
    if ((n_StrIsInt(n_arg1)) == (1)) {
        n_limit = (n_StrToInt(n_arg1));
    }

    printf("%s", NANO_STRVAL(n_testfun(n_limit),
                            (n_arg2)));

    printf("%s", NANO_STRVAL((NANO_MAKESTR("\n")));
    return (0);
}
```
/* C main function */
int main (int argc, char *argv[])
{
    N_String arg1 = NANO_MAKESTR("" );
    if (1 < argc ) {
      arg1 = NANO_MAKESTR(argv[1]);
    }
    N_String arg2 = NANO_MAKESTR("" );
    if (2 < argc ) {
      arg2 = NANO_MAKESTR(argv[2]);
    }
    n_main(arg1, arg2);
}
Top-Down Parsing
Basic Idea of Recursive Descent

- One parsing function per non-terminal
- Initial function corresponds to start symbol
- Each function:
  - Uses an oracle to pick the correct production
  - Processes the right hand side against the input as follows:
    - If the next symbol is a terminal, it consumes that terminal from the input (if it’s not in the input: error)
    - If the next symbol is a non-terminal, it calls the corresponding function

Oracle: Based on next character to be read!
- Good case: Every production can be clearly identified
- Bad case: Common initial parts of right hand sides ➔ ?
Example/Exercise

- Consider the following productions from $G_1$:
  - $S \rightarrow aA$
  - $A \rightarrow Bb$
  - $B \rightarrow aA$
  - $B \rightarrow \epsilon$

- What is the language produced?
- How can we parse $aabb$?
- What happens if we use the following productions from $G_2$?
  - $S \rightarrow aSb$
  - $S \rightarrow ab$

- Productions in $G_2$ have **common prefixes**
  - Common prefixes make the oracle work hard(er)
  - How can we get $G_1$ from $G_2$?

Left factoring! **Plus. . .**
Definition: A grammar $G = \langle V_N, V_T, P, S \rangle$ is left-recursive, if there exist $A \in V_N$, $w \in (V_N \cup V_T)^*$ with $A \xrightarrow{+} Aw$.

- Left recursion leads to infinite loops in recursive descent parsers
  - To parse $A$, we first need to parse $A \ldots$
- Solution: Reformulate grammar
We will again consider the set of well-formed expressions over $x, +, *, (, )$ as an example, i.e. $L(G)$ for $G$ as follows

- $V_N = \{ E \}$
- $V_T = \{(, ), +, *, x\}$
- Start symbol is $E$
- Productions:
  1. $E \to x$
  2. $E \to (E)$
  3. $E \to E + E$
  4. $E \to E * E$
We will again consider the set of well-formed expressions over $x, +, *, (, )$ as an example, i.e. $L(G)$ for $G$ as follows

- $V_N = \{E, T, F\}$
- $V_T = \{(, ), +, *, x\}$
- Start symbol is $E$
- Productions:
  1. $E \rightarrow E + T$
  2. $E \rightarrow T$
  3. $T \rightarrow T * F$
  4. $T \rightarrow F$
  5. $F \rightarrow (E)$
  6. $F \rightarrow x$

What happens if we want to parse this using recursive descent?
Exercise: Recursive Descent for Expressions

Consider the following productions:

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T \ast F \)
4. \( T \rightarrow F \)
5. \( F \rightarrow (E) \)
6. \( F \rightarrow x \)

Can we find an equivalent grammar that can be top-down parsed?

How?
Extend your compiler project to generate a basic C program:

- Finish the basic nanoLang compiler
  - Your program should produce a correct C program that compiles and implements the nanoLang semantics
  - A testprogram ("testprog3.nano") is on the web site.

- Bonus: Implement full string functionality (including automatic memory management and garbage collection)
Feedback round

- What was the best part of today’s lecture?
- What part of today’s lecture has the most potential for improvement?
  - Optional: how would you improve it?
Goals for Today

▶ Training exam
▶ Solution discussion
Review: Goals for Today

- Training exam
- Solution discussion
Feedback round

▶ What was the best part of the course?
▶ Suggestions for improvements?