CS3025 Compiladores

Tarea de Programación 2

Salida: 22/09/2023 Entrega: 29/Septiembre/2023 5pm

Grupos de 3 (maximo) Puntos: 6.5

Objetivo: Implementar un analizador sintactico (parser) para el lenguaje de la máquina de pila SM definido en la primera sección. Este lenguaje es el mismo (con 1 o 2 modificaciones) que el que se usó en la primera tarea. El parser deberá realizar el análisis sintactico y construir un programa para la máquina virtual SVM mediante el uso de acciones semanticas.

Se provee:

- Codigo del scanner y esqueleto del parser a implementar (`svm_parser.hh` y `svm_parser.cpp`).
- SVM: Un intérprete de programas SM, implementado por los archivos `svm.hh` y `svm_cpp`.
- El archivo `test_run.cpp` para realizar pruebas.

1. Sintaxis de SM

El lenguaje de máquina de pila SML está definido por la siguiente gramática:

<program> ::= <instruction>+
<instruction> ::= Label? (<instr0> | <instr1> <num> | <instr2> <id>) <eol>
<instr0> ::= **skip** | **pop** | **dup** | **swap** | **add** | **sub** | **mul** | **div** | **print**
<instr1> ::= **push** | **store** | **load**
<instr2> ::= **jmpeq** | **jmpgt** | **jmpge** | **jmplt** | **jmple** | **goto**

Las instrucciones se han dividido en tres tipos, aquellas que no tienen argumento, aquellas que usan un argumento entero, y las que usan un label (id) como argumento.
Las instrucciones pueden llevar un Label o etiqueta opcional (al comienzo de la instrucción).
Todas las instrucciones deben acabar en EOL.

La especificación léxica, **implementada por el scanner**, está dada por los siguientes patrones

digito ::= [0-9]
carácter :: = [a-zA-Z]
<palabra-reservada> ::= **push** | **jmpeq** | **jmpgt** | **jmpge** | **jmplt** | **jmple** | **goto** | **pop** | **dup** | **swap** | **add** | **sub** | **mul** | **div** | **store** | **load** | **print** | **skip**
<id> ::= carácter | (caracter | digito | '_')*
<label> ::= <id> :
<num> ::= digito+
<eol> ::= '\n' +
<ws> ::= (' ' | '\t')+

Las unidades léxicas <id>, <num>y <eol> , y todas las palabras reservadas, generan un token. El patrón <ws> no genera token - los espacios en blancos son ignorados.

Además se genera un token ERR especial para reportar errores. Los tokens ERR, NUM, ID y LABEL contienen sus lexemas completos, con la excepcion de LABEL, que remueve el ':´al final. Los tipos de token serán definidos por `enum Type`.
Todo esto esta implementado en `svm_parser.cpp`.

2. La maquina virtual SVM

La máquina virtual SVM esta implementada por los archivos `svm.hh` y `svm.cpp`.
Esta máquina ejecuta programas (listas de instrucciones especificadas por la gramática de arriba) efectuando operaciones por intermedio de una pila de datos (enteros). Toda operación que necesita un argumento o genera un resultado, interactua con la pila:

- `push n`: coloca al entero n arriba de la pila.
- `pop`: remueve el elemento arriba de la pila
- `dup`: duplica el elemento de arriba de la pila
- `swap`: intercambia los dos elementos de arriba de la pila
- `print`: imprime los elementos de la pila (para debugging)
- `add, sub, mul` y `div` efectúan las respectivas operaciones aritmeticas con los dos últimos elementos de la fila. Los 2 operandos son removidos de la pila y el resultado se coloca en su lugar. El orden es importante. Si llamamos `top` al elemento de arriba y `next` al siguiente, la ejecución de `sub` dejara arriba de la pila el resultado de `next – top.`
- `goto label` salta a la instrucción etiquetada por `label`.
- `jmpeq label, jmplt label, jmple label, jmpgt label, jmpge label` saltan a la instrucción etiqueta por `label` si la operación de comparación es exitosa. Las mismas reglas para el orden de operadores usadas con expresiones aritmeticas aplican.
  Es decir, si usamos p.ej. `jmplt` efectuamos la comparación `next < top`.
- `skip` no tiene ningún efecto.

Ademas la SVM provee 8 registros, numerados del r=0 al 7, que pueden ser leidos o escritos con:

- `store r`: remueve el valor encima de la pila y lo guarda en el registro r.
- `load r`: coloca el valor guardado en registro r encima de la pila.

Todo esto esta implementado `svm.hh` y `svm.cpp`.
Podemos realizar pruebas mediante el programa `svm_run.cpp`. Se compila:

```
>> g++ svm_run.cpp svm.cpp svm_parser.cpp
```

Este programa puede crear programas svm explicitamente via constructores (con el flag useparser=false) o llamando al parser. La versión actual llama a constructores, imprime el programa generado, lo ejecuta y finalmente imprime la pila final.

```
>>> ./a.out
```

Hay un segundo “programa” comentado (programa 2). Que hace?
Los archivos ejemplo1.svm y ejemplo2.svm contienen las versión “de texto” de estos programas.

3. QUE HACER: El Parser

Si compilamos `svm_run.cpp` con `useparser=true` podemos tomar como argumentos programas SVM. Por ejemplo, al ejecutar:

```
>>> ./a.out ejemplo0.svm
```

Obtenemos

```
Reading program from file ejemplo0.svm
Program:
pop
add
add
----------------
Running ....
error: Can't pop from an empty stack
```

El parser solo puede realizar análisis sintácticos de un pequeño subconjunto del lenguaje. En el caso del ejemplo0.svm, al parser pudo crear un program SVM pero este programa genero errores al ejecutarse (no podemos hacer pop a una pila vacia!). Tenemos que modificar el parser para aceptar y generar todos los programas posibles de acuerdo con la sintaxis.

Se pide:

a) (1.5 pt) Modificar el scanner (en `svm_parser.cpp`) para aceptar comentarios de una sola línea que empiecen con ‘%´.
b) (4 pts) Modificar el parser (en `svm_parser.cpp`) para analizar todas las instrucciones validas en SVM.
    a. `parseInstruction`() deberá un objeto `Instruction`.
    b. Modificar la funcion `tokenToIType` para considerar todos los tokens.
c) (1 pt) Completar el archivo `factorial.svm` con una implementación de la función factorial. El programa deberá calcular el factorial del numero arriba de la pila – el numero colocado por la instrucción `push`.

Entregar todos los archivos (los mismos incluidos aca, con las modificaciones indicadas) en un zip o indicar el repositorio usado. Pueden incluir archivos extra de ejemplos de código svm.