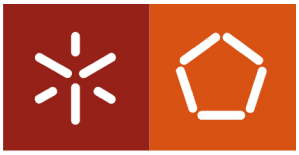

Universidade do Minho

LEI — Licenciatura de Engenharia Informática

Processamento de Linguagens

Compilador de uma LPS

Orlando Costa - a67705, Paulo Araujo - a58925, Rui Oliveira - a67661



Braga, 1 de Abril de 2015

Resumo

Este relatório descreve o desenvolvimento de um compilador para uma linguagem de programação simples (lps).

A linguagem desenvolvida foi baseada no C, e suporta:

- Variáveis globais
- Ciclos: for, while, do while
- Estruturas de Condição: If .. Else
- Expressões Aritméticas e lógicas
- Funções com argumentos
- Declaração de variáveis locais dentro das funções

O compilador foi desenvolvido com recurso ao analisador léxico Flex e ao analisador sintático Yacc.

Conteúdo

Capítulo 1

Introdução

O presente trabalho enquadra-se na unidade curricular de Processamento de Linguangens da Licenciatura em Engenharia Informática da Universidade do Minho. O trabalho pretende aumentar a experiência em engenharia de linguagens,

1.1 Linguagem de programação imperativa simples

Previamente ao desenvolvimento do compilador existe a necessidade de definir uma linguagem sobre a qual este atua, com base numa qualquer linguagem imperativa. Neste sentido e por simplicidade e familiaridade, a linguagem de programação C é a selecionada. Esta linguagem foi simplificada por forma a adaptar-se aos requisitos propostos, sofrendo as seguinte modificações na sua estrutura:

- Apenas permite manusear variáveis do tipo inteiro (escalar ou array). - Suporta apenas as instruções vulgares de controlo de fluxo de execução (condicional e cíclica), tais como if-else, for, while e do-while. - As instruções que controlam inserção e output de valores (tipicamente printf e scanf) estão adaptadas para suportar apenas inteiros, e então estão renomeadas (printi e scani). - As expressões lógicas devem estar rodeadas por parentises para facilitar a sua distinção e ordem quando em conjunto com expressões aritméticas.

<Programa exemplo (programa exemplo que mostre de forma simples todas as funcionalidades - funçoes, atribuiçoes, ciclos, expressoes logicas e aritmeticas)>

1.2 Arquitetura

O sistema desenvolvido é principalmente constituído por 2 modelos: parser.l, compiler.y, que são respetivamente o analisador léxico e analisador sintático.

Na Figura 1.1, podemos observar as dependências entre os ficheiros.

O analisador sintático utiliza o ficheiro vmCompiler.h, sendo que este o modulo responsável pelo tratamento das variáveis e funções existentes (adicionar/consultar variáveis).

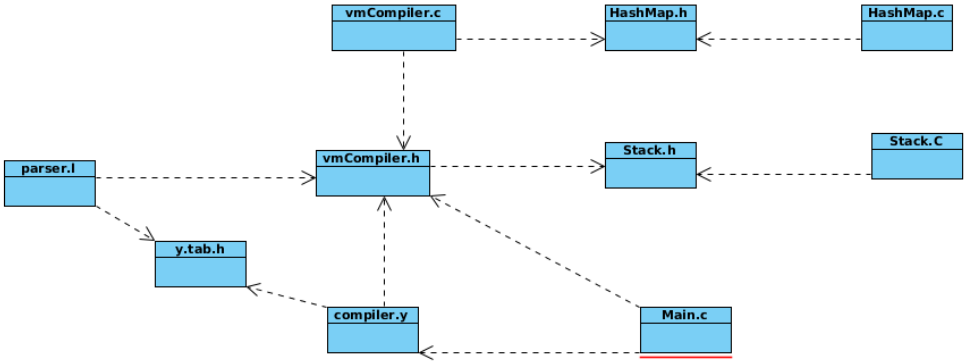

Figura 1.1: Diagrama das pedendencias dos ficheiros

Figura 1.2: Diagrama das estruturas usadas em vmCompiler

O Sistema utiliza também duas estruturas de dados: uma HashMap e uma Stack. A hashmap é utilizada respetivamente para guardar as variáveis e as funções, enquanto que a stack permite o controlo das labels dos ciclos durante a compilação.

