UNIVERSIDADE DE ÉVORA

Trabalho Final de Compiladores

Yaroslav Kolodiy `139859` Eduardo Medeiros `139873`

Junho, Ano Letivo 2019/2020

Compiladores

Prof. Pedro Patinho

Índice

1 Introdução

O objetivo deste trabalho consiste em implementar um compilador para a linguagem *Ya!*. Esta implementação foi dividida em quatro partes sendo elas respectivamente: analisador lexical, analisador sintático, analisador semântico e gerador de código.

Através do conhecimento adquirido pelo grupo ao longo do semestre tentamos implementar a melhor maneira possível usando a linguagem C e as ferramentas bison e flex.

2 Analisadores

2.1 Lexical

O analisador lexical tem como objetivo "partir"o código escrito em *Ya!* em partes/tokens, i.e. em elementos específicos. Estes tokens podem ser palavras reservadas da linguagem (and, not, if, etc...), ou expressões regulares. Estas últimas servem, por exemplo, para identificar números, ids ou strings. A ordenação destes tokens, reservadas e regex, vai ser importante nos analisadores seguintes.

O números/literais são identificados através das seguintes expressões regulares (regex):

- INT [0-9]+
- FLOAT [0-9]*[̇0-9]+

Um id é qualquer conjunto de caracteres que possa ser gerado pela seguinte regex:

- ID [a-zA-Z][a-zA-Z_0-9]*

Uma string é qualquer conjunto de caracteres que não seja "nem \n pois todas as string aparecem entre aspas. Tal pode ser representado da seguinte maneira:

- STRING [^¨\n]*

2.2 Sintático

O analisador sintático, de um modo mais abstrato consiste em verificar se os tokens provenientes do analisador lexical estão por uma determinada ordem.

Esta ordem é consiste numa gramática. O analisador verifica se o uma determinada sequência de tokens unifica com alguma produção de uma determinada regra da gramática. Se tal não acontecer é porque foi encontrado um erro sintático.

Caso não exista erro, há medida que se percorre a gramática vai sendo gerada a Abstract Parse Tree (APT) com os nós respectivos a cada produção.

As regras/não terminais escolhidos foram:

- program
- decls

- decl
- argdefs
- argdef
- args
- ids
- stms
- stm
- type
- lit
- exp

Estas regras/não terminais foram escolhidas tendo em conta o descrito no enunciado.

Os nós da APT têm nomes idênticos ao dos não terminais, e.g. t_decls. Para cada produção existe uma função específica que é chamada quando uma sequência de tokens unifica com a mesma. Cada função gera um tipo de nó da APT diferente.

2.3 Semântico

Agora que o programa origianal já foi dividido em tokens e os mesmos usados apra gerar a APT, no processo de análise semântica esta é percorrida e vai sendo gerada a Symbol Table (ST). A ST é usada para verificar se os "requisitos"da linguagem são cumpridos, i.e. se o código *Ya!* está de acordo com as regras semânticas.

Exemplos de erros semânticos:

- a:int = "string";
- b:string = true;

Na linguagem *Ya!* apenas podem existir dois ambitos/scopes simultaneamente, um global e um local da função em que se está. Mesmo existindo esta restrição da linguagem, a ST consiste numa linked list cujo conteúdo de cada nó é uma hashtable, deste modo é possível escalar caso sejam pretendidos mais ambitos/scopes. O conteúdo das hashtables são structs ST_Data que por sua vez, guardam o tipo da variável ou função, e o offset e o tamanho das mesmas respectivamente. No entanto, cada ST_Data, apenas pode guardar uma variável ou uma função, nunca as duas simultaneamente.

As funções implementadas para manipular a ST foram, *ST_new_scope*, *ST_discard*, *ST_insert* e *ST_lookup*. As duas primeiras funções permitem criar e eliminar um ambito/scope, ou seja, criar ou eliminar um nó da linked list. As duas outras consistem, respectivamente, em inserir na hashtable do nó atual um novo elemento, variável ou função e em procurar, dando o id, uma variável ou função em toda a ST, se não tiver no nó atual da linked list passa para o seguinte até que chegue ao fim e não tenha mais ambitos onde procurar, dois no caos do *Ya!*.

3 Gerador de Código

Após o programa ter passado pelos analisadores anteriores, sem manifestar qualquer erro, na parte da geração de código vai ser feita uma travessia pela APT (que está correta de acordo com as implementações anteriores) e será gerado o código MIPS para cada nó. Para cada tipo de nó presente na APT, o código MIPS gerado vai ser diferente pois os comportamentos pretendidos dos mesmo também são diferentes. Estes comportamentos, caso manipulem a pilha, devem devolvê-la tal como foi recebida, ou seja, qualquer alteração deve ser reposta. O resultado da geração de código de cada nó deve ser retornado no registo $t0.

4 Conclusão

Com este trabalho podemos aprofundar os nossos conhecimentos acerca de um funcionamento de um compilador e dos processos que o mesmo usa, desde que lhe é fornecido um programa até que gera um executável do mesmo.

Neste trabalho existem certos detalhes que não foram possíveis de implementar. Os exemplos mais notórios são, o não tratamento correto de strings e arrays, uma vez que os comportamentos em código MIPS são "semelhantes"e nenhum foi implementado corretamente, e funções, estas não os guardam os seus argumentos.

4.1 Compilação e Execução

O trabalho deve ser compilado através do comando

$$make$$

e corrido através do comando

$$./yac$$