Taller de Diseño de Software

2022

Jeremías Parladorio - Franco Agustín Nolasco

1. Características

Diseño

- Tratando de hacer un diseño mantenible separamos las estructuras de datos en distintos directorios, también separamos por funcionalidad, ya sea construcción de símbolos, generación de código, etc.
- Tratamos de hacer el recorrido de los árboles lo más genérico posibles, usando punteros a funciones, contando actualmente con una función para realizar chequeo de tipos, generación de instrucciones 3D y una para evaluar (la cual fue descontinuada después de comenzar con el proyecto real).

Testing/CI

- Decidimos desde el comienzo llevar un archivo Makefile para hacer el build del proyecto, lo cuál nos facilitó y aceleró considerablemente el proceso de desarrollo y prueba (*make npc* y *make test*).
- Decidimos testear nuestro compilador a través del framework de testing *pytest*. Dividimos nuestra suite en 2 partes:
    1. Programas válidos: son el conjunto de programas sintáctica y semánticamente correctos. Sobre estos programas testeamos (1) que compilen exitosamente (sin output de error) y (2) que al ejecutar el código assembly generado, obtengamos el resultado esperado.
    2. Programas inválidos: son el conjunto de programas que poseen al menos un error sintáctico o semántico. Sobre estos programas testeamos (1) que la compilación no sea exitosa (i.e., que falle el proceso de compilación, retornando un exit status igual a 1) y (2) que el mensaje de error generado sea el esperado de acuerdo al error inyectado en el programa.
- Utilizamos integración continua (CI) con el objetivo de usar nuestra suite de tests para hacer testing de regresión: cada vez que hacemos un *git push* a nuestro repositorio, volvemos a correr todos los tests del compilador más todos los tests correspondientes a los archivos que fueron modificados en el commit. Esto nos permite saber si estamos introduciendo algún error con nuestras modificaciones de manera automática.

Error handling

- Pusimos un gran énfasis en el manejo y la comunicación de los errores capturados por nuestro compilador. Nuestros principales objetivos fueron:
    1. Informar la mayor cantidad posible de errores por cada intento de compilación. Comenzamos informando sólo el primer error encontrado, pero

luego cambiamos nuestro approach para poder informar todos los errores posibles. Actualmente sólo terminamos el proceso de compilación (en caso de haber error) al comenzar el proceso de generación de código, o bien cuando encontramos ciertos errores sintácticos que no son prácticos de acumular (e.g., si se escribe *int* en lugar de *integer*).

2. Ser lo más descriptivos posibles a la hora de informar dichos errores. Algunos de los mensajes de error que se proveen son:
    - Unknown symbol '<x>'.
    - Cannot use reserved word '<x>' as an identifier.
    - Invalid identifier ‘<x>’.
    - Identifier '<id>' is trying to be re-declared.
    - Arithmetic expressions do not accept bool type operands.
    - Expected <type-1> but <type-2> was found.
    - Undeclared identifier '<id>' .
    - Expected <type-1> but <type-2> was found for <procedure> in the *i*th parameter.
    - Too many arguments for <procedure>.
    - Arguments missing for <procedure>.
    - Boolean expressions do not accept integer type operands.
    - Identifier '<id>' from procedure '<procedure>' is trying to be re-declared.
    - Missing return statement in function <procedure>.
    - Could not find a function main.
    - Cannot pass arguments to main function.
    - Function main() can only return integer or void types.
    - Unknown symbol '<x>'. Did you mean '<y>'?

    Con respecto a este último error, nuestra idea fue ofrecer sugerencias de corrección para errores sintácticos simples. Por ejemplo, algo que surgió en varias ocasiones fue escribir ‘Program’ en lugar de ‘program’ o ‘boolean’ en lugar de ‘bool’. En estos casos, nos dimos cuenta que podíamos tomar el lexema no reconocido (e.g., ‘boolean’) y calcular la distancia de Levenshtein con todos los lexemas válidos, y dar como sugerencia aquel que tenga la menor distancia de edición con el lexema no reconocido, en este ejemplo, ‘bool’.

2. Dificultades

De diseño

- Agregar recursión fue una de las cosas que se nos hicieron un poco más complicadas pues teníamos que definir el símbolo del procedimientos antes de procesar el bloque del mismo, de forma tal que el procedimiento fuera visible dentro de su propio bloque. Si bien la idea parece simple, hubo que lidiar con detalles en la implementación que lo hicieron un poco complejo en su momento.
- Cuando decidimos pasar de informar sólo el primer error a informar todos, surgieron muchos problemas, principalmente de *segmentation fault.* Esto nos implicó el uso de un debugger (gdb) para facilitar el proceso de descubrimiento de errores, en el cuál se hizo evidente que había que hacer cambios considerables ya que desde el inicio nuestro diseño no consideró la posibilidad de “arrastrar” ciertos errores (e.g., crear

símbolos re-declarados) con el objetivo de poder seguir detectando errores más allá del primero.

De desarrollo

- Por cuestiones de incompatibilidad en arquitectura de procesadores (x86 vs. ARM), al momento de empezar a generar código assembly tuvimos que mudar el entorno de desarrollo a un container Docker corriendo Ubuntu. Esto por momentos ralentizó el proceso de prueba.

3. Limitaciones

- Tipos Básicos: contamos solo con tres tipos básicos: integer, bool y void. Esto limita mucho las cosas que podemos realizar con nuestro lenguaje. Agregar un tipo char no debería ser muy complejo. El tipo de datos float es fundamental para resolver problemas matemáticos interesantes.
- Abstracciones: no contamos con la posibilidad de definir estructuras de datos, ya sea arreglos o estructuras definidas por el usuario (struct de C u objetos). Esto impide definir colecciones como listas, pilas, colas, árboles, grafos, etc.
- Parámetros: contamos solo con la posibilidad de pasar hasta seis parámetros a un procedimiento, si bien (en nuestra opinión) no es una buena práctica tener procedimientos con muchos parámetros pueden existir soluciones a problemas que así lo requieran y no estaríamos contando con ello.
- Offset: en la versión actual del compilador estamos usando un offset nuevo cada vez que hay alguna declaración de un símbolo de tipo ID (de variables o procedimientos) o de tipo parámetro, al igual que también con las operaciones y llamadas a procedimientos. Lo primero que deberíamos hacer es que cada procedimiento tenga su propia generación de offset, de esa forma podemos repetir offsets entre distintos procedimientos sin generar problemas. Otro lugar donde podemos recortar offset en los valores temporales. Haciendo análisis estático podemos ver cuántas variables se utilizarán en un procedimiento, entonces a la hora de generar los símbolos para subexpresiones podemos hacer que cada expresión sea independiente de la otra, pudiendo usar el mismo offset para los cómputos parciales siempre y cuando estos formen parte de distintas expresiones.
- Recursión indirecta: actualmente no podemos realizar recursión indirecta debido a que la definición de un procedimiento sucede inmediatamente después de su declaración, por lo cual no podríamos declarar varios procedimientos para después en sus definiciones referirlos entre sí.