Compilers

Feature overview

2nd Bachelor Computer Science
2023 – 2024

Kasper Engelen
kasper.engelen@uantwerpen.be

1 Introduction

In this document you will find an overview of all mandatory and optional features. For more information about a feature, please consult the assignment for that project. All mandatory features are expected to be implemented. Optional features will only be graded if all mandatory features are implemented. The implemented features are graded separately for LLVM IR and MIPS.

In principle every point listed below counts as one point, unless otherwise specified. For a feature to be graded, it needs to be fully implemented across the **entire pipeline of the project**: lexing, grammar, parsing, AST, error checking, and code generation.

2 Mandatory features

2.1 Project 1

- Binary operations `+`, `-`, `*`, and `/`.
- Binary operations `>`, `<`, and `==`.
- Unary operators `+` and `-`.
- Parenthesis to overwrite the order of operations.
- Logical operators `&&`, `||`, and `!`.
- Comparison operators `>=`, `<=`, and `!=`.
- Binary operator `%`.
- Shift operators `<<`, `>>`.
- Bitwise operators `&`, `|`, `~`, and `^`.
- Abstract syntax tree and visualisation
- Constant folding (e.g., a sub-tree of the AST "3+4" gets replaced by a single node "7")

2.2 Project 2

- Add an `int main() { ... }` function.
- Extra reserved keywords need to be supported: `const`, `char`, `int`, and `float`.
- Extra literals: float, integer, character. Scientific notation is not necessary.
- Variables: declarations, definitions, assignments, identifiers in expressions.
- Pointers: declaration, definition, operators `*` (dereference) and `&` (address). Smart pointers are not necessary.
- Const variables. For pointers, only "pointer to const" needs to be supported. The type is `const int*`. This means the value pointed to is const. The variable itself can be re-assigned with a different address.

- Implicit conversions. We consider the order `float isRicherThan int isRicherThan char`. Pay attention to warnings.
- Explicit conversions (i.e., the cast operator). Any conversion done this way should not cause a warning.
- Pointer arithmetic (all of the following need to be implemented):
  - Assignment: `p = 0`,
  - Addition and subtraction: `p - q`, `p + 2`, `q - 5`, etc. Take the size of the datatype into account!
  - Increment, decrement: `p++`, `q--`, etc.
  - Comparison: `p < end`, `p >= 0`, `p == 0`, etc.

  Note: this is only relevant for arrays.
- Increment/Decrement Operations: `i++`, `i--`. Both suffix and prefix variants.
- Constant propagation.
- Syntax errors
- Semantic error: Use of undeclared var
- Semantic error: Redeclaration of an undeclared var
- Semantic error: Operations or assignments of incompatible types
- Semantic error: Assignment to an rvalue
- Semantic error: Assignment to a const variable
- Symbol table and visualisation

2.3 Project 3

- Comments: Store comments in the AST, and add them to LLVM IR and MIPS code at the appropriate line.
- Comments: Add the original C-code as a comment in the LLVM IR and MIPS code.
- Typedefs
- **Note**: the PDF for assignment 3 mentions a printf function. This is temporary, however, and should not be present in the final version of the compiler.

2.4 Project 4

- Conditional statements: `if` and `else` must be supported. You can assume that curly braces are required.
- Loops. You have to implement `while`, `for`, `break`, and `continue`.
- Scopes: anonymous, if-else, for, while.
- Switch statements: `switch`, `break`, `case`.
- Your compiler should support enumerations.
- Support scopes in symbol table, semantic analysis, constant propagation

2.5 Project 5

- Function scopes.
- Local and global variables.
- Functions: definitions; declarations; calling, recursive calls; parameters (const, float, int, pointers); returns; void functions;
- Pre-processor: `#define`. Parameters do not need to be supported.
- Pre-processor: `#include`.
- Semantic analysis: Functions and their scopes in the symbol table
- Semantic analysis: Consistency of return type.
- Semantic analysis: Parameter types passed to call should be consistent with the function signature.
- Semantic analysis: Consistency between forward declarations and function definitions. Functions must be declared/defined before calling them. Functions should not be re-defined.

- Optimisations (1pt): no code after break, continue, or return.

2.6 Project 6

- Arrays: simple one dimensional arrays
- Arrays: multi-dimensional arrays.
- Arrays: array initialisation: `int arr[3] = {1,2,3}`.
- Arrays: Operations on array elements.
- Strings encoded as zero-terminated char-arrays. String literals. Passing strings around as `char*`.
- Support for IO: `printf` and `scanf` that support `char*` strings. The header `stdio.h` is treated as a special instruction that makes `printf` and `scanf` available. Including the actual `stdio.h` header is not necessary.
- Semantic analysis: Type checking for array types (parameter passing, assignment, indexing).
- Semantic analysis: Checking the length of array initialisers (`{ ... }`) during assignment.

2.7 Project 7

- User-defined structs. The structs should support members with primitive fields, arrays, enum types, as well as pointer types. You do not have to support default values for the struct members.
- Semantic analysis: Type checking for accessing and assigning struct members.
- Semantic analysis: Type checking for accessing and assigning union members (if implemented).
- Semantic analysis: Type checking for function pointers: assigning function pointers, calling functions, etc. (if implemented)

3 Optional features

3.1 Project 2

- Const casting: make it possible to have a non-const pointer to a const value:

3.2 Project 4

- `else if` statements.

3.3 Project 5

- Overloading of functions on the amount and types of the parameters.
- Prevent headers from being included twice by implementing include guards.
- Semantic analysis: Check for all paths in a function body whether a return statement is present.
- Optimisation: Do not generate code for unused variables
- Optimisation: Do not generate code for conditions that are always false.

3.4 Project 6

- Dynamic arrays (stored on the heap).

3.5 Project 7

- Structs that contain other structs as a value.
- Dynamic allocation of structs (stored on the heap).
- Unions.
- Function pointers. All the checks and errors that apply to ordinary pointers should also be implemented for function pointers.

- File reading using `fgets`. You can implement file pointers (`FILE*`) together with `fopen` and `fclose`, but this is not required. You may also just pass filenames to `fgets`.
- File writing using `fputs`. You can implement file pointers (`FILE*`) together with `fopen` and `fclose`, but this is not required. You may also just pass filenames to `fputs`.
- Dynamically allocated strings and character buffers. These can work in combination with `printf`, `fgets`, `fputs`, etc.