Na figura 1.2 é possível observar as estruturas utilizadas em vmCompiler:

- Scope - possuem uma map com a informação das variáveis, onde a chave é nome da variável e o valor é um EntryVar;
- EntryFun - guarda a informação sobre o tipo de uma função (argumentos de entrada e tipo de retorno);
- EntryVar - guarda o tipo, nome o endereço relativo de uma dada variável;

Com as estruturas anteriores em mente, as variáveis criadas em vmCompiler são as representadas na figura 1.3, onde se podem ver duas variáveis do tipo Scope, uma para o contexto global e outra para o contexto interior a uma função.

Existe um map de EntryFun (mFuncMap) onde a chave é o nome da função.

A variável DecFunAux consiste num apontador temporário para uma função declarada.

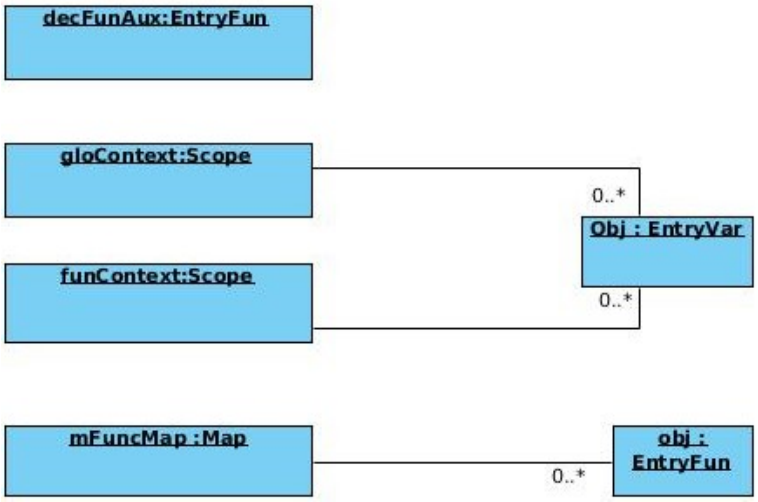

Figura 1.3: Diagrama dos objetos existente em vmCompiler

1.3 Estruturas de dados

1.3.1 Stack

De forma a evitar confusão na atribuição de *labels* relativas a *ifs* e *loops* é utilizado um contador de condições. À medida que é encontrada uma instrução que implique o uso de uma condição, este contador é incrementado e o seu valor é colocado numa stack. Deste modo, o valor que se encontra no topo da stack é relativo ao último *ciclo/if* encontrado. Sempre que é encontrado o final de uma condição, o valor no topo da stack é removido. Através do uso de um contador e de uma stack, é muito mais simples gerir as *labels* e as operações de controlo, como *JUMPs* e *JZs*. A stack utilizada implementa apenas as funções necessárias para a sua inicialização, inserção, remoção e consulta. Com as operações de push/pop são inseridos/removidos valores no topo da stack, e com a operação de get, apenas é consultado o valor no topo da stack, sem que este seja removido. Esta última operação é útil para a geração de instruções 'JZ' na geração de código VM.

1.3.2 HashMap

Capítulo 2

Compilador

2.1 Analisador léxico

O analisador léxico construido deteta todos os símbolos terminais da linguagem (palavras reservadas, sinais e variáveis). É de se destacar a deteção de comentários, que são ignorados. De forma a que fosse mais fácil detetar os erros de sintaxe, o parser conta as linhas que já interpretou. Desta forma o analisador sintático quando dá erro diz a linha onde o erro aconteceu. Para passar valores como o nome de variaveis ou números utiliza-se o yylval.

2.2 Analisador sintatico/semantico

As funções são declaras depois das variáveis globais para as funções terem acesso às variáveis globais.

2.3 Geração de codigo máquina

A cada regra da gramática, associamos acções a serem executadas à medida que estas são reconhecidas. Assim sendo, é feita uma tradução da linguagem desenvolvida para a linguagem *assembly* da VM, à medida que cada instrução ou expressão é identificada. A maioria destas acções implica uma instrução de escrita no ficheiro de output. Todas estas acções que implicam escrita no ficheiro são triviais, existindo apenas algumas excepções como o caso do ciclo 'for'. No ciclo 'for', foi necessária a utilização de instruções 'JUMP', de forma a ser possível seguir o seu fluxo de execução normal. Após a identificação e execução das acções associadas à expressão lógica presente no ciclo 'for', é gerado o salto condicional respectivo, assim como um salto para as instruções associadas ao corpo do ciclo. Além disso, é gerada uma *label* que irá corresponder ao incremento do ciclo que irá ser identificado de seguida. Identificado o final do corpo do ciclo, é feito um salto para a *label* correspondente ao incremento do ciclo, que para além das respetivas instruções, conterá outro 'JUMP' para o teste da expressão lógica.

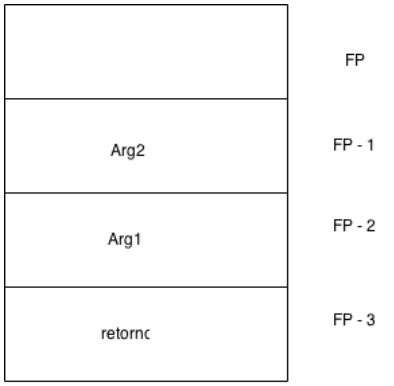

Figura 2.1: Chamada de uma função

2.3.1 Funções

Para a implementação de funções existiram algumas implicações que tiveram que ter ser resolvidas, teve-se que passar a lidar com mais do que um contexto, ou seja as variáveis podem ser globais ou locais.

As declarações das funções vem logo após às declarações das variáveis globais para que dentro das funções já se possa aceder às variáveis globais.

As declarações de variáveis são feitas no inicio da função, e no hash table das variáveis guarda-se não só o endereço mas também o contexto (ou seja se é local ou global). Com isto nos acessos ás variáveis utiliza-se 'PUSHG' ou 'PUSHL' mediante seja respetivamente variável global ou local.

A passagem de argumentos para a função é tratada como uma declaração especial em que o endereço é negativo. Já o retorno da função é colocado também num endereço negativo que foi previamente alocado na chamada da função.

Para perceber melhor vamos utilizar um exemplo, assumindo que estamos a chamar uma função com 2 argumentos. Como podemos ver na figura 2.1, os endereços são negativos ao fp, e o endereço onde a função colocará o retorno é $fp - 3$. Após a execução da função é feito o 'pop' dos argumentos e assim o valor de retorno da função está no topo da stack.

Capítulo 3

Testes

3.0.2 Teste 1

Input

Output

3.0.3 Teste 2

Input

Output

3.0.4 Teste 3

Input

Output

3.0.5 Teste 4

Input

```
// asd
int i;
int j;

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

i = 3;
j = 4;
```

```
i = fun(i + 2,j);
```

Output

```
START
PUSHI 0
PUSHI 0
JUMP init
fun:NOP
PUSHL -2
PUSHL -1
ADD
STOREL -3
RETURN
init:NOP
PUSHI 3
STOREG 0
PUSHI 4
STOREG 1
nPUSHI 0
PUSHG 0
PUSHI 2
ADD
PUSHG 1
CALL fun
POP 2
STOREG 0
STOP
```

3.0.6 Teste 5

Input

```
// asd
int i;
int j;

# int fun(int a) {
    int ret;
    if (a > 0) {
        ret = fun(a-1);
    }
    return ret;
}

printi(fun(i + 2));
```

Output

```
START
PUSHI 0
PUSHI 0
JUMP init
fun:NOP
PUSHI 0
PUSHL -1
PUSHI 0
SUP
JZ endCond1
nPUSHI 0
PUSHL -1
PUSHI 1
SUB
CALL fun
POP 1
STOREL 1
endCond1
PUSHL 1
STOREL -2
RETURN
init:NOP
nPUSHI 0
PUSHG 0
PUSHI 2
ADD
CALL fun
POP 1
WRITEI
STOP
```

Capítulo 4

Conclusão